VAX C Run-Time Library Reference Manual

Order Number: AI–JP84A–TE

March 1987

This document describes the functions and macros in the VAX C Run-time Library.

Revision/Update Information:	This is a new manual.
Operating System and Version:	VMS Version 4.6 or higher, or MicroVMS Version 4.6 or higher
Software Version:	VAX C Version 2.3

digital equipment corporation
maynard, massachusetts

First Printing, March 1987

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Contents

EXAMPLES

FIGURES

TABLES

Preface

This manual provides reference information on the VAX C Run-Time Library (RTL) functions and macros that provide I/O functionality, character and string manipulation, mathematical functionality, error detection, subprocess creation, system access, and windowing capabilities.

Intended Audience

This manual is intended for experienced and novice programmers who need reference information on the functions and macros contained in the VAX C Run-Time Library.

Structure of This Document

This manual describes the VAX C Run-Time Library. It provides information about portability concerns between operating systems and categorical descriptions of the functions and macros. This manual has twelve chapters and four appendixes. They are as follows:

- Chapter 1, VAX C Run-Time Library Information, provides an overview of the VAX C Run-Time Library.
- Chapter 2, Standard I/O Functions and Macros, explains the standard I/O functions and macros.
- Chapter 3, Terminal I/O Functions, discusses the terminal I/O functions.
- Chapter 4, UNIX I/O Functions and Macros, explains the UNIX I/O functions and macros.[1]
- Chapter 5, Character-Handling Functions and Macros, describes the character-handling functions and macros.
- Chapter 6, String- and List-Handling Functions and Macros, describes the list-handling functions and macros.
- Chapter 7, Math Functions, explains the math functions.

[1] UNIX is a registered trademark of American Telephone and Telegraph Company.

- Chapter 8, Error-Handling Functions, discusses the error-handling functions.
- Chapter 9, Memory Allocation Functions, explains the memory allocation functions.
- Chapter 10, Subprocess Functions, describes the subprocess functions.
- Chapter 11, System Functions, explains the system functions.
- Chapter 12, Curses Screen Management Functions and Macros, describes the Curses screen management functions and macros.
- Appendix A, VAX C RTL and RTLs of Other C Implementations, provides a comparison of VAX C RTL functions and macros, and corresponding functions of other C implementations.
- Appendix B, VAX C Run-Time Modules and Entry Points, provides a description of the VAX C modules and the VAX run-time modules used in this implementation.
- Appendix C, VAX C Definition Modules, describes VAX C definition modules.
- Appendix D, Syntax Summary, provides a summary of all the VAX C Run-Time Library functions and macros.

Associated Documents

You may find the following documents useful when programming in VAX C:

- *Guide to VAX C* — For programmers who need tutorial information on using VAX C.
- *VAX C Installation Guide* — For system programmers who install the VAX C software.
- *VMS Master Index* — For programmers who need to work with the VAX machine architecture or the VMS system services.

  This index lists manuals which cover the individual topics concerning access to VMS.

Conventions Used in This Document

Convention	Meaning
RETURN	The symbol RETURN represents a single stroke of the RETURN key on a terminal.
CTRL/X	The symbol CTRL/X , where letter X represents a terminal control character, is generated by holding down the CTRL key while pressing the key of the specified terminal character.
$ RUN CPROG RETURN	In interactive examples, the user's response to a prompt is printed in red; system prompts are printed in black.
float x; . . . x = 5;	A vertical ellipsis indicates that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.
option, . . .	A horizontal ellipsis indicates that additional parameters, options, or values can be entered. A comma that precedes the ellipsis indicates that successive items must be separated by commas.
[output-source, . . .]	Square brackets, in function synopses and a few other contexts, indicate that a syntactic element is optional. Square brackets are not optional, however, when used to delimit a directory name in a VMS file specification or when used to delimit the dimensions of a multidimensional array in VAX C source code.
sc-specifier ::= **auto** **static** **extern** **register**	In syntax definitions, items appearing on separate lines are mutually exclusive alternatives.

Convention	Meaning
[a|b]	Brackets surrounding two or more items separated by a vertical bar (|) indicate a choice; you must choose one of the two syntactic elements.
Δ	A delta symbol is used in some contexts to indicate a single ASCII space character.
switch statement **fprintf** function	Boldface type identifies language keywords and the names of VMS and VAX C Run-Time Library functions.
arg1	Italics identifies variable names.

New and Changed Features

VAX C Version 2.3 supports the following new VAX C Run-Time Library functions:

Functions in both the System V Interface Definition and the Proposed ANSI C Language Standard

- The **asctime** function—converts the broken-down time passed in a predefined structure form into a string.
- The **assert** function—verifies a program assertion.
- The **bsearch** function—performs a binary search on a sorted array.
- The **clock** function—determines the amount of CPU time used.
- The **div** function—returns the quotient and remainder after the division of its arguments.
- The **gmtime** function—converts calendar time into a broken-down time relative to GMT (Greenwich Mean Time).
- The **memchr**, **memcmp**, **memcpy**, **memmove**, and **memset** functions perform opererations on areas of memory.
- The **qsort** function—performs a quick sort.
- The **setvbuf** function—allows you to specify the I/O that is to be buffered.
- The **strtod**, **strtol**, and **strtoul** functions allow you to manipulate strings. Specifically, the **strtod** function allows you to convert a string to a double-precision number, and the **strtol** and **strtoul** functions allow you to convert a string to an integer or unsigned integer, respectively.
- The **strtok** function—extracts a token from a string by using a specified set of token delimiters.
- The **system** function—passes a command string to be executed by the command processor.
- The **vprintf**, **vfprintf**, and **vsprintf** functions—perform formatted output comparable to the **printf**, **fprintf**, and **sprintf** functions.

Functions Defined in the System V Interface Definition

- The **execlp** and **execvp** functions—pass the name of an image to be activated in a child process.
- The **getcwd** function—returns the current working directory.
- The **getppid** function—returns the parent process ID of the calling process.

Functions Defined in the Proposed ANSI C Language Standard

- The **atexit** function—establishes an action function to be called at program termination time.
- The **difftime** function—computes the difference between two calendar times.
- The **fmod** function—computes the floating-point remainder of x/y.
- The **remove** function—deletes a closed file. This function is equivalent to the **delete** function.
- The **rename** function—renames a closed file.
- The **strerror** function—returns a C RTL error message string corresponding to a C RTL error code.

Enhancements to Existing Functionality

- The **fopen** and **freopen** functions can now be used to open binary files when the access mode contains a "b" character string. The "b" string cannot appear in the first character position.
- The **printf** and **scanf** functions now perform formatted output and input respectively with the addition of two new format flags (#, +) and the following new format specifiers: i, p, and n.
- The **ungetc** function guarantees one character of push back at all times. This function is only valid on stream files. Two calls to the **ungetc** function with no intervening I/O is no longer supported.
- The **VAXC$CRTL_INIT** function now allows you to initialize the VAX C RTL for calling from other VAX languages where C is not the main program.

Chapter 1

VAX C Run-Time Library Information

Before using the VAX C Run-Time Library (RTL) of functions and macros, you must be familiar with:

- The linking process
- The macro substitution process
- The difference between function definitions and function calls
- The valid file specifications
- The VMS-specific methods of input and output (I/O)
- The VAX C-specific portability concerns

These topics may seem unrelated, but a knowledge of all these issues is necessary to using the VAX C RTL. This chapter shows the connections among these topics and the VAX C RTL, and should be read before any of the other chapters in this manual.

The primary purpose of the VAX C RTL is to provide a means for C programs to perform I/O operations; the C language itself has no facilities for reading and writing information. In addition to I/O support, the VAX C RTL also provides a means to perform many other tasks.

Chapters 2 through 12 contain descriptions of all the functions and macros for the various tasks supported by the VAX C RTL. Each chapter describes the functions and macros in a particular functional category. The functional categories and their associated chapters are as follows:

- Standard, Terminal, and UNIX I/O functions and macros (Chapters 2, 3, and 4, respectively)
- Character-handling functions and macros (Chapter 5)
- String- and list-handling functions and macros (Chapter 6)

- Mathematical functions (Chapter 7)
- Signal functions (Chapter 8)
- Memory allocation functions (Chapter 9)
- Subprocess functions (Chapter 10)
- System functions (Chapter 11)
- Curses Screen Management functions and macros (Chapter 12)

1.1 Implementation of the VAX C Run-Time Library

When working with the VAX C RTL, you must be aware of the specifics of this implementation.

First, if you plan on using VAX C RTL functions in your programs, make sure that a function named "main" or a function that uses the "main_program" option exists in your program. For more information, refer to the *Guide to VAX C.*

The VAX C Run-Time Library functions are executed at run time, but references to these functions are resolved at link time. When you link your program, the linker resolves all references to VAX C Run-Time Library functions by searching any object code libraries that you specified on the LINK command line. If the linker locates the function code, it places a copy of the code in the program's local program section (psect). If the linker does not locate the function code, it translates the logical names LNK$LIBRARY_n to the name of an object library and then searches that library for the code.

You must define the logical names LNK$LIBRARY_n as one or more of the following libraries:

- SYS$LIBRARY:VAXCCURSE.OLB
- SYS$LIBRARY:VAXCRTLG.OLB
- SYS$LIBRARY:VAXCRTL.OLB

Depending on the needs of your program, you may have to access one, two, or all three of the libraries. The following list relates the needs of your program with the particular libraries that you must define.

1. If you do not need to use the Curses Screen Management package of VAX C RTL functions and macros, and you do not use the /G_FLOAT qualifier on the CC command line, you must define the logical as follows:

```
$ DEFINE  LNK$LIBRARY  SYS$LIBRARY:VAXCRTL.OLB [RETURN]
```

2. If you do plan to use the /G_FLOAT qualifier with the CC command line, but do not plan on using Curses, you must define the logicals as follows:

```
$ DEFINE  LNK$LIBRARY SYS$LIBRARY:VAXCRTLG.OLB [RETURN]
$ DEFINE  LNK$LIBRARY_1 SYS$LIBRARY:VAXCRTL.OLB [RETURN]
```

3. If you do plan to use the Curses Screen Management package, but do not plan to use the /G_FLOAT qualifier, you must define the logicals as follows:

```
$ DEFINE  LNK$LIBRARY SYS$LIBRARY:VAXCCURSE.OLB [RETURN]
$ DEFINE  LNK$LIBRARY_1 SYS$LIBRARY:VAXCRTL.OLB [RETURN]
```

4. Finally, if you do plan to use both Curses and the /G_FLOAT qualifier, you must define the three logicals as follows:

```
$ DEFINE  LNK$LIBRARY SYS$LIBRARY:VAXCCURSE.OLB [RETURN]
$ DEFINE  LNK$LIBRARY_1 SYS$LIBRARY:VAXCRTLG.OLB [RETURN]
$ DEFINE  LNK$LIBRARY_2 SYS$LIBRARY:VAXCRTL.OLB [RETURN]
```

The order of the specified libraries determines which versions of the VAX C RTL functions are found first by the linker. If the linker does not find the function code or if LNK$LIBRARY_n is undefined, it assumes that the function is not a VAX C RTL function and checks other default libraries before it assumes that the program is in error. It may be helpful to place these definitions in your LOGIN.COM file or some other command procedure so that you do not have to retype these definitions each time you use the VAX C RTL.

For more information concerning Curses, refer to Chapter 1, Curses Screen Management Functions and Macros. For more information concerning command procedures or the G_floating representation of **double** variables, refer to *Guide to VAX C*.

1.1.1 Using the VAX C RTL as a Shareable Image

Instead of using the object code of the VAX C RTL functions, you can, as an option, use the VAX C RTL as a shareable image. When you use the VAX C RTL as a shareable image, you do not receive a copy of the object code in your program's local psect; control is passed, by means of pointers, from your program to libraries containing the RTL images where the designated function executes. After execution, control returns to your program. This process has a number of advantages. You significantly reduce the size of a program's executable image, the program's image takes up less disk space, and the program swaps in and out of memory faster because of decreased size.

To use the VAX C RTL as a shareable image, check with your system manager to make sure that the VAX C RTL software was installed so as to allow access to the shared images. Specifically, make sure that the system manager answered YES to step 5 listed in the *VAX C Installation Guide*. If that has been done, you can create an options file.

If you do *not* use the /G_FLOAT qualifier on the CC command, create an options file, OPTIONS_FILE.OPT, containing the following line:

```
SYS$SHARE:VAXCRTL.EXE/SHARE
```

If you *do* use the /G_FLOAT qualifier on the CC command, create an options file containing the following line:

```
SYS$SHARE:VAXCRTLG.EXE/SHARE
```

You must *not* include the libraries SYS$SHARE:VAXCRTL.EXE and SYS$SHARE:VAXCRTLG.EXE in the same options file.

After you have created the appropriate options file, named OPTIONS_FILE.OPT, you can compile and link your program with the following commands:

```
$ CC  PROGRAM.C RETURN
$ LINK  PROGRAM.OBJ, OPTIONS_FILE/OPT RETURN
```

Note that the include files are distributed with VAX C. The RTL libraries are distributed with VMS.

1.1.2 Macros

You may need to use macros as well as functions from the VAX C RTL. Macros are resolved at compilation time instead of link time. The compiler replaces the macro reference with text found in a definition file. Macros are not the only segments of source code found in the definition files; these files contain many definitions that are needed for some of the RTL functions to work properly. Macro definitions differ from the other definitions by their use of parameters which are delimited by parentheses.

Consequently, you need to learn about VAX C text substitution in order to use the VAX C RTL wisely.

To understand text substitution, you should know how the Standard I/O definitions are created. Definitions are comprised of **#define** preprocessor directives. Traditionally in the C language, these **#define** directives are located in files that have the .H file extension. If during installation of the VAX C software these files were extracted, you can locate them in the directory SYS$LIBRARY. For example, you can type the STDIO.H file (which contains Standard I/O definitions and macros) at your terminal with the following command:

```
$ TYPE  SYS$LIBRARY:STDIO.H RETURN
```

If you encounter an error, speak to your system manager about extraction of the .H definition files.

Since it is often more efficient to access these files in a VAX C provided library, this manual refers to the .H definition files as definition modules. For more information concerning text libraries and modules, refer to the *Guide to VAX C.*

The following identifiers are defined in the *stdio* definition module:

```
#define TRUE   1
#define FALSE  0
#define EOF   (-1)
```

You can use these definitions by including the proper definition module; use the **#include** preprocessor directive in your source file. At compile time, the compiler replaces the identifiers, within the source code, with the defined token string. In the previous code example, all instances of the identifier TRUE are replaced with the number 1.

To include the Standard I/O definitions in your file, use the following preprocessor directive:

```
#include stdio
```

Some VAX C RTL "functions" are implemented as macros using the **#define** preprocessor directive. For example, to use the macro **_toupper**, use the following line in your source code program:

```
#include ctype
```

In the definition module, *ctype*, you can find the following macro definition:

```
#define  _toupper(c)  ((c) >= 'a' && (c) <= 'z' ? (c) & 0x5F : (c))
```

In your program, you call the macro **_toupper** with the following source line:

```
    .
    .
    .
_toupper(a);
    .
    .
    .
```

The compiler searches through the source code for calls to **_toupper**, replacing each occurrence with the token string found in the macro definition. In the previous example, the compiler places the argument specified in the macro call (the letter a) wherever the identifier c appears in the defined token string. The token string in the previous example is VAX C source code that translates a lowercase letter to an uppercase letter. If the specified character is already an uppercase letter or if it is not a letter at all, the character is returned unaltered.

When calling VAX C RTL macros, use caution in specifying arguments that cause side effects, such as those that use the increment and decrement operators. For example, in the case of **_toupper**, even though you have access to the source code token string, you cannot determine the order in which the compiler evaluates each occurrence of (c) in the token string. The leftmost occurrence of (c) may not be evaluated first by the compiler. For a discussion of the passing of arguments to macros, refer to the *Guide to VAX C*.

Whereas the linker searches object libraries for the VAX C RTL function code, the compiler searches text libraries or directories for the VAX C RTL macros. When including text modules in your source code, the compiler first searches text libraries specified on the compilation command line for the definition module. If the compiler does not find the module, it translates the logical name C$LIBRARY; you can define C$LIBRARY to be a user-defined library. If the compiler cannot locate the module in the defined library or if there was no translation for C$LIBRARY, the compiler searches the text library SYS$LIBRARY:VAXCDEF.TLB; this library is shipped with the VAX C compiler and contains the .H definition files. If the compiler cannot find the specified module, it generates an error message.

Depending on the form of the **#include** line, there are other places to look for definition files that may contain VAX C RTL macros. For complete information about library searches, refer to the *Guide to VAX C*.

1.2 VAX C RTL Function and Macro Syntax

Once you know how to link object modules and include text modules, you must learn how to reference VAX C functions and macros in your program. Each of the remaining chapters in this manual provides detailed descriptions of VAX C RTL functions and macros.

In all chapters, the style of syntax used to describe each function and macro follows the usual convention for function syntax. A syntax is a compact representation of the order of a function's or macro's argument list (if any), the arguments' types, and the type of the value returned by function or macro. If the return value of the function cannot be easily represented by a VAX C data type keyword, look for a description of the return values in the explanatory text. The syntax descriptions provide insight into the functionality of the function or macro. These descriptions do not necessarily describe how to call the function or macro in your source code.

For example, consider the syntax of the **feof** function:

```
#include stdio

int  feof(file_pointer)

FILE *file_pointer;
```

The description of **feof** states that it is implemented as a macro. The syntax shows the following:

- The macro is defined in a definition module. You must include the *stdio* module to use the **feof** macro.
- The macro returns a value of data type **int**. Do not declare VAX C RTL macros. This line in the syntax indicates the arguments and the return value, not the form of a declaration.

- There is one argument, *file_pointer*, that is a pointer to FILE; FILE is an external data definition in the *stdio* module.

To use **feof** in a program, you need only call the macro and precede the call at some point by the **#include** directive, as in the following example:

```
#include stdio                    /* Include Standard I/O     */

main()
{
   FILE *infile;                  /* Define a file pointer    */
      .
      .
      .                           /* Call the function feof   */
   while ( ! feof(infile) )       /* Until EOF reached        */
      {                           /* Perform file operations  */
         .
         .
         .
      }
}
```

Because some library functions take varying numbers of arguments, syntax descriptions have additional conventions not used in other VAX C function definitions:

- Optional parameters are enclosed in square brackets ([ ]).
- An ellipsis (. . .) is used to show that a given parameter may be repeated.
- In cases where the type of a parameter may vary, its type is not shown in the syntax.

Consider the **printf** syntax description:

```
#include stdio

int printf(format_specification [,output_source, . . . ])

char *format_specification;
```

The syntax description for **printf** shows that the argument, output_source, is optional, may be repeated, and is not always of the same data type. The remaining information about the arguments of **printf** is in the description of the function following the syntax.

1.2.1 DEC/Shell File Specifications

The VAX C RTL functions and macros often manipulate files. One of the major portability problems is the different file specifications used on various systems. Since many C applications are ported to and from UNIX systems, it is convenient for all compilers to be able to read and understand UNIX system file specifications.

Consequently, functions from the DEC/Shell Run-Time Library are included in the VAX C RTL as a convenience for those interested in porting C programs from UNIX systems to VMS. The DEC/Shell functions in the VAX C RTL perform file conversion, file translation, and command language interpreter (CLI) status reports. For example, the RTL function **SHELL$TO_VMS** converts DEC/Shell file specifications to VMS file specifications.

The advantage of including the DEC/Shell functions in the VAX C RTL is that you do not have to rewrite C programs containing UNIX system file specifications. VAX C can translate most valid UNIX system file specifications to VMS file specifications.

NOTE

- VAX C cannot translate UNIX file specifications with more than one period character (.).
- If the UNIX file specification contains a period, all slash characters (/) must precede that period.

Although you do not need to be concerned with calling the Shell functions, you must be aware of the differences between the UNIX system and VMS file specifications, as well as the method used by VAX C to access files. For example, VAX C will accept a valid VMS specification and most valid UNIX file specifications, but VAX C cannot accept a combination of both. Table 1–1 illustrates the differences between UNIX system and VMS file specification delimiters.

Table 1–1: UNIX and VMS File Specification Delimiters

Description	VMS	UNIX
Node delimiter	::	!/
Device delimiter	:	/
Directory path delimiter	[ ]	/
Subdirectory delimiter	[.]	/
File extension delimiter	.	.
File version delimiter	;	Not applicable

For example, the formats of two valid specifications and one invalid specification are as follows:

System	File Specification	Valid/Invalid
VMS	BEATLE::DBA0:[MCCARTNEY]SONGS.LIS	VALID
UNIX	beatle!/dba0/mccartney/songs.lis	VALID
—	BEATLE::DBA0:[MCCARTNEY.C]/songs.lis	INVALID

When VAX C translates file specifications, it looks for both VMS and UNIX system file specifications. Consequently, there may be differences between the way in which VAX C translates UNIX system file specifications and the way in which the UNIX systems translate the same UNIX file specification. For example, if the two methods of file specification are combined, as in the previous list, VAX C could possibly interpret [MCCARTNEY.C]/songs.lis as either [MCCARTNEY]songs.lis or [C]songs.lis. Therefore, when VAX C encounters a mixed file specification, an error occurs.

UNIX systems use the same delimiter for the device name, the directory names, and the file name. Due to the ambiguity of UNIX file specifications, VAX C may not translate a valid UNIX system file specification according to your expectations. For instance, the VMS equivalent of bin/today can be either [BIN]TODAY or [BIN.TODAY]. VAX C can make the correct interpretation only from the actual files present. If a file specification conforms to UNIX system file name syntax for a single file or directory, it will be converted to the equivalent VMS file name if one of the following is true.

1. If the specification corresponds to an existing VMS directory, it is converted to that directory name. For example, /dev/dir/sub is converted to DEV:[DIR.SUB] if DEV:[DIR.SUB] exists.
2. If the specification corresponds to an existing VMS file name, it is converted to that file name. For example, dev/dir/file is converted to DEV:[DIR]FILE if DEV:[DIR]FILE exists.
3. If the specification corresponds to a nonexistent VMS file name, but the given device and directory exist, it is converted to a file name. For example, dev/dir/file is converted to DEV:[DIR]FILE if DEV:[DIR] exists.

In the UNIX system environment, you reference files with a numeric file descriptor. Some file descriptors reference standard input and output devices; some descriptors reference actual files. If the file descriptor belongs to an unopened file, the VAX C RTL opens the file. VAX C equates file descriptors with the following VMS logical names:

File Descriptor	VMS Logical	Meaning
0	SYS$INPUT	Standard Input
1	SYS$OUTPUT	Standard Output
2	SYS$ERROR	Standard Error
3_9	SHELL$FILE_n	File/Pipe opened by the Shell

You can use the DEC/Shell as your command language interpreter instead of the default interpreter, the DIGITAL Command Language (DCL). For more information concerning the DEC/Shell, refer to the *Guide to VAX C*.

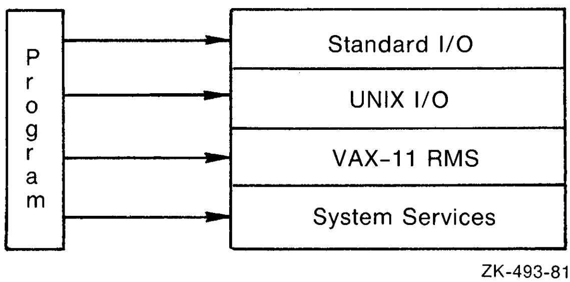
1.3 Input and Output on VMS

Once you have learned how to specify object libraries, how to specify text libraries, and how to call VAX C functions and macros, you are ready to use the RTL for its primary purpose: input and output.

Since every system has different methods of I/O, you should familiarize yourself with the VMS specific methods of file access. In this way, you will be equipped to predict possible differences in functionality when porting your source program from one operating system to another.

As shown in Figure 1–1, VAX C makes available four methods of I/O. The VMS system services "talk" directly to VMS, so they are "closest" to the operating system. The RMS functions use the system services, which in turn manipulate the operating system. The VAX C Standard and UNIX I/O functions and macros use the RMS functions, which in turn use the system services, which in turn manipulate the operating system. Since the VAX C Standard and UNIX I/O functions and macros must go through several layers of function calls before the system is manipulated, they are "furthest" from the operating system.

Figure 1–1: I/O Interface from C Programs



When the C programming language was developed on the UNIX operating system, the Standard I/O functions were meant to provide a convenient method of I/O that would be "powerful" enough so as to be efficient for most applications, and also to be portable so that the functions could be used on any system running C language compilers. VAX C adds functionality to this original specification. Since, as implemented in VAX C, the Standard I/O functions easily recognize line terminators, the VAX C Standard I/O functions are particularly useful for text manipulation. Also, VAX C implements some of the Standard I/O "functions" as preprocessor defined macros.

In a similar manner, the UNIX I/O functions originally were meant to provide a more direct access to the UNIX operating systems. These functions were meant to use a numeric file descriptor to represent a file; a UNIX system represents all peripheral devices as files, so as to provide a uniform method of access. Once again, VAX C adds functionality to the original specification. The UNIX I/O functions, as implemented in VAX C, are particularly useful for manipulating binary data. Also, VAX C implements some of the UNIX I/O "functions" as preprocessor defined macros.

The VAX C RTL includes the Standard I/O functions that were meant to exist on all C compilers, and also the UNIX I/O functions to maintain compatibility with as many other implementations of C as possible. However, both Standard I/O and UNIX I/O use VAX Record Management Services (RMS) to access files. So, in order to understand how the Standard and UNIX I/O functions manipulate RMS formatted files, you should understand the fundamentals of VAX Record Management Services. See Section 1.3.1 for more information concerning Standard and UNIX I/O in relationship to RMS files. For an introduction to RMS, refer to the *Guide to VAX/VMS File Applications*.

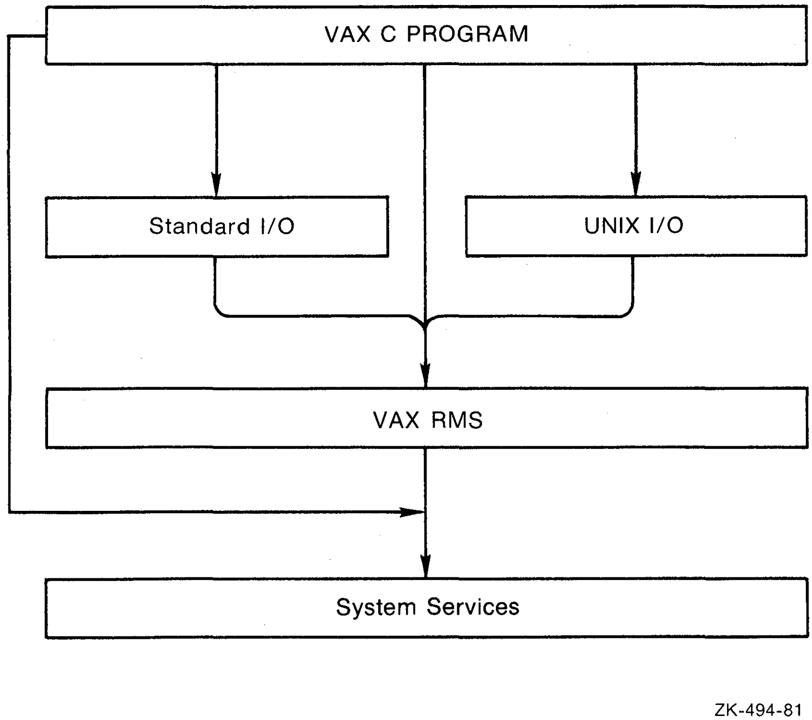
Before deciding which method is appropriate for you, you must first ask the question: Are you concerned with UNIX compatibility or with developing code that will run solely under VMS? If UNIX compatibility is important, you probably want to use the highest level of I/O—Standard I/O and UNIX I/O—because that level is largely independent of the operating system. Also, the highest level is easier to learn quickly, an important consideration for new programmers.

If UNIX compatibility is not important to you or if you require the sophisticated file processing that the Standard I/O and UNIX I/O methods do not provide, you will find VAX RMS desirable.

If you are writing system-level software, you may need to access VMS directly through calls to system services. For example, you may need to access a user-written device driver directly through Queue I/O Request System Service ($QIO). To do this, you need to use the VMS level of I/O; this level is recommended for experienced VMS programmers only. For examples of programs that call VMS system services, refer to the *Guide to VAX C*.

Many programmers may never use the RMS or the system services of VMS. The Standard and UNIX I/O functions are efficient enough for a large number of applications. Figure 1–2 illustrates the dependency of the Standard I/O and the UNIX I/O functions on RMS, and the various methods of I/O available to the VAX C programmer.

Figure 1–2: Mapping Standard and UNIX I/O to RMS



1.3.1 RMS Record and File Formats

To understand the capabilities, as well as the restrictions, of the Standard and UNIX I/O functions and macros, you need to understand VAX Record Management Services (RMS).

VAX RMS supports three types of file organization:

- Sequential
- Relative
- Indexed

Sequential files have consecutive records with no empty records in between; relative files have fixed-length cells that may or may not contain a record; and indexed files have records that contain data, carriage control information, and keys that permit various orders of access. The VAX C RTL functions can only access sequential files. If you wish to use the other file organizations, you must use the RMS functions. For more information concerning the RMS functions, refer to the *Guide to VAX C*.

VAX RMS is not concerned with the actual contents of records, so much as it is concerned about the record format, which is the way a record physically appears on the recording surface of the storage medium.

VAX RMS supports different record formats:

- Fixed length
- Variable length
- Variable with fixed-length control (VFC)
- Stream

You can specify a fixed-length record format at the time of file creation. This means that all records occupy the same space in the file. You cannot change the record format once you have created the file.

The length of records in variable length, VFC, and stream file formats can vary up to a maximum size that must be specified when you create the file. With variable-length record or VFC format files, the size of the record is held in a header section at the beginning of the data record. With stream files, RMS terminates the records when it encounters a specific character, such as a carriage-control or line-feed character. Stream files are very useful for storing text.

RMS allows you to specify carriage control attributes for records in a file. Such attributes include the implied carriage-return or the FORTRAN formatted records. RMS interprets these carriage controls when the file is output to a terminal, a line printer, or other device. The carriage control information is not stored in the data records.

Files created with VAX C programs have, by default, stream format with a line-feed record separator and implied carriage-return attributes. (In this manual, this type of file is referred to as a *stream file*.) Stream files can be manipulated very easily using the Standard and the UNIX I/O functions of the VAX C RTL. When using these files, there is no restriction on the ability to seek to any random byte of the file using the **fseek** or the **lseek** functions. However, if the file has one of the other RMS record formats, such as variable-length record format, then these functions, due to RMS restrictions, can seek only to record boundaries. Thus, unless you need to create or access files to be used with other VAX languages or utilities, it is recommended that you use the default VAX C stream format.

1.3.2 Stream Access to RMS Record Files

Stream access to record files is done with the record I/O facilities of RMS. The VAX C RTL emulates a byte stream by translating carriage control characters during the process of reading and writing records. Random access is allowed to record files, but positioning (with **fseek** and **lseek**) must be on a record boundary, and writes followed by reads (or reads followed by writes) do not work as with stream files. Positioning of a record file causes all buffered input to be discarded and buffered output to be written to the file.

Stream input from RMS record files is emulated by the VAX C RTL in two steps. First, the VAX C RTL reads a logical record from the file. Second, the VAX C RTL expands the record to simulate a stream of bytes by translating the record's carriage-control information (if any). In RMS terms, the VAX C RTL translates the information by one of the following methods:

- If the record attribute is implied carriage control (RAT=CR), then the VAX C RTL appends a newline to the record.
- If the record attributes are print carriage control (RAT=PRN), then the VAX C RTL expands and concatenates the prefix and postfix carriage controls before and after the record.
- If the record attributes are FORTRAN carriage control (RAT=FTN), then the VAX C RTL removes the initial control byte and appends the appropriate carriage control characters. The following rules describe the way the character in the first byte maps onto the prefix and postfix bytes that appear in the emulated stream. The identifier <record> denotes the bytes contained in the logical record exclusive of the first carriage-control byte; (\n) denotes the newline character; (\f) denotes the form-feed character; (\r) denotes the carriage-return character. Consider the following list.

NUL	→ <record>
0	→ \n\n <record> \r
1	→ \f <record> \r
+	→ <record> \r
$	→ \n <record>
all others	→ \n <record> \r

- If the record attributes are null (RAT=NONE) and the input is coming from a terminal, then the VAX C RTL appends the terminating character to the record. If the terminator is a carriage return or CTRL/Z, then the VAX C translates the character to a newline (\n).

  If the input is coming from a nonterminal file, then the VAX C RTL passes the record unchanged to the user program with no additional prefix or postfix characters.
- If the record format is variable length with fixed control (RFM=VFC), and the record attributes are not print carriage control (RAT is *not* PRN), then the VAX C RTL concatenates the fixed-control area to the beginning of the record.

As you read from the file, the VAX C Run-Time Library delivers a stream of bytes resulting from the translations. Information that is not read from an expanded record by one function call is delivered on the next input function call.

Stream output to RMS record files is performed by the VAX C Run-Time Library in two steps. First, the VAX C RTL forms a logical record from the bytes specified by the output function (**write**, for example) by translating any carriage-control bytes into RMS terms. Then, the VAX C RTL writes the logical record.

The first part of the stream output emulation is the formation of a logical record. As you write bytes to a record file, the emulator examines the information being written for record boundaries. The handling of information in the byte stream depends on the attributes of the destination file or device, as follows:

- If the record attributes specify no carriage-control information (RAT=null), then the VAX C RTL assumes that the stream of bytes presented in an output-function call is a logical record.

- If the destination file or device being written to has carriage-control information (RAT=CR, RAT=FTN, or RAT=PRN), then the emulator buffers output bytes while it searches for a newline character (\n). The emulator can buffer as many output bytes as the number of bytes contained in the maximum record size of the file. If the VAX C RTL encounters more than the number of bytes in the maximum record size of the file before it encounters a newline, then the VAX C RTL writes a record containing the data output thus far and clears the buffer. Otherwise, when a newline is found, the VAX C RTL forms the logical record by appending the newline to the buffered bytes.

The second part of stream output emulation is the actual writing of the logical record formed during the first step. The VAX C RTL executes one of the following steps to form the output record:

- If the output file record format is variable length with fixed control (RFM=VFC), and the record attributes do not include print carriage control (RAT is *not* PRN), then the VAX C RTL takes the beginning of the logical record to be the fixed-control header, and reduces the number of bytes written out by the length of the header. If there are too few bytes in the logical record, an error is signaled.
- If the record attribute is carriage control (RAT=CR), and if the logical record ends with a newline character (\n), the VAX C RTL drops the newline and writes the logical record with implied carriage control.
- If the record attribute is print carriage control (RAT=PRN), then the VAX C RTL writes the record with print carriage control. If the logical record ends with a newline character (\n), the VAX C RTL drops the newline, precedes the output record with a line feed character (\n), and follows the record with a carriage-return (\r). This is the reverse of the translation for stream input files with print carriage control attributes.
- If the record attributes are FORTRAN carriage control (RAT=FTN), then the VAX C RTL removes the first byte of the record, and concatenates prefix and postfix characters to the record. The following rules describe the way the character in the first byte maps onto the prefix and postfix bytes that appear in the emulated stream. The identifier <record> denotes the bytes contained in the logical record exclusive of the first carriage-control byte; (\n) denotes the newline character; (\f) denotes the form-feed character; (\r) denotes the carriage-return character. Consider the following list.

data	NULL <data>
data\r	+ <data>
\n data\r	<space> <data>
\f data\r	1 <data>
\n data	$ <data>

- If the record attribute is null (RAT=null), then the VAX C RTL performs a test to determine whether the logical record is to be written to a terminal device. If so, the VAX C RTL scans the record and replaces each newline character (\n) that is encountered by a carriage-return/line-feed pair (\r\n). Then, the VAX C RTL writes out the record with no carriage control.

1.4 Specific Portability Concerns

One of the last tasks in preparing to use the VAX C RTL, if you are going to port your source programs across systems, is to be aware of specific differences between the VAX C RTL and the run-time libraries of other implementations of the C language. This section describes some of the problems that programmers encounter when porting programs to and from VMS. Although portability is closely tied to the implementation of the run-time library, this section also contains information on the portability of other VAX C constructs.

It is not a goal of VAX C to duplicate all run-time functions that exist on every implementation of the language. VAX C implements a reasonable subset of existing C language functions and attempts to maintain complete portability in functionality whenever possible. Many of the Standard and UNIX I/O functions and macros contained in the VAX C Run-Time Library are functionally equivalent to those of other implementations.

However, in some instances functions provided by other implementations are not provided by VAX C because those functions conflict with the VMS operating system environment. In some cases, conflicting functions are replaced by an equivalent, more efficient VAX C function or macro. For example, the **unlink** function found on implementations running on UNIX operating systems has been replaced by the VAX C **delete** function.

In other cases, VAX C includes functions or macros that provide no functionality under VMS but are necessary so that programmers may port their programs to the VMS environment. For example, the **nonl** macro has no functionality in the VMS environment, but if you port a program from a UNIX system to VMS, the presence of **nonl** in the source code does not generate an error.

The RTL function and macro descriptions elaborate on issues presented in this section and describe concerns not documented here. Also, Appendixes A, B, and C provide information concerning the porting of C programs. Appendix A, VAX C RTL and RTLs of Other C Implementations, compares the functionality of VAX C RTL functions and macros with those of other implementations. Appendix B, VAX C Run-Time Modules and Entry Points, describes the run-time modules and entry points used by VAX C. Appendix C, VAX C Definition Modules, lists the .H definition files that are included in the compilation process to provide macro definitions and definitions used by some RTL functions; it may be helpful to review the definitions contained within these files.

The following list documents issues of concern for programmers who wish to port C programs to VMS:

- VAX C does not implement the global symbols **end**, **edata**, and **etext**.
- You should not attempt to substitute your own code for functions that are already supplied by VAX C. For example, the VAX C version of **strcpy** expects a return value. If you were to include a version of **strcpy** which did not return a value, the procedure would not perform correctly. The following code is an example of this:

  ```
  strcpy(p, q)
  char *p, *q;
  {
     while(*p++  =  *q++);
  }
  ```

  This use of **strcpy** will not work because code inside the VAX C Run-Time Library expects, and makes use of, a return value.
- There are differences in the way that VMS and UNIX systems lay out virtual memory. In UNIX, the address space between 0 and the break address are accessible to the user program. In VMS, the first page of memory is not accessible.

  If a program tries to reference location 0 on VMS, a hardware error (ACCVIO) is returned and the program terminates abnormally. VMS reserves the first page of address space to catch incorrect pointer references, such as a reference to a location pointed to by a null pointer. For this reason, some existing programs that run on UNIX systems may fail and should be rewritten.

- Some C programmers code all external declarations in **#include** files. Then, specific declarations that require initialization are redeclared in the relevant module. This practice causes the VAX C compiler to issue a warning message about multiply declared variables in the same compilation. One way to avoid this warning is to make the redeclared symbols **extern** variables in the **#include** files.
- The **asm** call is not supported by VAX C.
- Some C programs call the counted string functions **strcmpn** and **strcpyn**. These names are not used by VAX C. Instead, you can define macros that expand the **strcmpn** and **strcpyn** names into the equivalent names **strncmp** and **strncpy**.
- The VAX C compiler does not support the initialization form:

  ```
  int foo 123;
  ```

  Programs using this form of initialization will have to be changed.
- The fixed limit to the length of a string that VAX C accepts is 65,535 characters, or bytes. Long strings must be divided, and programs that use string arrays may need to be changed.
- VAX C defines the compile-time constants vax, vms, vax11c, vaxc, VAX, VMS, VAX11C, VAXC, and CC$g_float. These constants are useful for programs that must run compatibly on various machines and operating systems. For more information, refer to the *Guide to VAX C*.
- The C language does not guarantee any memory order for the variables in a declaration such as

  ```
  int a, b, c;
  ```

- The VMS Linker usually places VAX C **extern** variables in program sections (psects) of the same name as the variable. The linker then links the psects alphabetically by name. If you are porting a C program from another operating system to VMS, you may find that the order of items in the program has been allocated differently in virtual memory. This causes existing programs with hidden bugs to fail.
- The dollar sign ($) and the underscore (_) are legal characters in VAX C identifiers.

- The C language does not define any order for the evaluation of expressions in function parameter lists or in general expressions. The way in which different C compilers evaluate an expression is only important when the expression has "side effects," as in

  ```
  a[i] = i++;
  ```

  and

  ```
  f(p++, p++)
  ```

  Neither VAX C nor any other C compiler can guarantee that such expressions evaluate in the same order on all C compilers.

- The size of an integer is 32 bits on VAX C. Programs that were written for other machines and that assume a different size for a variable of type **int** will have to be modified.
- The C language defines structure alignment to be dependent on the machine for which the compiler is designed. By default, VAX C aligns structure members on byte boundaries. Other implementations may align structure members differently.
- References to structure members in VAX C must not be ambiguous. For more information, refer to the *Guide to VAX C*.
- Although registers are allocated based upon how frequently a variable is used, the keyword **register** gives the compiler a "strong hint" that the programmer wants to place a particular variable into a register. Whenever possible, the variable is placed into a register. Any scalar variable with the storage class **auto** or **register** may be allocated to a register as long as the variable's address is not taken with the ampersand operator (&) and as long as it is not a member of a structure or union.
- When moving programs from one operating system to another, the operations of the different linkers must also be taken into account. The VMS Linker does not load an object module from an object library unless the module contains a function definition, a **globaldef** definition, or a **globalvalue** definition that is needed to resolve a reference in another component of the program. When you refer to an **extern** variable from a program, the linker does not load the library module if the module contains only a compile-time initialization of the variable. This is a restriction, which can be avoided in one of two ways.

  In the following example, the program PROG.C contains an external declaration of a variable; the module LABDATA.C initializes the variable.

PROG.C:

```
main()
{
      .
      .
      .
   extern  float  lab_data[];
      .
      .
      .
}
```

LABDATA.C:

```
float  lab_data  =  { 1, 2, 3, 4, 5, 6, 7, 8 };

lab_data()
{
   .
   .
   .
}
```

You can link the object code for the program and the module either by naming the LABDATA object file in the LINK command, or by explicitly extracting the module from a library (here, it is part of the MYLIB library), as follows:

```
$ LINK  PROG,LABDATA,SYS$LIBRARY:VAXCRTL/LIB RETURN

$ LINK PROG,MYLIB/LIB/INCLUDE=LABDATA,- RETURN
_$ SYS$LIBRARY:VAXCRTL/LIB RETURN
```

You can also bundle the initialization in a module that will be loaded, that is, in a module that contains a function definition, a **globaldef** definition, or a **globalvalue** definition.

Chapter 2

Standard I/O Functions and Macros

In VAX C, and in most other implementations of C, stream files and their associated functions form the Standard I/O facilities. Stream files are files treated as streams of bytes. A series of bytes is read from or written to a stream file directly, with no record structure. (For more information concerning RMS file organization, refer to the *Guide to VAX C*. For more information concerning the VAX C RTL and RMS file organization, refer to Chapter 1, VAX C Run-Time Library Information.)

Stream files in VAX C correspond to RMS stream files with the line feed terminator attribute. In performing stream access to stream files, the VAX C RTL uses the block I/O facilities of RMS. A stream of bytes is either written to or read from a file with no translation. If the file has been opened for update, it can be read (**fread**) and written (**fwrite**) at the current byte position in the file. Note that file sharing is not supported for stream files.

The **fopen** Standard I/O function creates or opens existing stream files. You process stream files with conventional Standard I/O functions such as **fseek**, **ftell**, **fread**, **fwrite**, and **fprintf**. An **fread** followed by an **fwrite** places bytes in the file after the last byte of the previous **fread**. An **fwrite** followed by an **fread** causes reading to begin after the last byte of the previous **fwrite**.

A stream file can be positioned to an arbitrary byte at any time (**fseek**). If positioned beyond the end-of-file, the file is extended with NUL bytes. The file may be positioned relative to the beginning-of-file, relative to the current position, or relative to the end-of-file. The first byte in the file is byte zero; therefore, specifying zero as the absolute position in an **fseek** call positions the file at its first byte. You can also determine the current byte position of a stream file with the **ftell** function.

You must open a file for update if the file is going to be written randomly. For example:

```
#include stdio

main()
{
   FILE  *outfile;
   outfile  =  fopen("diskfile.dat", "w+");
      .
      .
      .
}
```

Here, the stream file DISKFILE.DAT is opened for "write update" access.

The Standard I/O functions access files by file pointer. A file pointer is defined in the include definition module *stdio* as follows:

```
typedef  struct  _iobuf  *FILE;
```

You can find the definition of the _iobuf identifier in the *stdio* module.

To declare a file pointer, use the following line:

```
FILE  *file_ptr;
```

NOTE

This definition of a file pointer differs from that of other implementations of the C language. So long as you access files using the functions and macros provided as part of the VAX C Run-Time Library, portability with respect to file pointers is possible.

2.1 Conversion Specifications

Several of the Standard I/O functions (including the Terminal I/O functions) use conversion characters to specify data formats for input and output. Consider the following example:

```
int     x = 5.0;
FILE    *outfile;
    .
    .
    .
fprintf(outfile, "The answer is %d.\n", x);
```

The decimal value of the variable x replaces the conversion specification %d in the string to be written to the file associated with the identifier outfile.

Each conversion specification begins with a percent sign (%). This sign is followed by an optional assignment-suppression character (*), an optional number giving the maximum field width, and a conversion character.

2.1.1 Conversion of Input Information

A conversion specification for the input of information can include three kinds of items:

1. White-space characters (spaces, tabs, and newlines), which match optional white-space characters in the input field.
2. Ordinary characters (not %), which must match the next nonwhite-space character in the input.
3. Conversion specifications, which govern the conversion of the characters in an input field and their assignment to an object indicated by a corresponding input pointer.

Each input pointer is an address expression indicating an object whose type matches that of a corresponding conversion specification. Conversion specifications form part of the format specification. The indicated object is the target that receives the input value. There must be as many input pointers as there are conversion specifications, and the addressed objects must match the types of the conversion specifications.

Table 2–1 describes the conversion characters for formatted input.

Table 2–1: Conversion Characters for Formatted Input

Character	Meaning
d	Expect a decimal integer in the input. The corresponding argument must point to an **int**.
o	Expect an octal integer in the input (with or without a leading zero). The corresponding argument must point to an **int**.
x	Expect a hexadecimal integer in the input (without a leading 0x). The corresponding argument must point to an **int**.

Table 2–1 (Cont.): Conversion Characters for Formatted Input

Character	Meaning
c	Expect a character in the input. The corresponding argument must point to a **char**. The usual skipping of white-space characters can be disabled in this case, so that *n* white-space characters can be read with *%nc*. If a field width is given with c, the given number of characters is read and the corresponding argument should point to an array of **char**.
s	Expect a string in the input. The corresponding argument must point to an array of characters that is large enough to contain the string plus the terminating NUL character (\0). The input field is terminated by a space, tab, or newline.
e, f	Expect a floating-point number in the input. The corresponding argument must point to a **float**. The input format for floating-point numbers is [+|−]nnn[.ddd]][{E|e}[+|−]nn], where the n's and the d's are decimal digits (as many as indicated by the field width minus the signs and the letter E).
i	Expect an integer whose type is determined by the leading input characters. For example, a leading zero is equated to octal. The form 0X is equated to hexadecimal and all other forms are equated to decimal. Each corresponding argument must be an integer pointer.
ld, lo, lx	Same as d, o, and x, except that a **long** integer of the specified radix is expected. (Retained for portability only, since **long** and **int** are the same in VAX C.)
le, lf	Same as e, and f, except that the corresponding argument is a **double** instead of a **float**. The same effect can be achieved by using an uppercase E or F.
hd, ho, hx	Same as d, o, and x, except that a **short** integer of the specified radix is expected.
[. . .]	Expect a string that is not delimited by white-space characters. The brackets enclose a set of characters (not a string). Ordinarily, this set (or "character class") is made up of the characters that comprise the string field. Any character not in the set will terminate the field. However, if the first (leftmost) character is an up-arrow, then the set shows the characters that terminate the field. The corresponding argument must point to an array of characters.

Remarks

- The delimiters of the input field can be changed with the bracket ([ ]) conversion specification. Otherwise, an input field is defined as a string of nonwhite-space characters. It extends either to the next white-space character or until the field width, if specified, is exhausted. The function reads across line/record boundaries, since the newline character is a white-space character.
- A call to one of the input conversion functions resumes searching immediately after the last character processed by a previous call.
- If the assignment-suppression character (*) appears in the format specification, no assignment is made. The corresponding input field is interpreted and then skipped.
- The arguments must be pointers or other address-valued expressions, since VAX C permits only calls by value. To read a number in decimal format and assign its value to n, you must use

  ```
  scanf("%d", &n)
  ```

  not

  ```
  scanf("%d", n)
  ```

- White space in a format specification matches optional white space in the input field. The format specification

  ```
  field = %x
  ```

  matches

  ```
  field = 5218
  field=5218
  field= 5218
  field =5218
  ```

  but not

  ```
  fiel d=5218
  ```

2.1.2 Conversion of Output Information

The format specification string for the output of information may contain two kinds of items:

- Ordinary characters, which are simply copied to the output.
- Conversion specifications, each of which causes the conversion of a corresponding output source to a character string, in a particular format.

Table 2–2 describes the conversion characters for formatted output.

Table 2–2: Conversion Characters for Formatted Output

Character	Meaning
d	Convert to decimal format.
o	Convert to octal format.
x	Convert to unsigned hexadecimal format (without leading 0x). An uppercase X causes the hexadecimal digits A-F to be printed in uppercase. A lowercase x causes those digits to be printed in lowercase.
u	Convert to unsigned decimal format (giving a number in the range zero to 4,294,967,295).
c	Output single character (NUL characters are ignored).
s	Write characters until NUL is encountered or until number of characters indicated by the precision specification is exhausted. If the precision specification is zero or omitted, all characters up to a NUL are output.
e	Convert **float** or **double** to the format [–]m.nnnnnnE[+|–]xx, where the number of n's is specified by the precision (default = 6). If the precision is explicitly zero, the decimal point appears but no n's appear. An E is printed if the conversion character is an uppercase E. An e is printed if the conversion character is a lowercase e.
f	Convert **float** or **double** to the format [-]m..m.nnnnnn, where the number of n's is specified by the precision (default - 6). Note that the precision does not determine the number of significant digits printed. If the precision is explicitly zero, no decimal point appears and no n's appear.

Table 2–2 (Cont.): Conversion Characters for Formatted Output

Character	Meaning
g	Convert **float** or **double** to d, e, or f format, whichever is shorter (suppress insignificant zeros).
%	Write out the percent symbol. No conversion is performed.

The following characters can be used between the percent sign (%) and the conversion character. They are optional, but if specified, they must occur in the order listed.

Character	Meaning
- (hyphen)	Left justify the converted output source in its field.
width	Use this integer constant as the minimum field width. If the converted output source is wider than this minimum, write it out anyway. If the converted output source is narrower than the minimum width, pad it to make up the field width. Padding is with spaces normally, and with zeros if the field width is specified with a leading zero; this does not mean that the width is an octal number. Padding is on the left normally and on the right if a minus sign is used.
. (period)	Separates field width from precision.
precision	Use this integer constant to designate the maximum number of characters to print with s format, or the number of fractional digits with e or f format.
l (lowercase letter "L")	Indicates that a following d, o, x, or u specification corresponds to a **long** output source. In VAX C, all **int** values are long by default.
* (asterisk)	Can be used to replace the field width specification and/or the precision specification. The corresponding width or precision is given in the output source.

2.2 Opening and Closing Files

The following sections describe the Standard I/O functions that open and close files.

2.2.1 fclose

The **fclose** function closes a file by flushing any buffers associated with the file control block and freeing the file control block and buffers previously associated with the file pointer.

The syntax of the function is as follows:

```
#include stdio

int fclose (FILE *file_ptr);
```

Arguments

The *file_ptr* argument is a pointer to the file to be closed.

Additional Information

When a program terminates normally, **fclose** is called automatically for all open files. On success, **fclose** returns zero. If the buffered data cannot be written to the file, or if the file control block is not associated with an open file, **fclose** returns EOF (a preprocessor constant defined in the **#include** module *stdio*).

2.2.2 fdopen

The function **fdopen** associates a file pointer with a file descriptor returned by an **open**, **creat**, **dup**, **dup2**, or **pipe** function.

The syntax of the function is as follows:

```
#include stdio

FILE *fdopen (int file_desc, char *a_mode);
```

Arguments

The arguments for the **fdopen** function are as follows.

file_desc The file descriptor returned by **open**, **creat**, **dup**, **dup2**, or **pipe**.

a_mode One of the character strings "r", "w", "a", "r+", "w+", "rb", "r+b", "rb+", "wb", "w+b", "wb+", "ab", "a+b", "ab+", or "a+", for read, write, append, read update, write update, or append update, respectively.

The access modes have the following effects:

- "r" opens an existing file for reading.
- "w" creates a new file, if necessary, and opens the file for writing. If the file already exists, it creates a new file with the same name and a higher version number.
- "a" opens the file for append access. An existing file is positioned at end-of-file, and data is written there. If the file does not exist, the VAX C RTL creates it.

The update access modes allow a file to be opened for both reading and writing. When used with existing files, "r+" and "a+" differ only in the initial positioning within the file. The modes are as follows:

- "r+" opens an existing file for read update access. It is opened for reading, positioned initially at beginning-of-file, but writing is also allowed.
- "w+" opens a new file for write update access.
- "a+" opens a file for append update access. The file is positioned at end-of-file (writing) initially. If the file does not exist, the VAX C RTL creates it.
- "b" means binary access mode. In this case, no conversion of carriage control information is attempted.

Additional Information

The **fdopen** function allows you to access a file, originally opened by one of the UNIX I/O functions, with Standard I/O functions. Ordinarily, a file can be accessed by either a file descriptor or by a file pointer, but not both, depending on the way you open it. For more information, refer to Chapter 1, VAX C Run-Time Library Information.

On success, **fdopen** returns a nonzero value which is the file pointer. On error, **fdopen** returns zero.

See also **freopen** and **fopen** in Sections 2.2.4 and 2.2.3.

2.2.3 fopen

The function **fopen** opens a file by returning the address of a FILE structure.

The syntax of the function is as follows:

```
#include stdio

FILE *fopen (const char *file_spec, const char *a_mode, ...);
```

Arguments

The arguments for the **fopen** function are as follows:

file_spec	A character string containing a valid file specification.
a_mode	An access mode indicator. See Section 2.2.2 for a description of *a_mode*.
...	Represents optional file attribute arguments. The file attribute arguments are the same as those used in the **creat** function (see Chapter 4, UNIX System I/O Functions).

Additional Information

On error, this function returns the null pointer value; the constant NULL is defined in the definition module *stdio* to be the null pointer value. The function returns NULL to signal the following errors:

- File protection violations
- Attempts to open a nonexistent file for read access
- Failure to open the specified file

The file control block may be freed with the **fclose** function, or by default on normal program termination.

See also **fdopen** and **freopen** in Sections 2.2.2 and 2.2.4.

2.2.4 freopen

The **freopen** function substitutes the file, named by a file specification, for the open file addressed by a file pointer. The latter file is closed.

The syntax of the function is as follows:

```
#include stdio

FILE *freopen (const char *file_spec, const char *a_mode, FILE *file_ptr, ...);
```

Arguments

The arguments for the **freopen** function are as follows:

file_spec A pointer to a string that contains a valid VMS or DEC/Shell file specification. After the function call, the given file pointer is associated with this file.

a_mode An access mode indicator. See Section 2.2.2 for a description of *a_mode*.

file_ptr A file pointer.

... Represents optional file attribute arguments. The file attribute arguments are the same as those used in the **creat** function (see Chapter 4, UNIX System I/O Functions).

Additional Information

On error, this function returns the null pointer value; the constant NULL is defined in the definition module *stdio* to be the null pointer value.

You typically use **freopen** to associate one of the predefined names stdin, stdout, or stderr with a file. For more information concerning these predefined names, refer to Chapter 3, Terminal I/O Functions and Macros.

See also **fdopen** and **fopen** in Sections 2.2.2 and 2.2.3.

2.3 Reading from Files

The following sections describe the Standard I/O functions and macros that read data from files.

2.3.1 getc, fgetc, getw

The **fgetc** and **getw** functions and the **getc** macro return characters from a specified file.

The syntax descriptions are as follows:

```
#include stdio

int fgetc (FILE *file_ptr);
int getc (FILE *file_ptr);
int getw (FILE *file_ptr);
```

Arguments

The argument *file_ptr* is a pointer to the file to be accessed.

Additional Information

The compiler substitutes the following text for a call to the macro **getc**(file_ptr):

```
fgetc(file_ptr)
```

The **getc** macro returns the next character as an **int** from the specified file. The file is left positioned after the returned character, and the next **getc** call takes the character from that position. The **fgetc** function and the **getc** macro are functionally equivalent.

The **getw** function returns the next four characters from the specified input file as an **int**. No conversion is performed. If end-of-file is encountered during the retrieval of any of the four characters, then EOF (a preprocessor constant defined in the **#include** module *stdio*) is returned and all four characters are lost.

The two functions and the macro return EOF on end-of-file or error, but since EOF is a perfectly good integer, **feof** and **ferror** should be used to check their success.

2.3.2 fgets

The **fgets** function reads a line from a specified file, up to a specified maximum number of characters or up to and including the newline character, whichever comes first; the function stores the string in the *str* argument.

The syntax of the function is as follows:

```
#include stdio

char *fgets (char *str, int maxchar, FILE *file_ptr);
```

Arguments

The arguments for the **fgets** function are as follows:

str	The address where the fetched string will be stored.
maxchar	Specifies the maximum number of characters to fetch.
file_ptr	A file pointer.

Additional Information

The function terminates the line with a NUL (\0) character. Unlike **gets**, **fgets** places the newline that terminates the input line into the user buffer if it fits. On end-of-file or error, the function returns NULL (which is defined in the *stdio* definition module to be the null pointer value). Otherwise, it returns the address of the first character in the line.

2.3.3 fread

The **fread** function reads a specified number of items from the file.

The syntax of the function is as follows:

```
#include stdio

size_t fread (void *ptr, size_t size_of_item,
                         size_t number_items, FILE *file_ptr);
```

Arguments

The arguments for the **fread** function are as follows:

ptr	A pointer to the location, within memory, in which to place the information being read. You determine the type of the object pointed to by the type of the items being read.
size_of_item	The size of the items being read, in bytes.
number_items	The number of items to be read.
file_ptr	A pointer that indicates the file from which the items are to be read.

Additional Information

The type size_t is defined in the standard include module *stdio*. The reading begins at the current location in the file. The items read are placed in storage beginning at the location given by the first argument. The size of an item in bytes must also be specified.

If the file pointed to by *file_ptr* is a record file, **fread** will only read the number of items specified in *number_items*.

The function returns the number of items actually read. If **fread** encounters the end-of-file or an error, it returns zero (not EOF).

2.3.4 fscanf, sscanf

The **fscanf** function performs formatted input from a specified file, and the **sscanf** function performs formatted input from a character string in memory.

The syntax descriptions of the functions are as follows:

```
#include stdio

int fscanf (FILE *file_ptr, const char *format_spec, ...);
int sscanf (char *str, const char *format_spec, ...);
```

Arguments

The arguments for the **fscanf** and **sscanf** functions are as follows:

file_ptr	A pointer to the file that provides input text for **fscanf**.
format_spec	Contains characters to be taken literally from the input or converted and placed in memory at the specified . . . argument.
. . .	Optional expressions whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, the input pointers can be omitted. Otherwise, the function calls must have exactly as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input_ptrs. Conversion specifications are matched to input sources in simple left-to-right order.
str	The address of the character string that provides the input text to **sscanf**.

An example of a conversion specification is as follows:

```
main ()
{
   int   temp, temp2;
   FILE  *file_ptr;

   fscanf(file_ptr, "%d %d", &temp, &temp2);
   printf("The answers are %d, and %d.", temp, temp2);
}
```

Given a file, designated by the argument file_ptr, with the following contents

```
4 17
```

sample input from the previous example is as follows:

```
$ RUN  EXAMPLE RETURN
The answers are 4, and 17.
```

For a complete description of the format specification and the input pointers, refer to Section 2.1.1.

Additional Information

The functions return the number of successfully matched and assigned input items. If end-of-file (or the end of the string) is encountered, the functions return EOF (a preprocessor constant defined in the *stdio* definition module).

2.3.5 ungetc

The **ungetc** function pushes back a character into the input stream and leaves the stream positioned before the character.

```
#include stdio

int ungetc (char character, FILE *file_ptr):
```

Arguments

The arguments for the **ungetc** function are as follows:

character A value of type **char**.

file_ptr A file pointer.

Additional Information

When using the **ungetc** function, the character is said to be "pushed back" onto the file, since it will be returned by the next **getc** call. The function returns the push-back character or EOF if it cannot push the character back.

One push-back is guaranteed, even if there has been no previous activity on the file. The function **fseek** erases all memory of pushed-back characters. Note that the pushed back character is not written to the underlying file.

2.4 Writing to Files

The following sections describe the Standard I/O functions and macros used to write to files.

2.4.1 fprintf, sprintf

The **fprintf** function performs formatted output to a specified file, and the **sprintf** function performs formatted output to a string in memory.

The syntax descriptions of the functions are as follows:

```
#include stdio

int fprintf (FILE *file_ptr, const char *format_spec, ...);
int sprintf (char *str, const char *format_spec, ...);
```

Arguments

The arguments for the **fprintf** and **sprintf** functions are as follows:

file_ptr	A pointer to the file to which output is to be written.
format_spec	Contains characters to be written literally to the output or converted as specified in the argument output_src.
. . .	Optional expressions whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, the output sources may be omitted. Otherwise, the function calls must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources. Conversion specifications are matched to output sources in simple left-to-right order.
str	The address of the string that will receive the formatted output.

An example of a conversion specification is as follows:

```
main()
{
   int  temp = 4, temp2 = 17;
   FILE *file_ptr;

   fprintf(file_ptr, "The answers are %d, and %d.", temp, temp2);
}
```

Sample output (to the file designated by file_ptr) from the previous example is as follows:

```
The answers are 4, and 17.
```

For a complete description of the format specification and the output source, refer to Section 2.1.1.

Additional Information

These functions return the number of successfully matched and assigned output items.

2.4.2 fputs

The **fputs** function writes a character string to a file without copying the string's NUL terminator (\0).

The syntax of the function is as follows:

```
#include stdio

int fputs (const char *str, FILE *file_ptr);
```

Arguments

The arguments for the **fputs** function are as follows:

str	A pointer to a character string.
file_ptr	A file pointer.

2.4.3 fwrite

The **fwrite** function writes a specified number of items to the file.

The syntax of the function is as follows:

```
#include stdio

size_t fwrite (void *ptr, size_t size_of_item,
                          size_t number_items, FILE *file_ptr);
```

Arguments

The arguments for the **fwrite** function are as follows:

ptr	A pointer to the memory location from which information is being written.
size_of_item	The size of the items being written, in bytes.
number_items	The number of items being written.
file_ptr	A file pointer and indicates the file to which the items are being written.

Additional Information

The type size_t is defined in the standard include module *stdio*.

The function returns the number of items actually written. The number of records actually written depends upon the maximum record size of the file.

If the file is a record-mode file, **fwrite** outputs at least number_items records, each of length size_of_item.

2.4.4 putc, fputc, putw

The **putc** macro and the **fputc** and **putw** functions write characters to a specified file.

The syntax descriptions are as follows:

```
#include stdio

int putc  (char character, FILE *file_ptr);
int fputc (char character, FILE *file_ptr);
int putw (int integer, FILE *file_ptr);
```

Arguments

The arguments for the **putc** macro and the **fputc** and **putw** functions are as follows:

character	An object of type **char**.
integer	An object of type **int** or **long**.
file_ptr	A file pointer.

Additional Information

The compiler substitutes the following text for a call to the macro **putc**(character, file_ptr):

```
fputc(character, file_ptr)
```

The **putc** macro writes a single character to a file and returns the character. The file pointer is left positioned after the character. The **fputc** function is functionally equivalent to **putc**. The **putw** function writes four characters to the output file as an **int**. No conversion is performed.

The two functions and the macro return EOF (defined in the *stdio* definition module) to designate output errors. Since EOF is itself an integer, **ferror** should be used to detect errors encountered by **putw**.

2.5 Maneuvering in Files

The following sections describe the Standard I/O functions used to position the file pointer.

2.5.1 fflush

The **fflush** function writes out any buffered information for the specified file.

The syntax of the function is as follows:

```
#include stdio

int fflush (FILE *file_ptr);
```

Arguments

The argument *file_ptr* is a file pointer.

Additional Information

The **fflush** function returns zero when it is successful. If the buffered data cannot be written to the file, or if the file control block is not associated with an output file, **fflush** returns EOF (a preprocessor constant defined in the *stdio* definition module).

Note that output files are normally buffered if, and only if, they are not directed to a terminal, but stderr is not buffered by default.

2.5.2 fseek

The **fseek** function positions the file to the specified byte offset in the file.

The syntax of the function is as follows:

```
#include stdio

int fseek (FILE *file_ptr, int offset, int direction);
```

Arguments

The arguments for the **fseek** function are as follows:

file_ptr	A file pointer.
offset	The offset specified in bytes.
direction	An integer indicating whether the offset is measured forward from the current read or write address (1), forward from the beginning of the file (0), or backwards from the end-of-file (2).

Additional Information

The **fseek** function returns EOF (a preprocessor constant defined in the *stdio* definition module) for improper seeks; zero for successful seeks.

In general, **fseek** should always be directed to an absolute position returned by **ftell**. With stream files, the direction argument can be 0, 1, or 2. With record files, an **fseek** to a position that was not returned by **ftell** causes unpredictable behavior.

See also **ftell**.

2.5.3 ftell

The **ftell** function returns the current byte offset to the specified stream file.

```
#include stdio

int ftell (FILE *file_ptr);
```

Arguments

The argument *file_ptr* is a file pointer.

Additional Information

The **ftell** function measures the offset from the beginning of the file. With record files, **ftell** returns the starting position of the current record, not the current byte offset.

This function is useful only for handing an offset to **fseek**, to reposition the file to where it was when **ftell** was called. The function **ftell** returns EOF upon error.

2.5.4 rewind

The **rewind** function sets the file to its beginning.

The syntax of the function is as follows:

```
#include stdio

int rewind (FILE *file_ptr);
```

Arguments

The argument *file_ptr* is a file pointer.

Additional Information

The **rewind** function is equivalent to **fseek** (file-pointer, 0, 0). The function returns EOF to indicate failure; zero to indicate success. The **rewind** function can be used with either record or stream files.

2.6 Additional Standard I/O Functions and Macros

The following sections describe the Standard I/O functions that perform various tasks.

2.6.1 access

The **access** function checks a file to see whether a specified access mode is allowed.

The syntax of the function is as follows:

```
#include stdio

int access (char *file_spec, int mode);
```

Arguments

The arguments for the **access** function are as follows:

file_spec A character string that gives a VMS or DEC/Shell file specification. The usual defaults and logical name translations are applied to the file specification.

mode Interpreted as follows:

Mode Argument	Access Mode
0	Tests to see if the file exists.
1	Execute.
2	Write (implies delete access).
4	Read.

Combinations of access modes are indicated by summing the values. For example, the integer 7 indicates RWED.

NOTE

The function **access** does not accept network files as arguments.

Additional Information

The **access** function returns zero if the access is allowed and EOF if not allowed.

2.6.2 clearerr

The **clearerr** macro resets the error and end-of-file indications for a file (so that **ferror** and **feof** will no longer return a nonzero value).

The syntax of the macro is as follows:

```
#include stdio

void clearerr (FILE *file_ptr);
```

Arguments

The argument *file_ptr* is a file pointer.

Additional Information

Note that VAX C implements **clearerr** as a macro.

2.6.3 feof

The **feof** macro tests a file to see if the end-of-file has been reached.

The syntax of the macro is as follows:

```
#include stdio
int feof (FILE *file_ptr);
```

Arguments

The argument *file_ptr* is a file pointer.

Additional Information

If end-of-file has been reached, **feof** returns a nonzero integer; if not, it returns 0. Note that VAX C implements **feof** as a macro.

2.6.4 ferror

The **ferror** macro returns a nonzero integer if an error has occurred while reading or writing a file.

The syntax of the macro is as follows:

```
#include stdio
int ferror (FILE *file_ptr);
```

Arguments

The argument *file_ptr* is a file pointer.

Additional Information

A call to the macro continues to return this indication until the file is closed or until **clearerr** is called. Note that VAX C implements **ferror** as a macro.

2.6.5 fgetname

The **fgetname** function returns the file specification associated with a file pointer.

The syntax of the function is as follows:

```
#include stdio

char *fgetname (FILE *file_ptr, char *buffer, ...);
```

Arguments

The arguments for the **fgetname** function are as follows:

file_ptr A file pointer.

buffer A pointer to a character string that is large enough to hold the file specification.

. . . Represents an optional additional argument that can be either 1 or 0. If you specify 1, the function **fgetname** returns the file specification in VMS format. If you specify 0, the function **fgetname** returns the file specification in DEC/Shell format. If you do not specify this argument, this function returns the file name according to your current command language interpreter. For more information concerning DEC/Shell file specifications, refer to Chapter 1, VAX C Run-Time Library Information.

Additional Information

The **fgetname** function places the file specification at the address given in *buffer* and returns the address of *buffer*. The buffer should be an array large enough to contain a fully qualified file specification (the maximum length is 256 characters). When an error occurs, **fgetname** returns 0.

2.6.6 mktemp

The **mktemp** function creates a unique file name from a template.

The syntax of the function is as follows:

```
#include stdio

char *mktemp (char *template);
```

Arguments

The *template* argument is a pointer to a user-defined template. You supply the template in the form, "namXXXXXX". The six trailing X's are replaced by a unique series of characters. You may supply the first three characters.

Additional Information

The **mktemp** function returns a pointer to the file name it creates. If a unique file name cannot be created, **mktemp** returns a pointer to an empty string (\0).

2.6.7 remove, delete

The **remove** and **delete** functions cause a file to be deleted.

The syntax of the **remove** and **delete** functions is as follows:

```
#include stdio

int remove (const char *file_spec);

int delete (const char *file_spec);
```

Arguments

The argument *file_spec* is a pointer to the string that is a VMS file specification or a DEC/Shell file specification.

Additional Information

The **remove** and **delete** functions return a nonzero value if the operation fails.

Note that the **remove** and **delete** functions are functionally equivalent in the VAX C RTL.

2.6.8 rename

The **rename** function gives a new name to an existing file.

The syntax of the **rename** function is as follows:

```
#include stdio

int rename (const char *old_file_spec, const char *new_file_spec);
```

Arguments

The arguments to the **rename** function are as follows:

old_file_spec	A pointer to a string that is the existing name of the file to be renamed.
new_file_spec	A pointer to a string that is the new name to be given to the file.

Additional Information

The **rename** function returns a nonzero value if the operation fails.

If you attempt to rename a file that is currently open, the behavior is undefined. Note that you cannot rename a file from one physical device to another. Both the old and new file specifications must reside on the same device.

2.6.9 setvbuf, setbuf

The functions **setvbuf** and **setbuf** associate a buffer with an input or output file.

The syntax of the functions is as follows:

```
#include stdio

int setvbuf (FILE *file_ptr, char *buffer, int type, size_t size);

int setbuf(FILE *file_ptr, char *buffer);
```

Arguments

The arguments to the **setvbuf** and **setbuf** functions are as follows:

file_ptr A pointer to a file.

buffer A pointer to an array. If either _IOFBF or _IOLBF is specified as a value for *type*, input/output operations will be done using the array pointed to by *buffer*. The buffer must be large enough to hold an entire input record.

If *buffer* is a NULL pointer, input/output operations will be done using the buffer automatically allocated by the VAX C Run-Time Library. If _IONBF is specified by *type*, input/output operations will be completely unbuffered and the pointer in *buffer* is ignored.

type A value that determines how the file will be buffered.

The following values for *type* are defined in stdio:

- _IOFBF causes input/output to be fully buffered if possible.
- _IOLBF causes output to be line buffered if possible (the buffer will be flushed when a new-line character is written, when the buffer is full, or when input is requested).
- _IONBF causes input/output to be completely unbuffered if possible. _IONBF causes *buffer* and *size* to be ignored.

size The number of bytes in the array pointed to by *buffer*. The constant BUFSIZ in *stdio* is recommended as a good buffer size.

Additional Information

The **setvbuf** and **setbuf** functions can be used after a file is opened but must be used before any input or output operations.

The functions return a nonzero value if an invalid value is given for *type* or *size*; otherwise, they return a zero value.

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the file in the same block.

A buffer is normally obtained by calling **malloc**. For more information, refer to Chapter 9, Memory Allocation Functions.

2.6.10 tmpfile

The **tmpfile** function creates a temporary file that is opened for update.

The syntax of the function is as follows:

```
#include stdio

FILE *tmpfile (void);
```

Additional Information

The file exists only for the duration of the process and is preserved across forks. The function returns the address of a FILE pointer (defined in the *stdio* definition module), or a null pointer value (NULL) if there is an error.

2.6.11 tmpnam

The **tmpnam** function creates a character string that can be used in place of the file-name argument in other function calls.

The syntax of the function is as follows:

```
#include stdio

char *tmpnam (char *name);
```

Arguments

The *name* argument is a character string containing a name to be used in place of file-name arguments in other functions or macros. If the *name* argument is the null pointer value NULL, **tmpnam** returns the address of an internal storage area. If *name* is not NULL, then it is taken to be the address of an area of length L_tmpnam (defined in the *stdio* definition module). In this case, **tmpnam** returns the *name* argument. Successive calls to **tmpnam** with a NULL argument cause the function to overwrite the current name.

2.7 Program Examples

Example 2–1 illustrates the use of the **fopen**, **ftell**, **sprintf**, **fputs**, **fseek**, **fgets**, and **fclose** functions.

Example 2–1: Using the Standard I/O Functions

```
/*  This program establishes a file pointer, writes lines from *
 *  a buffer to the file, moves the file pointer to the second *
 *  record, copies the record to the buffer, and then prints   *
 *  the buffer to the screen.                                  */

#include stdio

main ()
{
   char    buffer[32];
   int     i, pos;
   FILE    *fptr;
                             /*  Set file pointer              */
   fptr = fopen("data.dat", "w+");
   if (fptr <= NULL)
      {
         perror("fopen");
         exit ();            /*  Exit if fopen error           */
      }

   for (i=1; i<5; i++)
      {
         if (i == 2)         /*  Get position of record 2      */
            pos = ftell(fptr);
                             /*  Print a line to the buffer    */
         sprintf(buffer, "test data line %d\n", i);
                             /*  Print buffer to the record    */
         fputs(buffer, fptr);
      }
                             /*  Go to record number 2         */
   if (fseek(fptr, pos, 0) < 0)
      {
         perror("fseek");    /*  Exit on fseek error           */
         exit ();
      }
                             /*  Put record 2 in the buffer    */
   if   (fgets(buffer, 32, fptr) == NULL)
      {
         perror("fgets");    /*  Exit on fgets error           */
         exit();
      }
                             /*  Print the buffer              */
   printf("Data in record 2 is: %s", buffer);
   fclose(fptr);            /*  Close the file                */
}
```

Sample output, to the terminal, from the previous example is as follows.

```
$ RUN  EXAMPLE RETURN
Data in record 2 is: test data line 2
```

Sample output, to DATA.DAT, from the previous example is as follows:

```
test data line 1
test data line 2
test data line 3
test data line 4
```

Chapter 3

Terminal I/O Functions

VAX C defines three file pointers that allow you to perform I/O to and from the logical devices usually associated with the user's terminal (for interactive jobs) or a batch stream (for batch jobs). Since, in VMS, the three process permanent files SYS$INPUT, SYS$OUTPUT, and SYS$ERROR perform the same functions for both interactive and batch jobs, the term "Terminal I/O" refers to both terminal and batch stream I/O. The file pointers stdin, stdout, and stderr are defined when you include the *stdio* definition module using the **#include** preprocessor directive.

The file pointer stdin is associated with the terminal to perform input. This file is equivalent to SYS$INPUT. The file pointer, stdout, is associated with the terminal to perform output. This file is equivalent to SYS$OUTPUT. The file pointer, stderr, is associated with the terminal to report run-time errors. This file is equivalent to SYS$ERROR.

Also, three file descriptors exist that refer to the terminal. The file descriptor 0 is equivalent to SYS$INPUT, 1 is equivalent to SYS$OUTPUT, and 2 is equivalent to SYS$ERROR. For more information concerning file descriptors, refer to Chapter 4, UNIX System I/O Functions.

When performing I/O at the terminal, you can use Standard I/O functions and macros (specifying the pointers stdin, stdout, or stderr as arguments), you can use UNIX I/O functions (giving the corresponding file descriptor as an argument), or you can use the Terminal I/O functions and macros. There is no functional advantage of using one type of I/O over another; the Terminal I/O functions may save keystrokes due to the absence of arguments.

The following sections describe the Terminal I/O functions.

3.1 getchar

The **getchar** function reads a single character from the standard input (stdin).

The syntax of the function is as follows:

```
#include stdio

int getchar (void);
```

Additional Information

The **getchar** function returns EOF on end-of-file or error.

The **getchar** function is identical to **fgetc**(stdin).

3.2 gets

The **gets** function reads a line from the standard input (stdin).

The syntax of the function is as follows:

```
#include stdio

char *gets  (char *str);
```

Arguments

The argument *str* is a pointer to a character string used to hold the information fetched from stdin.

Additional Information

The newline character (\n) that ends the line is replaced by the function with an ASCII NUL character (\0). The function returns its argument, which is a pointer to a character string containing the acquired line. If an error occurs or if end-of-file is encountered before a newline is encountered, the function returns NULL, the null pointer value.

3.3 printf

The **printf** function performs formatted output from the standard output (stdout).

The syntax of the function is as follows:

```
#include stdio

int printf  (const char *format_spec, ...);
```

Arguments

The arguments for the **printf** function are as follows:

format_spec Contains characters to be written literally to the output or converted as specified in the . . . arguments.

. . . Represents optional expressions whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, the output sources may be omitted. Otherwise, the function call must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources. Conversion specifications are matched to output sources in simple left-to-right order.

An example of a conversion specification is as follows:

```
main()
{
   int  temp = 4, temp2 = 17;

   printf("The answers are %d, and %d.", temp, temp2);
}
```

Sample output from the previous example is as follows:

```
$ RUN  EXAMPLE RETURN
The answers are 4, and 17.
```

Additional Information

The **printf** function returns the number of characters written.

3.4 putchar

The **putchar** function writes a single character to the standard output (**stdout**) and returns the character.

The syntax of the function is as follows:

```
#include stdio

int putchar (char character);
```

Arguments

The argument *character* is an object of type **char**.

Additional Information

The function **putchar** returns EOF to designate output errors.

The function **putchar** is identical to **fputc**(character, stdout).

3.5 puts

The function **puts** writes a character string to the standard output (stdout), followed by a newline.

The syntax of the function is as follows:

```
#include stdio

int puts (char *str);
```

Arguments

The argument *str* is a pointer to a character string to be written to stdout.

Additional Information

The function does not copy the terminating NUL character to the output stream.

3.6 scanf

The function **scanf** performs formatted input from the standard input (stdin).

The syntax of the function is as follows:

```
#include stdio

int scanf  (const char *format_spec, ...);
```

Arguments

The arguments for the **scanf** function are as follows:

format_spec — Contains characters to be taken literally from the input or converted and placed in memory at the specified input_sources.

... — Represents optional expressions that are pointers to objects whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, these input pointers may be omitted. Otherwise, the function call must have exactly as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input_pointers. Conversion specifications are matched to input sources in simple left-to-right order.

An example of a conversion specification is as follows:

```
main()
{
   int  temp, temp2;

   scanf("%d %d", &temp, &temp2);
   printf("The answers are %d, and %d.", temp, temp2);
}
```

Sample input and output from the previous example is as follows:

```
$ RUN  EXAMPLE [RETURN]
4 17 [RETURN]
The answers are 4, and 17.
```

Additional Information

The function returns the number of successfully matched and assigned input items. If end-of-file is encountered, the function returns EOF (a preprocessor constant defined in the *stdio* definition module).

3.7 Program Examples

Example 3–1 illustrates the **printf** function.

Example 3–1: Output of the Conversion Specifications

```
/*  This program uses the printf function to  print the       *
 *  various conversion specifications and their effect on the  *
 *  output.                                                    */
                                     /*  Include proper module   *
                                      *   in case printf has to  *
                                      *   return EOF.            */
#include stdio

main()

{
   double val  = 123.3456e+3;
   char   c    = 'C';
   int    i    = -1500000000;
   char  *s    = "thomasina";

/*  Print the specification code, a colon, two tabs, and the   *
 *  formatted output value delimited by the angle bracket      *
 *  characters (< >).                                          */

   printf("%%9.4f:\t\t<%9.4f>\n", val);
   printf("%%9f:\t\t<%9f>\n", val);
   printf("%%9.0f:\t\t<%9.0f>\n", val);
   printf("%%-9.0f:\t\t<%-9.0f>\n\n", val);

   printf("%%11.6e:\t\t<%11.6e>\n", val);
   printf("%%11e:\t\t<%11e>\n", val);
   printf("%%11.0e:\t\t<%11.0e>\n", val);
   printf("%%-11.0e:\t\t<%-11.0e>\n\n", val);

   printf("%%11g:\t\t<%11g>\n", val);
   printf("%%9g:\t\t<%9g>\n\n", val);

   printf("%%d:\t\t<%d>\n", c);
   printf("%%c:\t\t<%c>\n", c);
   printf("%%o:\t\t<%o>\n", c);
   printf("%%x:\t\t<%x>\n\n", c);

   printf("%%d:\t\t<%d>\n", i);
   printf("%%u:\t\t<%u>\n", i);
   printf("%%x:\t\t<%x>\n\n", i);

   printf("%%s:\t\t<%s>\n", s);
   printf("%%-9.6s:\t\t<%-9.6s>\n", s);
   printf("%%-*.*s:\t\t<%-*.*s>\n", 9, 5, s);
   printf("%%6.0s:\t\t<%6.0s>\n\n", s);
}
```

Sample output from the previous example is as follows:

```
$ RUN  EXAMPLE [RETURN]
%9.4f:        <123345.6000>
%9f:          <123345.600000>
%9.0f:        <   123346>
%-9.0f:       <123346>

%11.6e:       <1.233456e+05>
%11e:         <1.233456e+05>
%11.0e:       <     1.e+05>
%-11.0e:      <1.e+05     >

%11g:         <     123346>
%9g:          <   123346>

%d:           <67>
%c:           <C>
%o:           <103>
%x:           <43>

%d:           <-1500000000>
%u:           <2794967296>
%x:           <a697d100>

%s:           <thomasina>
%-9.6s:       <thomas   >
%-*.*s:       <thoma    >
%6.0s:        <thomasina>
$
```

Chapter 4

UNIX I/O Functions and Macros

The UNIX I/O functions and macros access files by a file descriptor. A file descriptor is an integer that identifies the file. A file descriptor is declared as follows:

```
int  file_desc;
```

In this case, the identifier file_desc is the name of the file descriptor.

When you create a file using the UNIX I/O functions and macros, you can supply values for the following RMS file attributes:

- Allocation quantity
- Block size
- Default file extension
- Default file name
- File access context options
- File-processing options
- File sharing options
- Multiblock count
- Multibuffer count
- Maximum record size
- Record attributes
- Record format
- Record processing options

For more information concerning RMS, refer to the *Guide to VAX C*.

UNIX I/O functions such as **creat** associate the file descriptor with a file. Consider the following example:

```
file_desc =  creat("INFILE.DAT", 0, "rat=cr", "rfm=var");
```

This statement creates the file, INFILE.DAT, with mode argument 0, carriage-return control, variable-length records, and it associates the argument file_desc with the file. When the file is accessed for other operations, such as reading or writing, the file descriptor is used to refer to the file. For example:

```
write(file_desc, buffer, sizeof(buffer));
```

This statement writes the contents of the buffer to INFILE.DAT.

There may be circumstances when you should use UNIX I/O functions and macros instead of the Standard I/O functions and macros. For a detailed discussion of both forms of I/O and how they manipulate the RMS file formats, refer to Chapter 1, VAX C Run-Time Library Information.

4.1 Opening and Closing Files

The following sections describe the UNIX I/O functions that open and close files.

4.1.1 close

The **close** function closes the file associated with a file descriptor.

The syntax of the function is as follows:

```
#include unixio

int close (int file_desc);
```

Arguments

The argument *file_desc* is a file descriptor.

Additional Information

The **close** function returns 0 if the file is properly closed. It returns -1 if the file descriptor is undefined or if an error occurs while the file is being closed (for example, if the buffered data cannot be written out).

NOTE

Upon image exit, all buffered data is written to the file if it was opened for writing or update, and the file is closed.

4.1.2 creat

The **creat** function creates a new file.

The syntax of the function is as follows:

```
#include unixio

int creat (char *file_spec, unsigned int mode, ...);
```

Arguments

The arguments to the **creat** function are as follows.

file_spec	A NUL-terminated string containing any valid file specification.
mode	An **unsigned** value that specifies the file-protection mode; the compiler performs a bitwise AND operation on the mode and the complement of the current protection mode.

Modes can be constructed by using the bitwise OR operator (|) to mode combinations. The modes are as follows:

0400	OWNER:READ
0200	OWNER:WRITE
0100	OWNER:EXECUTE
0040	GROUP:READ
0020	GROUP:WRITE
0010	GROUP:EXECUTE
0004	WORLD:READ
0002	WORLD:WRITE
0001	WORLD:EXECUTE

When you supply a mode argument of zero, **creat** gives the file the user's default file protection.

The system is always given the same privileges as the owner. A WRITE privilege also implies a DELETE privilege.

. . . Represents an optional argument list of character strings of the form

```
"keyword = value", . . .  ,"keyword = value"
```

Keyword is an RMS (Record Management Services) field in the file access block (FAB) or record access block (RAB), and *value* is valid for assignment to that field. Some fields permit you to specify more than one value. In these cases, the values are separated by commas.

Table 4–1 lists the set of valid keywords and values.

Table 4–1: File Access Block and Record Access Block Keywords

Keyword	Value	Description
"alq = n"	decimal	Allocation quantity
"bls = n"	decimal	Block size
"ctx = bin"	decimal	No translation of '\n' to the terminal
"ctx = nocvt"	decimal	No conversion of FORTRAN carriage control bytes
"ctx = rec"	string	Force record mode access
"ctx = str"	string	Force stream mode access
"deq = n"	decimal	Default extension quantity
"dna = filespec"	string	Default filename string
"fop = val, val, . . . "		File processing options:
	ctg	Contiguous
	cbt	Contiguous-best-try
	tef	Truncate at end-of-file
	cif	Create if nonexistent
	sup	Supersede
	scf	Submit as command file on close
	spl	Spool to system printer on close
	tmd	Temporary delete
	tmp	Temporary (no file directory)
	nef	Not end-of-file
"fsz = n"	decimal	Fixed header size
"mbc = n"	decimal	Multiblock count
"mbf = n"	decimal	Multibuffer count
"mrs = n"	decimal	Maximum record size
"rat = val, val . . . "		Record attributes:
	cr	Carriage-return control
	blk	Disallow records to span block boundaries
	ftn	FORTRAN print control
	prn	Print file format
"rfm = val"		Record format:

Table 4–1 (Cont.): File Access Block and Record Access Block Keywords

Keyword	Value	Description
	fix	Fixed-length record format
	stm	RMS-11 stream record format
	stmlf	Stream format with line-feed terminator
	stmcr	Stream format with carriage-return terminator
	var	Variable-length record format
	vfc	Variable-length record with fixed control
	udf	Undefined
"rop = val"		Record processing operations:
	asy	Asynchronous I/O
	tmo	Timeout I/O
"shr = val"		File sharing options:
	del	Allows users to delete
	get	Allows users to read
	mse	Allows mainstream access
	nil	Prohibits file sharing
	put	Allows users to write
	upd	Allows users to update
	upi	Allows one or more writers
"tmo = n"	decimal	I/O timeout value

NOTE

You cannot share the default VAX C stream file I/O. If you wish to share files, you must specify "ctx=rec" to force record access mode. You must also specify the appropriate "shr" options depending upon the type of access you want.

Additional Information

If the file already exists, a version number one greater than any existing version is assigned to the file.

If the file did not previously exist, it is given the file protection that results from performing a bitwise AND on the mode argument and the complement of the current protection mask. The VAX C RTL opens the new file for reading and writing, and it returns the corresponding file descriptor. For more information concerning **umask** and **chmod**, refer to Chapter 11, System Functions.

The **creat** function returns an integer file descriptor. It returns –1 to indicate errors including protection violations, undefined directories, and conflicting file attributes.

See also **open**, **close**, **read**, **write**, and **lseek** in this chapter.

4.1.3 dup, dup2

The **dup** and **dup2** functions allocate a new descriptor that refers to a file specified by a file descriptor returned by **open**, **creat**, or **pipe** (refer to Chapter 10, Subprocess Functions).

The syntax of the functions is as follows:

```
#include unixio

int dup  (int file_desc1);
int dup2 (int file_desc1, int file_desc2);
```

Arguments

The arguments for the **dup** and **dup2** functions are as follows:

file_desc1 The file descriptor being duplicated.

file_desc2 The new descriptor to be assigned to the file designated by *file_desc1*.

Additional Information

Both functions return the new file descriptor. The **dup2** function causes its second argument to refer to the same file as its first argument.

Both functions return –1 if their arguments are invalid. The argument *file_desc1* is invalid if it does not describe an open file; *file_desc2* is invalid if the new descriptor cannot be allocated. If *file_desc2* is connected to an open file, that file is closed.

4.1.4 open

The **open** function positions the file at its beginning (byte 0).

The syntax of the function is as follows:

```
#include unixio
#include file

int open (char *file_spec, int flags, unsigned int mode, ...);
```

Arguments

The arguments for the **open** function are as follows:

file_spec — A NUL-terminated character string containing a valid file specification.

flags — Values defined in the *file* definition module and have the following meanings:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_NDELAY	Open for asynchronous input.
O_APPEND	Append on each write.
O_CREAT	Create a file if it does not exist.
O_TRUNC	Create a new version of this file.
O_EXECL	Error if attempting to create existing file.

These flags are set using the bitwise OR operator (|) to separate specified flags. Opening a file with the O_APPEND causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, **open** creates a new file by incrementing the version number by one, leaving the old version in existence. If O_CREAT is set and the named file does not exist, the VAX C RTL creates it with any attributes specified in the fourth and subsequent arguments (. . .). If O_EXECL is set with O_CREAT, then if the file already exists, the attempted open returns an error.

mode — Sets the file protection. Modes can be constructed by using the bitwise OR operator (|) to separate specified modes. The modes are as follows:

0400	OWNER:READ
0200	OWNER:WRITE
0100	OWNER:EXECUTE
0040	GROUP:READ
0020	GROUP:WRITE
0010	GROUP:EXECUTE
0004	WORLD:READ
0002	WORLD:WRITE
0001	WORLD:EXECUTE

When you supply a mode argument of zero, **open** gives the file the user's default file protection.

The system is always given the same privileges as the owner. A WRITE privilege also implies a DELETE privilege.

. . . — Represents an optional argument list of character strings of the following form:

```
"keyword = value, . . .  "
```

Keyword is an RMS (Record Management Services) field in the file access block (FAB) or record access block (RAB), and *value* is valid for assignment to that field. Some fields permit you to specify more than one value. In these cases, the values are separated by commas.

Table 4–1 lists the set of valid keywords and values.

Additional Information

The **open** function returns –1 if the file does not exist, if it is protected against reading or writing, or if the file, for any other reason, cannot be opened.

NOTE

> If you intend to do random writing to a file, the file *must* be opened for update by specifying a flags value of O_RDWR.

See also **creat**, **read**, **write**, **close**, **dup**, **dup2**, and **lseek** in this chapter.

4.2 Reading and Writing

The following sections describe the UNIX I/O functions that read from and write to files.

4.2.1 read

The **read** function reads bytes from a file and places them in a buffer.

The syntax of the **read** function is as follows:

```
#include unixio

int read (int file_desc, void *buffer, size_t nbytes);
```

Arguments

The arguments to the **read** function are as follows:

file_desc	A file descriptor. The specified file descriptor must refer to a file currently opened for reading.
buffer	The address of contiguous storage in which the input data is placed.
nbytes	The maximum number of bytes involved in the read operation.

Additional Information

The **read** function returns the number of bytes actually read. The return value does not necessarily equal nbytes. For example, if the input is from a terminal, at most one line of characters is read.

NOTE

> In general, the **read** function will not span record boundaries in a record file. A separate read must be done for each record.

A return value of 0 means that end-of-file was encountered. A return value of –1 indicates any sort of read error, including physical input errors, illegal buffer addresses, protection violations, undefined file descriptors, and so forth.

4.2.2 write

The **write** function writes a specified number of bytes from a buffer to a file.

The syntax of the **write** function is as follows:

```
#include unixio

int write (int file_desc, void *buffer, size_t nbytes);
```

Arguments

The arguments for the **write** function are as follows:

file_desc — A file descriptor. The specified file descriptor must refer to a file currently opened for writing or update.

buffer — The address of contiguous storage from which the output data is taken.

nbytes — The maximum number of bytes involved in the write operation.

Additional Information

The **write** function returns the number of bytes actually written. It returns –1 for errors, including undefined file descriptors, illegal buffer addresses, and physical I/O errors.

NOTE

- If the write is to an RMS record file and the buffer contains embedded newline characters, more than one record may be written to the file. Even if there are no embedded newline characters, if nbytes is greater than the maximum record size for the file, more than one record may be written to the file.
- If the write is to a mailbox and the third argument, nbytes, specifies a length of zero, an end-of-file message is written to the mailbox. For more information, refer to Chapter 10, Subprocess Functions.

4.3 Maneuvering in Files

The following sections describe the UNIX I/O functions that position the pointer within the file.

4.3.1 lseek

The **lseek** function positions a file to an arbitrary byte position and returns the new position as an **int**.

The syntax of the **lseek** function is as follows:

```
#include unixio

int lseek (int file_desc, int offset, int direction);
```

Arguments

The arguments for the **lseek** function are as follows:

file_desc	An integer returned by **open**, **creat**, **dup**, or **dup2**.
offset	Measured in bytes.
direction	Tells the function where to begin the offset; the new position is relative either to the beginning of the file (direction=SEEK_ABS), the current position (direction=SEEK_CUR), or the end of the file (direction=SEEK_END).

Additional Information

The **lseek** function can position a stream file on any byte offset but can position a record file only on record boundaries. The available Standard I/O functions always position a record file at its first byte, at the end-of-file, or on a record boundary. Therefore, the arguments given to **lseek** must specify either the beginning or end of the file, a zero offset from the current position (an arbitrary record boundary), or the position returned by a previous, valid **lseek** call.

The following call obtains the position of the current record in an RMS record file (which has the descriptor, file1):

```
/*   RELATIVE TO CURRENT POSITION         */

pos = lseek(file1, 0, 1)
```

The return value in pos can then be used later in the program (perhaps after the file has been repositioned by **write** or **read**) to return to this position, as in the following example:

```
/* POSITION RELATIVE TO BEGINNING          */

newpos = lseek(file1, pos, 0);
```

CAUTION

If, while accessing a stream file, you seek beyond the end-of-file and then write to the file, the **lseek** function creates a "hole" by filling the skipped bytes with zeros.

In general, for record files, **lseek** should only be directed to an absolute position that was returned by a previous valid call to **lseek** or to the beginning or end of a file. If a call to **lseek** does not satisfy these conditions, the results are unpredictable.

The **lseek** function returns –1 if the file descriptor is undefined or if you attempt to seek before the beginning of the file.

See also **open**, **creat**, **dup**, and **dup2** in this chapter; for **fseek**, refer to Chapter 2, Standard I/O Functions and Macros.

4.4 Additional UNIX I/O Functions and Macros

The following sections describe the UNIX I/O functions and macros used to perform various tasks.

4.4.1 fileno

The macro **fileno** returns an integer file descriptor that identifies the specified file.

The syntax of the macro **fileno** is as follows:

```
#include stdio

int  fileno(FILE *file_ptr);
```

Arguments

The argument *file_ptr* is a file pointer. For more information concerning file pointers, refer to Chapter 2, Standard I/O Functions and Macros.

Additional Information

VAX C implements **fileno** as a macro.

4.4.2 fstat, stat

The **fstat** and **stat** functions access information about the file descriptor or the file specification.

The syntax of the functions is as follows:

```
#include unixio
#include stat

void fstat (int file_desc, stat_t *buffer);

void stat  (char *file_spec, stat_t *buffer);
```

Arguments

The arguments for the **fstat** and **stat** functions are as follows:

file_desc *file_spec*	A file descriptor (*file_desc*) or a valid VMS or DEC/Shell file specification (*file_spec*). Read, write, or execute permission of the named file is not required, but all directories listed in the file specification leading to the file must be reachable. For more information concerning the DEC/Shell, refer to Chapter 1, VAX C Run-Time Library Information.
buffer	A pointer to a structure of type stat_t which is defined in the *stat* definition module. The argument receives information about the particular file. The members of the structure pointed to by buffer are as follows:

Member	Type	Definition
st_dev	unsigned	Pointer to physical device name
st_ino[3]	unsigned short	Three words to receive file id
st_mode	unsigned short	File "mode" (prot, dir, . . .)
st_nlink	int	For UNIX system compatibility only
st_uid	unsigned	Owner user id
st_gid	unsigned short	Group member: from st_uid
st_rdev	char*	UNIX system compatibility—always zero
st_size	unsigned	File size in bytes
st_atime	unsigned	File access time; always same as st_mtime
st_mtime	unsigned	Last modification time
st_ctime	unsigned	File creation time
st_fab_rfm	char	Record format
st_fab_rat	char	Record attributes
st_fab_fsz	char	Fixed header size
st_fab_mrs	unsigned	Record size

The structure member, st_mode, is the status information mode and is defined in the *stat* definition module. The st_mode bits are as follows:

Bits	Constant	Definition
0170000	S_IFMT	Type of file
0040000	S_IFDIR	Directory
0020000	S_IFCHR	Character special
0060000	S_IFBLK	Block special
0100000	S_IFREG	Regular
0030000	S_IFMPC	Multiplexed char special
0070000	S_IFMPB	Multiplexed block special
0004000	S_ISUID	Set user id on execution
0002000	S_ISGID	Set group id on execution
0001000	S_ISVTX	Save swapped text even after use
0000400	S_IREAD	Read permission, owner
0000200	S_IWRITE	Write permission, owner
0000100	S_IEXEC	Execute/search permission, owner

Additional Information

Upon successful completion, these functions return zero; otherwise, they return −1.

The **fstat** and **stat** functions do not work on remote network files.

4.4.3 getname

The **getname** function returns the file specification associated with a file descriptor.

The syntax of the **getname** function is as follows:

```
#include unixio

char *getname (int file_desc, char *buffer, ...);
```

Arguments

The arguments for the **getname** function are as follows:

file_desc	A file descriptor.
buffer	A pointer to a character string that is large enough to hold the file specification.
...	Represents an optional argument that can be either 1 or 0. If you specify 1, the **getname** function returns the file specification in VMS format. If you specify 0, the **getname** function returns the file specification in DEC/Shell format. If you do not specify this argument, the **getname** function returns the file name according to your current command language interpreter. For more information concerning DEC/Shell file specifications, refer to Chapter 1, VAX C Run-Time Library Information.

Additional Information

The **getname** function places the file specification in a buffer and returns the buffer's address. The buffer should be an array large enough to contain a fully qualified file specification (the maximum length is 256 characters). When an error occurs, **getname** returns 0.

4.4.4 isapipe

The **isapipe** function returns 1 if the specified file descriptor is associated with a mailbox, and 0 if it is not. For more information concerning mailboxes, refer to Chapter 10, Subprocess Functions.

The syntax of the **isapipe** function is as follows:

```
#include unixio

int isapipe (int file_desc);
```

Arguments

The argument *file_desc* is a file descriptor.

Additional Information

The **isapipe** function returns a value of –1 to indicate an error (for example, if the file descriptor is not associated with an open file).

4.4.5 isatty

The **isatty** function returns 1 if the specified file descriptor is associated with a terminal, and zero if it is not.

The syntax of the **isatty** function is as follows:

```
#include unixio

int isatty (int file_desc);
```

Arguments

The argument *file_desc* is a file descriptor.

Additional Information

The **isatty** function returns value of –1 to indicate an error (for example, if the file descriptor is not associated with an open file).

4.4.6 ttyname

The **ttyname** function returns a pointer to the NUL-terminated name of the terminal device associated with file descriptor zero, the default input device (stdin).

The syntax of the function is as follows:

```
#include unixio

char *ttyname (void);
```

Additional Information

The **ttyname** function is provided only for UNIX compatibility and has limited functionality in the VMS environment.

4.5 Program Examples

Example 4–1 illustrates the use of both a file pointer and a file descriptor to access a single file.

Example 4-1: I/O Using File Descriptors and Pointers

```
/*  The following example creates a file with variable-length  *
 *  records (rfm = var) and the carriage-return attribute      *
 *  (rat = cr).                                                *
 *                                                             *
 *  The program uses creat to create and open the file, and    *
 *  fdopen to associate the file descriptor with a file        *
 *  pointer.  After using the fdopen function, the file        *
 *  must be referenced using the Standard I/O functions.       */

#include stdio
#include unixio
#define ERROR 0
#define ERROR1 -1
#define BUFFSIZE 132

main()
{
   char buffer[BUFFSIZE];
   int  fildes;
   FILE *fp;

   if ((fildes = creat("data.dat",0,"rat=cr",
                       "rfm=var")) == ERROR1)
      {
         perror("FILE3: creat() failed\n");
         exit(2);
      }

   if ((fp = fdopen(fildes,"w")) == NULL)
      {
         perror("FILE3: fdopen() failed\n");
         exit(2);
      }
   while(fgets(buffer,BUFFSIZE,stdin) != NULL)
      if (fwrite(buffer,strlen(buffer),1,fp) == ERROR)
         {
            perror("FILE3: fwrite() failed\n");
            exit(2);
         }

   if (fclose(fp) == EOF)
      {
         perror("FILE3: fclose() failed\n");
         exit(2);
      }
}
```

Chapter 5

Character-Handling Functions and Macros

The functions and macros in this chapter fall into two categories: character classification and character conversion. The following sections describe each of these types of functions and macros.

5.1 Character Classification Macros

VAX C implements all character classification "functions" as preprocessor defined macros. Do not pass arguments to those macros which may cause side effects, such as arguments with the increment and decrement operators. For more information concerning macros, refer to *Guide to VAX C*.

The character classification macros take a single argument on which they perform a logical operation. The argument can have any value; that is, it does not have to be an ASCII character. However, the value of the argument is reduced to modulo 128 to give a 7-bit ASCII character. This value is used as the value of the argument. In the case of the macro **isascii**, the function determines if the argument is an ASCII character (0 through 177 octal). The other macros determine whether the argument is a particular type of ASCII character, such as a graphic character or digit.

For all macros, a positive return value indicates true. A return value of zero indicates false. The following tables show, for each ASCII character, which functions return true.

The following list assigns a number to each of the character classification macros:

Macro Number	Macro	Macro Number	Macro
1	**isalnum**	7	**islower**
2	**isalpha**	8	**isprint**
3	**isascii**	9	**ispunct**
4	**iscntrl**	10	**isspace**
5	**isdigit**	11	**isupper**
6	**isgraph**	12	**isxdigit**

Table 5–1 lists the numbers of the macros (as assigned in the previous list) that return the value true for each of the given ASCII characters. The numeric code represents the octal value of each of the ASCII characters.

Table 5–1: Character Classification Macro Return Values (ASCII Table)

ASCII Values	Macro Numbers	ASCII Values	Macro Numbers
NUL 00	3,4	@ 100	3,6,8,9
SOH 01	3,4	A 101	1,2,3,6,8,11,12
STX 02	3,4	B 102	1,2,3,6,8,11,12
ETX 03	3,4	C 103	1,2,3,6,8,11,12
EOT 04	3,4	D 104	1,2,3,6,8,11,12
ENQ 05	3,4	E 105	1,2,3,6,8,11,12
ACK 06	3,4	F 106	1,2,3,6,8,11,12
BEL 07	3,4	G 107	1,2,3,6,8,11
BS 10	3,4	H 110	1,2,3,6,8,11
HT 11	3,4,10	I 111	1,2,3,6,8,11
LF 12	3,4,10	J 112	1,2,3,6,8,11
VT 13	3,4,10	K 113	1,2,3,6,8,11
FF 14	3,4,10	L 114	1,2,3,6,8,11

Table 5–1 (Cont.): Character Classification Macro Return Values (ASCII Table)

ASCII Values	Macro Numbers	ASCII Values	Macro Numbers
CR 15	3,4,10	M 115	1,2,3,6,8,11
SO 16	3,4	N 116	1,2,3,6,8,11
SI 17	3,4	O 117	1,2,3,6,8,11
DLE 20	3,4	P 120	1,2,3,6,8,11
DC1 21	3,4	Q 121	1,2,3,6,8,11
DC2 22	3,4	R 122	1,2,3,6,8,11
DC3 23	3,4	S 123	1,2,3,6,8,11
DC4 24	3,4	T 124	1,2,3,6,8,11
NAK 25	3,4	U 125	1,2,3,6,8,11
SYN 26	3,4	V 126	1,2,3,6,8,11
ETB 27	3,4	W 127	1,2,3,6,8,11
CAN 30	3,4	X 130	1,2,3,6,8,11
EM 31	3,4	Y 131	1,2,3,6,8,11
SUB 32	3,4	Z 132	1,2,3,6,8,11
ESC 33	3,4	[133	3,6,8,9
FS 34	3,4	\ 134	3,6,8,9
GS 35	3,4	] 135	3,6,8,9
RS 36	3,4	ˆ 136	3,6,8,9
US 37	3,4	– 137	3,6,8,9
SP 40	3,8,10	?` 140	3,6,8,9
! 41	3,6,8,9	a 141	1,2,3,6,7,8,12
″ 42	3,6,8,9	b 142	1,2,3,6,7,8,12
# 43	3,6,8,9	c 143	1,2,3,6,7,8,12
$ 44	3,6,8,9	d 144	1,2,3,6,7,8,12

Table 5–1 (Cont.): Character Classification Macro Return Values (ASCII Table)

ASCII Values	Macro Numbers	ASCII Values	Macro Numbers
% 45	3,6,8,9	e 145	1,2,3,6,7,8,12
& 46	3,6,8,9	f 146	1,2,3,6,7,8,12
' 47	3,6,8,9	g 147	1,2,3,6,7,8
(50	3,6,8,9	h 150	1,2,3,6,7,8
) 51	3,6,8,9	i 151	1,2,3,6,7,8
* 52	3,6,8,9	j 152	1,2,3,6,7,8
+ 53	3,6,8,9	k 153	1,2,3,6,7,8
' 54	3,6,8,9	l 154	1,2,3,6,7,8
- 55	3,6,8,9	m 155	1,2,3,6,7,8
?. 56	3,6,8,9	n 156	1,2,3,6,7,8
/ 57	3,6,8,9	o 157	1,2,3,6,7,8
0 60	1,3,5,6,8,12	p 160	1,2,3,6,7,8
1 61	1,3,5,6,8,12	q 161	1,2,3,6,7,8
2 62	1,3,5,6,8,12	r 162	1,2,3,6,7,8
3 63	1,3,5,6,8,12	s 163	1,2,3,6,7,8
4 64	1,3,5,6,8,12	t 164	1,2,3,6,7,8
5 65	1,3,5,6,8,12	u 165	1,2,3,6,7,8
6 66	1,3,5,6,8,12	v 166	1,2,3,6,7,8
7 67	1,3,5,6,8,12	w 167	1,2,3,6,7,8

Table 5–1 (Cont.): Character Classification Macro Return Values (ASCII Table)

ASCII Values	Macro Numbers	ASCII Values	Macro Numbers
8 70	1,3,5,6,8,12	x 170	1,2,3,5,6,8
9 71	1,3,5,6,8,12	y 171	1,2,3,5,6,8
: 72	3,6,8,9	z 172	1,2,3,5,6,8
; 73	3,6,8,9	{ 173	3,6,8,9
< 74	3,6,8,9	| 174	3,6,8,9
= 75	3,6,8,9	} 175	3,6,8,9
> 76	3,6,8,9	?~ 176	3,6,8,9
? 77	3,6,8,9	DEL 177	3,4

The following sections describe the character classification macros. All of these macros have a single argument that is an object of type **char**.

5.1.1 isalnum

The **isalnum** macro returns a nonzero integer if its argument is one of the alphanumeric ASCII characters. Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int isalnum (char character);
```

5.1.2 isalpha

The **isalpha** macro returns a nonzero integer if its argument is an alphabetic ASCII character. Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int isalpha (char character);
```

5.1.3 isascii

The **isascii** macro returns a nonzero integer if its argument is any ASCII character. Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int  isascii (char character)
```

5.1.4 iscntrl

The **iscntrl** macro returns a nonzero integer if its argument is an ASCII DEL character (177 octal) or any nonprinting ASCII character (code less than 40 octal). Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int iscntrl (char character);
```

5.1.5 isdigit

The **isdigit** macro returns a nonzero integer if its argument is a decimal digit character (0–9). Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int isdigit (char character);
```

5.1.6 isgraph

The **isgraph** macro returns a nonzero integer if its argument is a graphic ASCII character. Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int isgraph (char character);
```

Additional Information

Graphic ASCII characters are those with octal codes greater than or equal to 41 (!) and less than or equal to 176 (?~). In other words, they comprise the set of printable characters minus the space.

5.1.7 islower

The **islower** macro returns a nonzero integer if its argument is a lowercase alphabetic ASCII character. Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int islower (char character);
```

5.1.8 isprint

The **isprint** macro returns a nonzero integer if its argument is any ASCII printing character (ASCII codes from 40 octal to 176 octal). Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int isprint (char character);
```

5.1.9 ispunct

The **ispunct** macro returns a nonzero integer if its argument is an ASCII punctuation character; that is, if it is nonalphanumeric and greater than 40 octal. Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int ispunct (char character);
```

5.1.10 isspace

The **isspace** macro returns a nonzero integer if its argument is white space; that is, if it is an ASCII space, tab (horizontal or vertical), carriage-return, form-feed, or newline character. Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int isspace (char character);
```

5.1.11 isupper

The **isupper** macro returns a nonzero integer if its argument is an uppercase alphabetic ASCII character. Otherwise, it returns zero.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int isupper (char character);
```

5.1.12 isxdigit

The **isxdigit** macro returns a nonzero integer if its argument is a hexadecimal digit (0–9, A–F, or a–f).

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int isxdigit (char character);
```

5.2 Character Conversion Functions and Macros

The character conversion functions and macros convert one type of character to another type. The following sections describe the character conversion functions.

5.2.1 ecvt, fcvt, gcvt

Each of the **ecvt**, **fcvt**, and **gcvt** functions converts its argument to a NUL-terminated string of ASCII digits and returns the address of the string. The strings are stored in a memory location created by the functions.

The syntax descriptions of the functions are as follows:

```
#include stdlib

char *ecvt (double value, int ndigit, int *decpt, int *sign);
char *fcvt (double value, int ndigit, int *decpt, int *sign);
char *gcvt (double value, int ndigit, char *buffer);
```

Arguments

The arguments for the **ecvt**, **fcvt**, and **gcvt** functions are as follows.

value An object of type **double** that is converted to a NUL-terminated string of ASCII digits.

ndigit The number of ASCII digits to be used in the converted string.

decpt Contains the position of the decimal point relative to the first character in the returned string. A negative **int** value means that the decimal point is *decpt* number of spaces to the left of the returned digits, spaces being filled with zeros; a zero value means that the decimal point is immediately to the left of the first digit in the returned string.

sign Contains an integral value that indicates whether the argument *value* is positive or negative. If the value is negative, the functions place a nonzero value at the address specified by argument sign. Otherwise, the functions assign zero to the address specified by argument sign.

buffer A storage location to hold the converted string.

Additional Information

The functions **ecvt** and **fcvt** return, by means of the argument *decpt*, the position of the decimal point relative to the first character in the returned string.

The function **gcvt** places the converted string in a buffer and returns its address buffer. If possible, **gcvt** produces *ndigit* significant digits in FORTRAN-F format, or if not possible, in E-format. Trailing zeros may be suppressed.

Repeated calls to these functions overwrite any existing string.

5.2.2 toascii

The **toascii** macro converts its argument, an 8-bit ASCII character, to a 7-bit ASCII character.

The syntax of the macro is as follows:

```
#include stdlib
#include ctype

int  toascii(char character)
```

Arguments

The argument *character* is an object of type **char**.

5.2.3 tolower, _tolower

The **tolower** function and **_tolower** macro convert their argument, an ASCII character, to lowercase. If the argument is not an uppercase character, it is returned unchanged.

The syntax descriptions of the function and macro are as follows:

```
#include stdlib
#include ctype

int tolower (char character);
int _tolower (char character);
```

Arguments

The argument *character* is an object of type **char**.

Additional Information

VAX C implements **tolower** as a function and **_tolower** as a macro. You only have to include the *ctype* definition module if you are using **_tolower**.

5.2.4 toupper, _toupper

The **toupper** function and **_toupper** macro convert their argument, an ASCII character, to uppercase. If the argument is not a lowercase character, it is returned unchanged.

The syntax descriptions of the function and macro are as follows:

```
#include stdlib
#include ctype

int toupper (char character);
int _toupper (char character);
```

Arguments

The argument *character* is an object of type **char**.

Additional Information

VAX C implements **toupper** as a function and **_toupper** as a macro. You only have to include the *ctype* definition module if you are using **_toupper**.

5.3 Program Examples

Example 5–1 illustrates the use of character classification macros.

Example 5–1: Character Conversion Macros

```
/* The following program uses the isalpha, isdigit, and       *
 * isspace macros to count the number of occurrences of       *
 * letters, digits and white space characters entered through *
 * the standard input (stdin).                                */

#include ctype
#include stdio
#include stdlib

main()
{
   char c;
   int i = 0, j = 0, k = 0;

   while ((c = getchar()) != EOF)
      {
         if (isalpha(c))
            i++;
         if (isdigit(c))
            j++;
         if (isspace(c))
            k++;
      }

   printf("Number of letters: %d\n", i);
   printf("Number of digits:  %d\n", j);
   printf("Number of spaces:  %d\n", k);
}
```

Sample input and output from this program are as follows:

```
$ RUN  EXAMPLE1 RETURN
I saw 35 men with mustaches on Christopher Street. RETURN
CTRL/Z
Number of letters: 39
Number of digits:  2
Number of spaces:  9
$
```

Example 5–2 illustrates the use of the **ecvt** function.

Example 5–2: Converting Double Values to an ASCII String

```
/*  This program uses the ecvt function to convert a double     *
 *  value to a string.  The program then prints the string.    */

#include stdio
#include stdlib

main()

{

   double val;                      /*  Value to be converted    */

                                    /*  Variables for sign and  *
                                     *   decimal place          */
   int sign, point;

                                    /*  Array for converted     *
                                     *   string                 */
   static char string[20];

   val = -3.1297830e-10;

   printf("original value: %e\n", val);
   strcpy(string,ecvt(val, 5, &point, &sign));
   printf("converted string: %s\n", string);
   if (sign)
      printf("value is negative\n");
   else printf("value is positive\n");
   printf("decimal point at %d\n", point);
}
```

The output from this program is as follows:

```
$ RUN  EXAMPLE2 RETURN
original value: -3.129783e-10
converted string: 31298
value is negative
decimal point at -9
$
```

Example 5–3 illustrates the use of functions **toupper** and **tolower**.

Example 5–3: Changing Characters to and from Uppercase Letters

```
/*  This program uses the functions toupper and tolower to     *
 *  convert uppercase to lowercase and lowercase to uppercase  *
 *  using input from the standard input (stdin).               */

#include ctype
#include stdio          /*  To use EOF identifier   */
#include stdlib

main()
{
   char c, ch;

   while ((c = getchar()) != EOF)
      {
         if (c >= 'A' && c <= 'Z')
            ch = tolower(c);
         else
            ch = toupper(c);
         putchar(ch);
      }
}
```

Sample input and output from this program are as follows:

```
$ RUN  EXAMPLE3 RETURN
LET'S GO TO THE stonewall INN. CTRL/Z
let's go to the STONEWALL inn.
$
```

Chapter 6

String- and List-Handling Functions and Macros

This chapter discusses functions that manipulate strings. Some of these functions concatenate strings; others search a string for specific characters or perform some other comparison, such as determining the equality of two strings.

6.1 strcat, strncat

The **strcat** and **strncat** functions concatenate *str_2* to the end of *str_1*.

The syntax descriptions of the functions are as follows:

```
#include string

char *strcat (char *str_1, const char *str_2);
char *strncat (char *str_1, const char *str_2, size_t maxchar);
```

Arguments

The arguments to the **strcat** and **strncat** functions are as follows:

str_1
str_2 — Must be NUL-terminated character strings.

maxchar — Specifies the maximum number of characters to concatenate from *str_2*, unless the **strncat** first encounters a NUL terminator in *str_2*. If *maxchar* is zero or negative, no characters are copied from *str_2*.

Additional Information

Both **strcat** and **strncat** return the address of the first argument, *str_1*, which is assumed to be large enough to hold the concatenated result.

If **strncat** reaches the specified maximum, it sets the next byte in *str_1* to NULL.

6.2 strchr, strrchr

The **strchr** and **strrchr** functions return, respectively, the address of the first or last occurrence of a given character in a NUL-terminated string.

The syntax descriptions of the functions are as follows:

```
#include string

char *strchr (const char *str, int character);
char *strrchr (const char *str, int character);
```

Arguments

The arguments to the **strchr** and **strrchr** functions are as follows:

str	A pointer to a NUL-terminated character string.
character	An object of type **char**.

Additional Information

The **strchr** and **strrchr** functions return zero if the character does not occur in the string, otherwise they return the address of the first (**strchr**) or last (**strrchr**) occurrence of the specified character.

6.3 strcmp, strncmp

The **strcmp** and **strncmp** functions compare two ASCII character strings and return a negative, zero, or positive integer, indicating that the ASCII values of the individual characters in the first string are less than, equal to, or greater than the values in the second string.

The syntax descriptions of the functions are as follows:

```
#include string

int strcmp (const char *str_1, const char *str_2);
int strncmp (const char *str_1, const char *str_2, size_t maxchar);
```

Arguments

The arguments to the **strcmp** and **strncmp** functions are as follows:

str_1
str_2 — Pointers to character strings.

maxchar — Specifies a maximum number of characters (beginning with the first) to search in both *str_1* and *str_2*.

If *maxchar* is zero or negative, no comparison is performed and zero is returned (the strings are considered equal).

Additional Information

The returned value is obtained by subtracting the characters at the first position where the two strings disagree.

With either function, the comparison is terminated when a NULL is encountered in one of the strings.

6.4 strcpy, strncpy

These functions copy all or part of *str_2* into *str_1*.

The syntax descriptions of the functions are as follows:

```
#include string

char *strcpy (char *str_1, const char *str_2);
char *strncpy (char *str_1, const char *str_2, size_t maxchar);
```

Arguments

The arguments to the **strcpy** and **strncpy** functions are as follows:

str_1 *str_2*	Pointers to character strings.
maxchar	Specifies the maximum number of characters to copy from *str_2* to *str_1*, up to but not including the NUL terminator or *str_2*.

Additional Information

The **strcpy** function copies *str_2* into *str_1*, stopping after copying *str_2*'s NUL character.

The function **strncpy** copies no more than *maxchar* characters from *str_2* to *str_1*, up to but not including the null terminator of *str_2*. If *str_2* contains less than *maxchar* characters, *str_1* is padded with null characters. If *str_2* contains greater than or equal to *maxchar* characters, as many characters as possible are copied to *str_1*.

Both functions return the address of *str_1*.

NOTE

> The argument *str_1* is not necessarily terminated by a null character.

6.5 strcspn, strspn, strpbrk

The **strcspn** function searches a string for a character in a specified set of characters. The **strpbrk** function searches a string for the occurrence of one of a specified set of characters. The **strspn** function searches a string for the occurrence of a character that is not in a specified set of characters.

The syntax of the functions is as follows:

```
#include string

char *strcspn (const char *str, const char *charset);

char *strpbrk (const char *str, const char *charset);

char *strspn (const char *str, const char *charset);
```

Arguments

The arguments to these functions are as follows:

str	A pointer to a character string. If the argument string is a null string, zero is returned.
charset	A pointer to a character string containing the characters for which the function may or may not search.

Additional Information

These functions scan the characters in the string, stop when they encounter a character found in *charset*, and return the length of the string's segment formed by characters found or not found in *charset*.

If all or no characters match in the character strings pointed to by *str* and *charset*, **strcspn** and **strspn** return the length of string. The **strpbrk** function returns the address of the first character in the string that is in the set, or NULL if no character is in the set.

6.6 strlen

The **strlen** function returns the length of a string of ASCII characters. The returned length does not include the terminating NUL character (\0).

The syntax of the function is as follows:

```
#include string

int strlen (const char *str);
```

Arguments

The argument *str* is a pointer to the character string.

6.7 strtod, atof

The **strtod** and **atof** functions convert a given string to a double-precision number.

These functions recognize an optional sequence of "white-space" characters (as defined by isspace in **ctype**), then an optional plus or minus sign, then a sequence of digits optionally containing a single decimal point, then an optional letter (e or E) followed by an optionally signed integer. The first unrecognized character ends the conversion.

The string is interpreted by the same rules that are used to interpret floating constants.

The syntax of the **strtod** and **atof** functions is as follows:

```
#include stdlib

double strtod (const char *nptr, char **endptr);

double atof (const char *nptr);
```

Arguments

The arguments to the **strtod** and **atof** functions are as follows:

nptr A pointer to the character string to be converted to a double-precision number.

endptr The address of an object into which will be stored the address of the first unrecognized character that terminates the scan. If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

Additional Information

The **strtod** and **atof** functions return the converted value. For **atof**, overflows resulting from the conversion are not accounted for. For **strtod**, overflows are accounted for:

- If the correct value would cause overflow, HUGE_VAL (with a plus or minus sign according to the sign of the value) is returned and int errno is set to ERANGE.
- If the correct value would cause underflow, zero is returned and errno is set to ERANGE.

If the string starts with an unrecognized character, **endptr* is set to *nptr*, and a zero value is returned.

The function call **atof(str)** is equivalent to **strtod(str,(char **)0)**, arithmetic exceptions not withstanding.

6.8 strtok

The **strtok** function locates text tokens in a given string. The text tokens are delimited by one or more characters from a separator string that you specify. The function keeps track of its position in the string between calls and, as successive calls are made, the function will work through the string, identifying the text token following the one identified by the previous call.

The syntax of the **strtok** function is as follows:

```
#include string

char *strstr (char *s1, const char *s2);
```

The first call to the **strtok** function returns a pointer to the initial character in the first token and writes a NUL character into *s1* immediately following the returned token. Each subsequent call (with the value of the first argument remaining NULL) returns a pointer to a subsequent token in the string originally pointed to by *s1*. When no tokens remain in the string, the **strtok** function returns a NULL pointer.

Arguments

The arguments to the **strtok** function are as follows:

s1 A pointer to a string containing zero or more text tokens.

s2 A pointer to a separator string consisting of one or more characters. The separator string may differ from call to call.

Additional Information

Tokens in *s1* are delimited by NUL characters inserted into *s1* by the **strtok** function. Therefore, *s1* cannot be a **const** object. The **strtok** function is non-reentrant since a static global variable must be used to maintain the starting address within *s1* of subsequent calls to **strtok** with a NULL first argument.

6.9 strto, atoi, atol

These functions convert strings of ASCII characters to the appropriate numeric values.

The syntax descriptions of the functions are as follows:

```
#include stdlib

int atoi (const char *nptr);

long int atol (const char *nptr);

long int strtol (const char *nptr, char **endptr, int base);
```

Arguments

The arguments to the functions are as follows:

nptr A pointer to the character string to be converted to a long.

endptr The address of an object into which will be stored a pointer to a pointer to the first unrecognized character encountered in the conversion process (that is, the character that follows the last character in the string being converted). If *endptr* is a NUL pointer, the address of the first unrecognized character is not retained.

base The value, 2 through 36, to be used as the base for the conversion. Leading zeros after the optional sign are ignored, and 0x or 0X is ignored if the base is 16.

If the base is 0, the sequence of characters is interpreted by the same rules used to interpret an integer constant: after the optional sign, a leading zero indicates octal conversion, a leading 0x or 0X indicates hexadecimal conversion, and any other combination of leading characters indicates decimal conversion.

Additional Information

The functions recognize strings in various formats, depending on the value of the base, as follows:

- The **strtol** function ignores any leading white-space characters (as defined by isspace in **ctype**) in the given string. It recognizes an optional plus or minus sign, then a sequence of digits or letters that may represent an integer constant according to the value of the base. The first unrecognized character ends the conversion.

```
[white-spaces][+|-]digits
```

- The functions **atoi** and **atol** are functionally equivalent in VAX C.
- The **atoi** and **atol** functions do not account for overflows resulting from the conversion.
- The **strtol** function returns the converted value. If the correct value would cause overflow, LONG_MAX or LONG_MIN (according to the sign of the value) is returned and errno is set to ERANGE. If the string starts with an unrecognized character, **endptr* is set to *nptr,* and a zero value is returned.
- Truncation from long to int can take place upon assignment or by an explicit cast (arithmetic exceptions not withstanding). The function call **atol (str)** is equivalent to **strtol (str, (char**)0, 10)**. Similarly, the function call **atoi (str)** is equivalent to **(int) strtol (str, (char**)0, 10)**.

6.10 strtoul

The **strtoul** function converts the initial portion of the string pointed to by *nptr* to an unsigned long integer.

The syntax of the function **strtoul** is as follows:

```
#include stdlib

unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

Arguments

The arguments to the **strtoul** function are as follows:

nptr A pointer to the character string to be converted to a long.

endptr The address of an object into which will be stored a pointer to a pointer to the first unrecognized character encountered in the conversion process (that is, the character that follows the last character in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base The value, 2 through 36, to be used as the base for the conversion. Leading zeros after the optional sign are ignored, and 0x or 0X is ignored if the base is 16.

If the base is 0, the sequence of characters is interpreted by the same rules used to interpret an integer constant: after the optional sign, a leading zero indicates octal conversion, a leading 0x or 0X indicates hexadecimal conversion, and any other combination of leading characters indicates decimal conversion.

Additional Information

The **strtoul** function returns the converted value, if any. If no conversion is performed, zero is the returned value. If overflow occurs, errno is set to erange and the return value is ULONG_MAX as defined in the standard include module *stdlib*.

6.11 Accessing Binary Data

The functions discussed in the following sections allow you to copy buffers containing binary data.

6.11.1 memchr

The **memchr** function locates the first occurrence of the specified byte within the initial *size* bytes of a given object. It returns a pointer to the first occurrence of the character. If the character does not occur in the identified character string, the **memchr** function returns a NUL pointer.

The syntax of the **memchr** function is as follows:

```
#include string

int memchr (const void *s1, int c, size_t size);
```

Arguments

Arguments to the **memchr** function are as follows:

s	A pointer to the object to be searched.
c	The byte value to be located.
size	The length of the object to be searched.

Additional Information

Unlike **strchr**, **memchr** does not stop when a NUL character is encountered.

6.11.2 memcmp

The **memcmp** function compares two objects byte by byte. The compare operation starts with the first byte in each object. It returns an integer less than, equal to, or greater than 0, depending on whether the lexical value of the first object is less than, equal to, or greater than that of the second object.

The syntax of the **memcmp** function is as follows:

```
#include string

int memcmp (const void *s1, const void *s2, size_t size);
```

Arguments

Arguments to the **memcmp** function are as follows:

s1	A pointer to the first object.
s2	A pointer to the second object.
size	The length of the objects to be compared.

Additional Information

The **memcmp** function uses native character comparison. The sign of the value returned is determined by the sign of the difference between the values of the first pair of unlike bytes in the objects being compared. Unlike the **strcmp** function, the **memcmp** function does not stop when a NUL character is encountered.

6.11.3 memcpy, memmove

The **memcpy** and **memmove** functions copy a specified number of bytes from one object to another.

The syntax of the functions is as follows:

```
#include string

void *memcpy (void *s1, const void *s2, size_t size);

void *memmove (void *s1, const void *s2, size_t size);
```

Arguments

Arguments to the functions are as follows:

s1	A pointer to the first object.
s2	A pointer to the second object.
size	The length of the object to be copied.

Additional Information

The **memcpy** function copies *size* bytes from object 2 to object 1; it does not check for the overflow of the receiving memory area (object 1). It returns the value of *s1*, which is a pointer. Unlike the **strcpy** function, the **memcpy** function does not stop when a NUL character is encountered.

The **memmove** function is functionally equivalent to the **memcpy** function in VAX C.

6.11.4 memset

The **memset** function sets a specified number of bytes in a given object to a given value.

The syntax of the **memset** function is as follows:

```
#include string

void *memset (void *s, char value, size_t size);
```

Arguments

Arguments to the **memset** function are as follows:

s	Array pointer.
value	The value to be placed in *s*.
size	The number of bytes to be placed in *s*.

Additional Information

The **memset** function returns *s*. It does not check for the overflow of the receiving memory area pointed to by *s*.

6.12 Accessing Variable Length Argument Lists

The set of functions and macros defined and declared in the **varargs** and the **stdarg** definition module provides a portable method of accessing variable length argument lists. For example, VAX C functions such as **printf** and **execl** use variable length argument lists. User-defined functions with variable length argument lists that do not use **varargs** are not portable due to the different argument passing conventions of various machines.

The argument *va_alist*, the definition va_dcl, and the type va_list, are used to declare the argument list and the variable that is used to traverse the list. The identifier *va_alist* is a parameter in the function definition; va_dcl declares the parameter *va_alist*, a declaration which is not terminated with a semicolon (;); and the type va_list is used in the declaration of the variable used to traverse the list. You must declare at least one variable of type va_list when using **varargs**. The syntax of these names and declarations is as follows:

```
function_name(va_alist)
va_dcl
{
   va_list  ap;
      .
      .
      .
```

In order to use the **varargs** functions and macros, you must include the **varargs** definition module with the following preprocessor directive:

```
#include varargs
```

The following sections describe the **varargs** functions and macros.

6.12.1 va_arg

The **va_arg** macro is used to return the next item in the argument list.

The syntax of the macro is as follows:

```
#include stdarg  or #include varargs

type va_arg (va_list ap, type);
```

Arguments

The arguments to the **va_arg** macro are as follows:

ap Must always be declared and used as shown in the syntax description.

type A data type that is used to determine the size of the next item in the list. An argument list can contain items of varying sizes, but the calling routine must determine what type of argument is expected since it cannot be determined at run time.

NOTE

In VMS, all items in an argument list are aligned on the longword boundary. If you attempt to access an item in an argument list by using the **sizeof** operator, and that item is smaller than a longword (types **short** and **char**, for instance), you will be positioned in the middle of the longword increment and the return value will be incorrect. VAX C correctly aligns the argument pointer on the next longword before reading the next argument. This macro is responsible for proper incrementation involving elements of types **short** and **char**.

Also, when accessing argument lists, especially those passed to a subroutine (written in VAX C) by a program written in another programming language, consider the implications of the VAX Calling Standard. For more information concerning the VAX Calling Standard refer to the *Guide to VAX C*.

Additional Information

The **va_arg** macro will always interpret the object at the address specified by the *list-incrementor* according to the type *type*. If there is no corresponding actual argument, the behavior is undefined.

6.12.2 va_count

The **va_count** macro returns the number of longwords in the argument list.

The syntax of the macro is as follows:

```
#include varargs

void va_count(int count);
```

Arguments

The argument *count* is mandatory. The **va_count** macro places the number of longwords in the argument list into *count*.

Additional Information

The value returned in *count* is the number of longwords in the function argument block not counting the count field itself.

If the argument list contains items whose storage requirements are a longword of memory or less, the number in the argument *count* is also the number of arguments. However, if the argument list contains items of type **double** or data structures, *count* must be interpreted to obtain the number of arguments in the list.

This macro is VAX C specific and is not portable.

6.12.3 va_end

The macro **va_end** finishes the *varargs* session.

The syntax of the macro **va_end** is as follows:

```
#include stdarg or #include varargs

void va_end (va_list ap);
```

Arguments

The argument *ap* is the object that was used to traverse the argument list length. You must always declare and use the argument *ap* as shown in the syntax description.

Additional Information

You can execute multiple traversals of the argument list, each delimited by **va_start** . . . **va_end**. This macro will set *ap* equal to NULL.

6.12.4 va_start, va_start_1

The **va_start** and **va_start_1** functions are used to initialize a variable to the beginning of the argument list.

The syntax descriptions of the functions using **varargs** are as follows:

```
#include varargs

void va_start (va_list ap);

void va_start_1(va_list ap, int offset);
```

Arguments

The arguments to the **va_start** and **va_start_1** functions are as follows:

ap An object pointer. You must always declare and use the argument *ap* as shown in the syntax description.

offset Represents the number of bytes by which *ap* is to be incremented so that it points to a subsequent argument within the list (that is, not to the start of the argument list). Using a nonzero offset can initialize *ap* to the address of the first of the optional arguments that follow a number of fixed arguments.

Additional Information

The **va_start** function is called to initialize the variable *ap* to the beginning of the argument list.

The **va_start_1** function is called to initialize *ap* to the address of an argument that is preceded by a known number of defined arguments. For example, a VAX C RTL function which contains a variable-length argument list offset from the beginning of the entire argument list is **printf**. The variable-length argument list is offset by the address of the formatting string.

Arguments of types **char** and **short** use a full longword of memory when they are present in argument lists; arguments of type **float** use two longwords because they are converted to type **double**.

NOTE

> When accessing argument lists, especially those passed to a subroutine (written in VAX C) by a program written in another programming language, consider the implications of the VAX Calling Standard. For more information concerning the VAX Calling Standard refer to the *Guide to VAX C*.

The function **va_start_1** is VAX C specific and is not portable.

The syntax descriptions of the **va_start** function using stdargs, as defined in the draft proposed ANSI standard, are as follows:

```
#include stdargs

void va_start(va_list ap, parmN)
```

Arguments

ap — An object pointer. You must always declare and use the argument *ap* as shown in the syntax description.

parmN — The name of the last of the known fixed arguments.

Additional Information

The pointer *ap* is initialized to point to the first of the optional arguments that follow *parmN* in the argument list. This version of **va_start** should always be used in conjunction with functions that are declared and defined with function prototypes.

6.12.5 vprintf, vfprintf, vsprintf

The **vprintf**, **vfprintf**, and **vsprintf** functions print formatted output based on an argument list.

These functions are the same as the **printf**, **fprintf**, and the **sprintf** functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list that has been initialized by the macro **va_start** (and possibly subsequent va_arg calls).

The syntax of the **vprintf, vfprintf** and **vsprintf** functions is as follows:

```
#include stdio
#include stdarg

int vprintf  (const char *format, va_list arg);
int vfprintf (FILE *file_ptr, const char *format, va_list arg);
int vsprintf (char *s, const char *format, va_list arg);
```

Arguments

The arguments to the **vfprintf** and **vsprintf** functions are as follows:

file_ptr	A pointer to a file.
format	A format specification.
arg	A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.
str	A pointer to a string.

Additional Information

The **vprintf, vfprintf,** and **vsprintf** functions return the number of characters transmitted or a negative value if an output error occurs.

6.13 Program Examples

Example 6–1 illustrates the use of **strcat** and **strncat**.

Example 6–1: Concatenation of Two Strings

```
/*  This example uses strcat and strncat to concatenate two     *
 *  strings.                                                    */

#include stdio

main()
{
   static char string1[] = "Concatenates ";
   static char string2[] = "two strings ";
   static char string3[] = "up to a maximum number of \
characters.";

   static char string4[] = "imum number of characters.";

   printf("strcat:\t%s\n", strcat(string1, string2));
   printf("strncat (-1):\t%s\n", strncat(string1, string3, -1));
   printf("strncat (11):\t%s\n", strncat(string1, string3, 11));
   printf("strncat (40):\t%s\n", strncat(string1, string4, 40));
}
```

Sample output from this program is as follows:

```
$ RUN  EXAMPLE1 RETURN
strcat: Concatenates two strings
strncat (-1): Concatenates two strings
strncat (11): Concatenates two strings up to a max
strncat (40): Concatenates two strings up to a maximum number of characters.
$
```

Example 6–2 illustrates the use of **strcspn**.

Example 6–2: Four Arguments to the strscpn Function

```
/*  The next example shows how strcspn interprets four          *
 *  different kinds of arguments.                               */

#include stdio

main()
{
   FILE *outfile;
   outfile = fopen("strcspn.out", "w");

   fprintf(outfile, "strcspn with null string: %d\n",
                     strcspn("abcdef", ""));

   fprintf(outfile, "strcspn with null string: %d\n",
                     strcspn("", "abcdef"));

   fprintf(outfile, "strcspn(\"xabc\", \"abc\"): %d\n",
                     strcspn("xabc", "abc"));

   fprintf(outfile, "strcspn(\"abc\", \"def\"): %d\n",
                     strcspn("abc", "def"));
}
```

Sample output, to the file strcspn.out, is as follows:

```
$ RUN  EXAMPLE2 RETURN

strcspn with null charset:  6
strcspn with null string:  0
strcspn(xabc,abc):  1
strcspn(abc,def):  3
```

Example 6–3 illustrates the use of the *varargs* definition module.

Example 6–3: The *varargs* Functions and Macros

```
/*  This program uses the varargs functions, macros, and       *
 *  definitions to implement the VAX C Run-Time Library        *
 *  function execl.                                            */

#include varargs                     /* Include proper module    */

execl(va_alist)                      /* Use the identifier       */
va_dcl                               /* Declare the parameter    */
                                     /* NOTE: No (;) with va_dcl */
{
   va_list   incrmtr;                /* Declare list incrementor */
   char        *file;                /* Declare a file           */
   char    *args[100];               /* Array to store arguments */
   int     noargs = 0;               /* Define "last argument"   */

   va_start(incrmtr);                /* Begin the session        */
   file = va_arg(incrmtr, char*);    /* First arg placed in file */
                                     /* Place args in array      */
   while(args[noargs++] = va_arg(incrmtr, char*))  /* User provided argument
                                                      list must terminate with
                                                      a zero */
      ;
   va_end(incrmtr);                  /* End varargs session      */
   return execv(file, args);         /* Return proper values     */
}
```

Chapter 7

Math Functions

This chapter describes the mathematical functions that are included in the VAX C Run-Time Library. To help you detect run-time errors, the *errno* definition module defines the following two symbolic values that are returned by many (but not all) of the mathematical functions:

- EDOM indicates that an argument is inappropriate; that is, the argument is not within the function's domain.
- ERANGE indicates that a result is out of range; that is, the argument is too large to be represented by the machine.

When using the math functions, you can check the external variable errno for either or both of these values and take the appropriate action if an error has occurred.

The following program example checks the variable errno for the value EDOM, which would indicate that a negative number was specified as input to the function **sqrt**:

```
#include errno
#include math
#include stdio

main()
{
   double input, square_root;

   printf("Enter a number: ");
   scanf("%le", &input);
   errno = 0;
   square_root = sqrt(input);
```

```
    if (errno == EDOM)
       perror("Input was negative");
    else
       printf("Square root of %e = %e\n",
               input, square_root);
}
```

If you did not check errno for this symbolic value, the **sqrt** function would return zero when a negative number was entered. For more information concerning the *errno* definition module, refer to Chapter 8, Error-Handling Functions.

The following sections describe the math functions.

7.1 abs, fabs

The **abs** function returns the absolute value of an integer. The **fabs** function returns the absolute value of a floating-point value.

The syntax descriptions of the functions are as follows:

```
#include math

int abs (int integer);
double fabs (double x);
```

7.2 acos

The **acos** function returns a value in the range zero to pi, which is the arc cosine of its radian argument.

The syntax of the function is as follows:

```
#include math

double acos (double x);
```

Additional Information

When $|x| > 1$, the value of acos(x) is zero and the **acos** function sets errno to EDOM.

7.3 asin

The **asin** function returns a value in the range –pi/2 to pi/2, which is the arc sine of its radian argument.

The syntax of the function is as follows:

```
#include math

double asin (double x);
```

Additional Information

When |x| > 1, the value of asin(x) is zero and the **asin** function sets errno to EDOM

7.4 atan

The **atan** function returns a value in the range –pi/2 to pi/2, which is the arc tangent of its radian argument.

The syntax of the function is as follows:

```
#include math

double atan (double x);
```

7.5 atan2

The **atan2** function returns a value in the range –pi to pi. The returned value is the arc tangent of y/x, where y and x are the two arguments.

The syntax of the function is as follows:

```
#include math

double atan2 (double y, double x);
```

7.6 cabs, hypot

The **cabs** and **hypot** functions return:

```
sqrt(x*x + y*y)
```

The syntax descriptions of the functions are as follows:

```
#include math

double cabs (cabs_t z);
double hypot (double x, double y);
```

Additional Information

The type cabs_t is defined in the standard include module *math.h* as follows:

```
typedef struct {double x,y;} cabs_t;
```

7.7 ceil

The **ceil** function returns (as a **double**) the smallest integer that is greater than or equal to its argument.

The syntax of the function is as follows:

```
#include math

double ceil (double x);
```

7.8 cos

The **cos** function returns the cosine of its radian argument.

The syntax of the function is as follows:

```
#include math

double cos (double x);
```

7.9 cosh

The **cosh** function returns the hyperbolic cosine of its argument.

The syntax of the function is as follows:

```
#include math
double cosh (double x);
```

7.10 exp

The **exp** function returns the base e raised to the power of the argument.

The syntax of the function is as follows:

```
#include math
double exp (double x);
```

Additional Information

If an overflow occurs, **exp** returns the largest possible floating-point value and sets errno to ERANGE. The constant HUGE in the *math* definition file is defined to be the largest possible floating-point value.

7.11 floor

The **floor** function returns (as a **double**) the largest integer that is less than or equal to its argument.

The syntax of the function is as follows:

```
#include math
double floor (double x);
```

7.12 fmod

The **fmod** function computes the floating-point remainder of the first argument to **fmod** divided by the second. If the quotient cannot be represented, the behavior is undefined.

The syntax of the **fmod** function is as follows:

```
#include math

double fmod (double x, double y);
```

Additional Information

The **fmod** function returns *x* if *y* is zero. Otherwise, it returns the value f, which has the same sign as *x*, such that x ==i * y + f for some integer i, where the magnitude of f is less than the magnitude of *y*.

7.13 frexp

The **frexp** function returns the mantissa of a **double** value.

The syntax of the function is as follows:

```
#include math

double frexp (double value, int *eptr);
```

Arguments

The arguments to the **frexp** function are as follows:

value An object of type **double**.

eptr A pointer to an **int**, to which **frexp** returns the exponent.

7.14 ldexp

The **ldexp** function returns its first argument multiplied by 2 raised to the power of its second argument; that is, $x(2^e)$.

The syntax of the function is as follows:

```
#include math
double ldexp  (double x, int e);
```

Arguments

x A base value, of type double, that is to be multiplied by 2^e.

e The integral exponent value to which 2 is raised.

Additional Information

If underflow occurs, **ldexp** returns zero, and if overflow occurs, it returns the largest possible value of the appropriate sign. In both cases, the function sets errno to ERANGE. The constant HUGE is defined in the *math* definition module to be the largest possible value of the appropriate sign.

7.15 ldiv, div

The **ldiv** and **div** functions return the quotient and the remainder after the division of their arguments.

```
#include stdlib
ldiv_t ldiv(long int numer, long int denom);
div_t div(int numer, int denom);
```

Arguments

numer A numerator of type **long int** or **int**.

denominator A denominator of type **long int** or **int**.

Additional Information

The types div_t and ldiv_t are defined in the standard include module *stdlib* as follows:

```
struct DIV_T
     {
        int quot, rem;
     };
typedef struct DIV_T div_t;

struct LDIV_T
     {
        long quot, rem;
     };

typedef struct LDIV_T ldiv_t;
```

The functions **ldiv** and **div** are functionally equivalent in VAX C.

7.16 labs

The **labs** function returns the absolute value of an integer as a **long int**.

The syntax of the function **labs** is as follows:

```
#include stdlib
long int labs(long int j);
```

7.17 log, log10

The **log** and **log10** functions return the logarithm of their arguments.

The syntax descriptions of the functions are as follows:

```
#include math

double log (double x);
double log10 (double x);
```

Additional Information

The **log** function returns the natural (base e) logarithm of the argument, which must be of type **double**; the returned value is also **double**. The **log10** function returns the **double** base 10 logarithm of its **double** argument.

If the argument is zero or negative, the functions return zero and set errno to EDOM.

7.18 modf

The **modf** function returns the positive fractional part of its first argument and assigns the integral part, expressed as a double, to the object whose address is specified by the second argument.

The syntax of the function is as follows:

```
#include math

double modf (double value, double *iptr);
```

Arguments

The arguments to the **modf** function are as follows:

value	Must be an object of type **double**.
iptr	A pointer to an object of type **double**.

7.19 pow

The **pow** function returns the first argument raised to the power of the second argument.

The syntax of the function is as follows:

```
#include math

double pow (double base, double exp);
```

Arguments

base A value of type double that is to be raised to a power.

exp The exponent to which the power base is to be raised.

Additional Information

Both arguments must be **double** and the returned value is **double**. If the result overflows, **pow** returns the largest possible floating-point value and sets errno to ERANGE. It returns zero and sets errno to EDOM under the following conditions:

- If both arguments are zero.
- If *exp* is negative and nonintegral.
- If *base* is negative and *exp* is nonintegral.

The constant HUGE is defined in the *math* definition module to be the largest possible value.

7.20 rand, srand

The **rand** and **srand** functions return pseudorandom numbers in the range zero to $2^{31}-1$.

The syntax descriptions of the functions are as follows:

```
#include math

int rand(void);

int srand (int seed);
```

Additional Information

The **rand** function uses a multiplicative congruential random number generator with a repeat factor (period) of 2^{32}. The random number generator is reinitialized by calling **srand** with the argument 1, or it can be set to a specific point by calling **srand** with any other number.

7.21 sin

The **sin** function returns the sine of its radian argument.

The syntax of the function is as follows:

```
#include math
double sin (double x);
```

Additional Information

Both the argument and the sine value must be an object of type **double**.

7.22 sinh

The **sinh** function returns the hyperbolic sine of its argument.

The syntax of the function is as follows:

```
#include math
double sinh (double x);
```

Additional Information

Both the argument and the returned hyperbolic sine value must be an object of type **double**.

The value of sinh(x), if it causes an overflow, is a **double** value with the largest possible magnitude and the appropriate sign.

7.23 sqrt

The **sqrt** function returns the square root of its argument.

The syntax of the function is as follows:

```
#include math
double sqrt (double x);
```

Additional Information

The argument and the returned value are both objects of type **double**.

Additional Information

If x is negative, the **sqrt** function returns zero and sets errno to EDOM.

7.24 tan

The **tan** function returns a **double** value that is the tangent of its radian argument.

The syntax of the function is as follows:

```
#include math

double tan (double x);
```

Additional Information

The value of tan(x) at its "singular points" (. . . –3pi/2,–pi/2,pi/2 . . .) is the largest possible **double** value HUGE, and the **tan** function sets errno to ERANGE.

7.25 tanh

The **tanh** function returns a **double** value that is the hyperbolic tangent of its **double** argument.

The syntax of the function is as follows:

```
#include math

double tanh (double x);
```

7.26 Program Examples

Example 7–1 illustrates the functionality of the **tan**, **sin**, and **cos** functions.

Example 7–1: Calculating and Verifying a Tangent Value

```
/*  This example uses two functions --- mytan and main ---     *
 *  to calculate the tangent value of a number, and to check   *
 *  the calculation using the sin and cos functions.           */

#include math                    /*  Include modules           */
#include stdio

/*  This function is used to calculate the tangent using the   *
 *  sin and cos functions.                                     */

double mytan(x)
double x;
{
   double  y, y1, y2;

   y1 = sin (x);
   y2 = cos (x);

   if (y2 == 0)
      y = 0;
   else
      y = y1 / y2;

   return y;
}
main()
{
   double  x;
                                  /*  Print values: compare    */
   for (x=0.0; x<1.5; x += 0.1)
      printf("tan of %4.1f = %6.2f\t%6.2f\n", x, mytan(x), tan(x));
}
```

Sample output from the previous example is as follows:

```
  RUN  EXAMPLE [RETURN]
tan of  0.0 =   0.00        0.00
tan of  0.1 =   0.10        0.10
tan of  0.2 =   0.20        0.20
tan of  0.3 =   0.31        0.31
tan of  0.4 =   0.42        0.42
tan of  0.5 =   0.55        0.55
tan of  0.6 =   0.68        0.68
tan of  0.7 =   0.84        0.84
tan of  0.8 =   1.03        1.03
tan of  0.9 =   1.26        1.26
tan of  1.0 =   1.56        1.56
tan of  1.1 =   1.96        1.96
tan of  1.2 =   2.57        2.57
tan of  1.3 =   3.60        3.60
tan of  1.4 =   5.80        5.80
$
```

Chapter 8

Error-Handling Functions

When an error occurs during a call to any of the VAX C Run-Time Library functions, the function returns an unsuccessful status and sets the external variable, errno, to a value which indicates the reason for the failure. In this way, variable errno is useful in determining the cause of a run-time error.

The *errno* definition module declares the errno variable and symbolically defines the possible errno values. By including the *errno* definition module in your program, you can check for specific values after a function call. Table 8–1 lists the symbolic values that can be assigned to errno.

Table 8–1: Errno Symbolic Values

Symbolic Constant	Description
EPERM	Not owner
ENOENT	No such file or directory
ESRCH	No such process
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address
E2BIG	Argument list too long
ENOEXEC	Exec format error
EBADF	Bad file number
ECHILD	No child processes

Table 8–1 (Cont.): Errno Symbolic Values

Symbolic Constant	Description
EAGAIN	No more processes
ENOMEM	Not enough memory
EACCESS	Permission denied
EFAULT	Bad address
ENOTBLK	Block device required
EBUSY	Mount devices busy
EEXIST	File exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
ENFILE	File table overflow
EMFIL	Too many open files
ENOTTY	Not a typewriter
ETXTBSY	Text file busy
EFBIG	File too big
ENOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read-only file system
EMLINK	Too many links
EPIPE	Broken pipe
EDOM	Math argument
ERANGE	Result too large
EWOULDBLOCK	File I/O buffers are empty
EVMSERR	VMS-specific error code nontranslatable error

The errno values can also be translated to a message, similar to that found in UNIX systems, by the **perror** function. If **perror** cannot translate the errno value, it prints the following message, followed by the VMS error message associated with the value.

```
%s:non-translatable vms error code: xxxxxx vms message:
```

In the template, %s is the string you supply to **perror**; *xxxxxx* is the VMS message number.

The VMS error code is available in the vaxc$errno variable and can be examined in user programs. The vaxc$errno variable is declared in the *errno* definition module.

The following sections describe the Error-Handling functions.

8.1 abort

The **abort** function executes an illegal instruction that terminates the process.

The syntax of the function is as follows:

```
#include stdlib

void abort (void);
```

8.2 assert

The **assert** macro puts diagnostics into programs.

The syntax of the **assert** macro is as follows:

```
#include assert

void assert (int expression);
```

Arguments

The argument *expression* is an expression that has an int value.

Additional Information

When the **assert** macro is executed, if *expression* is false (that is, evaluates to zero), the **assert** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, and the source line number—the latter are respectively the values of the preprocessing macros __FILE__ and __LINE__) on the standard error file in an implementation-defined format. Then, it calls the **abort** function.

The message written by the **assert** macro has the following form:

```
Assertion failed:  expression, file aaa, line nnn
```

If *expression* is true (that is, evaluates to nonzero) or if the signal SIGABRT is being ignored, the **assert** macro returns no value.

Compiling with the CC command qualifier /DEFINE=NDEBUG or with the preprocessor directive #**define** NDEBUG ahead of the #**include assert** statement causes the **assert** macro to have no effect.

The **assert** function is implemented as a macro, not as a real function. If #**undef** is used to remove the macro definition and obtain access to a real function, the behavior is undefined.

8.3 atexit

The **atexit** function registers a function that will be called without arguments at program termination.

The syntax of the **atexit** function is as follows:

```
#include stdlib

void atexit (void (*func) (void));
```

Arguments

The argument *func* is a pointer to the function to be registered.

Additional Information

The **atexit** function returns a value that is not equal to zero if the registration succeeds. Up to 32 functions can be registered. However, you should not register a function more than once.

8.4 exit, _exit

The **exit** and **_exit** functions terminate the process from which they are called.

The syntax descriptions of the functions are as follows:

```
#include stdlib

void exit  (int status);
void _exit (int status);
```

Arguments

The argument *status* corresponds with an errno value. The errno values are defined in the *errno* definition module.

Additional Information

The **exit** and **_exit** functions return the specified status to the parent process, if any. If the program is invoked by the DIGITAL Command Language, the status is interpreted by DCL and a message is displayed. The two functions are identical; **_exit** is retained for reasons of compatibility with previous versions of VAX C.

8.5 perror

The **perror** function writes a short error message to **stderr** describing the last error encountered during a call to the VAX C Run-Time Library from a C program.

The syntax of the function is as follows:

```
#include stdio

int perror (const char *str);
```

Arguments

The argument *str* typically contains the name of the program that incurred the error.

The **perror** function writes out its argument (a user-supplied prefix to the error message), followed by a colon, followed by the message itself, followed by a newline.

8.6 strerror

The **strerror** function maps the error number in *errnum* to an error message string.

The syntax of the function **strerror** is as follows:

```
#include string

char *strerror(int errnum);
```

Additional Information

The return value is a pointer to a buffer that contains the appropriate error message. This buffer should not be modified by user programs. Moreover, calls to the **strerror** function may overwrite this buffer with a new message.

If the argument *errnum* does not correspond to a known RTL error code, the **strerror** function returns the null pointer value NULL.

8.7 Signal-Handling Functions

Signals are raised by a variety of events, including any of the following:

- A user typing CTRL/C at a terminal (thus raising the signal SIGINT)
- Certain programming errors
- A **gsignal** call

Signals are given the mnemonics (as in SIGINT) found in the *signal* definition module. Normally, all signals cause the termination of the receiving process. However, the **signal** function allows you to ignore most of them or to interrupt to a specific location for handling.

The syntax for a signal handler is as follows:

```
handler (sigint, code, scp);

int       sigint, code;
struct sigcontext *scp;
```

The argument sigint is the designated signal number, and the argument, code, designates the type of signal if more than one exists. The argument scp is a pointer to the structure, sigcontext (defined in the *signal* definition module), which contains information used to restore the context of the process as it was before the signal occurred. Once a signal handler is installed, it remains in effect until the program calls **sigvec** again to handle it.

The handler specified by the argument sv is established as the handler to be called when the signal specified by sigint is raised.

Table 8–2 shows the signals defined in the signal definition module, ways to generate the signals on VMS, and the attributes of the signal, such as whether or not the signal can be ignored. Unless noted otherwise, each signal can be reset and it can be caught or ignored.

Table 8–2: VAX C Signals

Name	Description	Generated by
SIGHUP	Hang up	Data set hang up
SIGINT	Interrupt	VMS CTRL/C interrupt
SIGQUIT	Quit	CTRL/C if the action for SIGINT is SIG_DFL (default)
SIGILL[1]	Illegal instruction	Illegal instruction, reserved operand, or reserved address mode
SIGTRAP[1]	Trace trap	TBIT trace trap or breakpoint fault instruction
SIGIOT	IOT instruction	Not implemented
SIGEMT	EMT instruction	Compatibility mode trap or op code reserved to customer
SIGFPE	Floating-point exception	Floating-point overflow
SIGKILL[2]	Kill	External signal only
SIGBUS	Bus error	Access violation or change mode user

[1]Not reset when caught.

[2]Cannot be caught or ignored.

Table 8–2 (Cont.): VAX C Signals

Name	Description	Generated by
SIGSEGV	Segment violation	Length violation or change mode supervisor
SIGSYS	System Call error	Bad argument to system call
SIGPIPE	Broken pipe	Not implemented
SIGALRM	Alarm clock	Timer AST
SIGTERM	Software terminate	External signal only

The following sections describe the signal-handling functions that you can use to recover from programming errors without aborting your program.

8.7.1 alarm

The **alarm** function sends the signal SIGALRM (defined in the *signal* definition module) to the invoking process after the number of seconds indicated by its argument has elapsed.

The syntax of the function is as follows:

```
#include signal

int alarm (unsigned int seconds);
```

Arguments

The argument *seconds* has a maximum limit of 4,294,967,295 seconds. Calling **alarm** with a zero argument cancels any pending alarms.

Additional Information

The **alarm** function returns the number of seconds remaining from a previous alarm request.

Unless it is caught or ignored, the signal generated by **alarm** terminates the process. Successive **alarm** calls reinitialize the alarm clock. Alarms are not stacked.

Because the clock has a 1-second resolution, the signal may occur up to 1 second early. If the SIGALRM signal is caught, resumption of execution may be delayed by an arbitrary amount because of scheduling delays.

8.7.2 gsignal, raise

The **gsignal** and **raise** functions generate a specified software signal. Generating a signal causes the action established by the **ssignal** function to be taken.

The syntax of the functions is as follows:

```
#include signal
int gsignal (int sig, ...);
```

```
#include signal
int raise (int sig, ...);
```

Arguments

The arguments to the **gsignal** and **raise** functions are as follows:

sig Identifies the signal to be generated.

... Represents an optional signal type. For example, signal SIGFPE—the arithmetic trap signal—has 10 different codes, each representing a different type of arithmetic trap. Table 8–3 presents the various codes.

Table 8–3: Signal Types

Hardware Condition	Signal	Code
Arithmetic Traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT

Table 8–3 (Cont.): Signal Types

Hardware Condition	Signal	Code
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Reserved instruction	SIGILL	ILL_PRIVIN_FAULT
Reserved operand	SIGILL	ILL_RESOP_FAULT
Reserved addressing	SIGILL	ILL_RESAD_FAULT
Compatibility mode	SIGILL	Hardware supplied
Length access control	SIGSEGV	—
Chme	SIGSEGV	—
Chms	SIGSEGV	—
Chmu	SIGSEGV	—
Trace pending	SIGTRAP	—
Bpt instruction	SIGTRAP	—
Protection violation	SIGBUS	—
Customer-reserved code	SIGEMT	—

The signal codes can be represented by mnemonics or numbers. The arithmetic trap codes are represented by the numbers 1–10, whereas the SIGILL codes are represented by the numbers 0–2. The code values are defined in the *signal* definition module.

Additional Information

The result of a **gsignal** or **raise** call is one of the following:

- If **gsignal** or **raise** specifies a sig argument that is outside the range defined in the signal module, then the specified function returns zero, and the variable errno is set to EINVAL. See Table 8–1 for more information.
- If **ssignal** establishes SIG_DFL (default action) for the signal, then the functions do not return. The image is exited with the VMS error code that corresponds to the signal.
- If **ssignal** establishes SIG_IGN (ignore signal) as the action for the signal, then **gsignal** or **raise** returns its argument, sig.
- Otherwise, **ssignal** must have established an action function for the signal. That function is called, and that function's return value is returned by **gsignal** or **raise**.

The **gsignal** and **raise** functions are VAX C specific and are not portable.

8.7.3 kill

The **kill** function sends a signal to the process specified by a process ID.

The syntax of the function is as follows:

```
#include signal

int kill (int pid, int sig);
```

Additional Information

Unless you have system privileges, the sending and receiving processes must have the same UIC. The **kill** function returns zero if the **kill** was successfully queued. It returns –1 to indicate errors, including:

- The receiving process has a different UIC and the user is not a SYSTEM user.
- The receiving process does not exist.

If *pid* is the process id of the invoking process, then the **kill** function acts as though the **raise** function had been called.

If **kill** is successful, the receiving process is always terminated. The termination status of the receiving process is the VMS error code that corresponds to the value of the signal that was sent.

8.7.4 longjmp, setjmp

The **setjmp** and **longjmp** functions provide a way to transfer control from a nested series of function invocations back to a predefined point without returning normally; that is, not by a series of **return** statements. The **setjmp** function saves the context of the calling function in an environment buffer. The **longjmp** function restores the context of the environment buffer.

The syntax descriptions of the functions are as follows:

```
#include setjmp

int setjmp (jmp_buf env):
void longjmp (jmp_buf env, int val);
```

Arguments

The arguments to the **setjmp** and **longjmp** functions are as follows:

env — Represents the environment buffer and must be an array of integers long enough to hold the register context of the calling function. The type jmp_buf is defined by a **typedef** found in the *setjmp* definition module. The contents of the general-purpose registers, including the program counter (PC), are stored in the buffer.

value — Passed from **longjmp** to **setjmp**, and then becomes the second return value of the **setjmp** call. If *value* is passed as zero, it will be converted to 1.

Additional Information

When **setjmp** is first called, it returns the value zero. If **longjmp** is then called, naming the same environment as the call to **setjmp**, control is returned to the **setjmp** call as if it had returned normally a second time. The return value of **setjmp** in this second return is the value supplied by the user in the **longjmp** call. To preserve the true value of **setjmp**, the function calling **setjmp** must not be called again until the associated **longjmp** is called.

The **setjmp** and **longjmp** functions rely on the VMS condition-handling facility to effect a nonlocal goto with a signal handler. The **longjmp** function is implemented by generating a VAX C RTL specified signal and allowing VMS to unwind back to the desired destination. Thus, the VAX C RTL must be in control of signal handling for any VAX C image. In order for VAX C to be in control of signal handling, you *must* establish all exception handlers through a call to the **VAXC$ESTABLISH** function. See Section 8.7.14 for more information.

CAUTION

The **longjmp** function may be invoked from a signal handler that has been established for any signal supported by the VAX C RTL, subject to the following nesting restrictions:

1. The **longjmp** function will not work if invoked from nested signal handlers. The result of the **longjmp** function, when invoked from a signal handler that has been entered as a result of an exception generated in another signal handler, is undefined.
2. The **setjmp** function should *not* be invoked from a signal handler unless the associated **longjmp** is to be issued before the handling of that signal is completed.

8.7.5 pause

The **pause** function causes its calling process to stop (hibernate) until the process receives a signal.

The syntax of the function is as follows:

```
#include signal

int pause (void);
```

Additional Information

Control is not returned to the process that called **pause**, except after a SYS$WAKE system service call. The process may be reawakened by **kill** or **alarm**.

8.7.6 sigblock

The **sigblock** function causes the signals in *mask* to be added to the current set of signals being blocked from delivery.

The syntax of the function is as follows:

```
#include signal

int sigblock (int mask);
```

Arguments

Signal *i* is blocked if the *i*–1 bit in *mask* is a 1. For example, to add the protection-violation signal to the set of blocked signals, use the following:

```
sigblock(1 << (SIGBUS - 1));
```

You can express signals in mnemonics (such as SIGBUS for a protection violation) or numbers as defined in the *signal* definition module, and you can express combinations of signals using the bitwise OR operator (|).

Additional Information

The **sigblock** function returns the previous set of masked signals.

8.7.7 signal

The **signal** function allows you either to catch or to ignore a signal.

The syntax of,the function is as follows:

```
#include signal
void (*signal (int sig, void (*func) (int, ...))) (int, ...);
```

Arguments

The arguments to the **signal** function are as follows:

sig The number or mnemonic associated with a signal. Customarily, the sig argument is one of the mnemonics defined in the *signal* definition module.

func Either the action to be taken when the signal is raised, or the address of a function needed to handle the signal.

If *func* is the constant SIG_DFL, the action for the given signal is reset to the default action which is the termination of the receiving process. If the argument is SIG_IGN, the signal is ignored. Not all signals can be ignored.

If *func* is neither SIG_DFL nor SIG_IGN, it specifies the address of a Signal-Handling function. When the signal is raised, the addressed function is called with *sig* as its argument. When the addressed function returns, the interrupted process continues at the point of interruption. (This is called "catching a signal.") Except as indicated in Table 8–2, signals are reset to SIG_DFL after they have been caught.

Additional Information

You must call **signal** each time you want to catch a signal.

The **signal** function returns the address of the function previously (or initially) established to handle the signal. If the sig argument is out of range, **signal** returns –1 and sets the variable errno to EINVAL. See Table 8–1 for more information.

8.7.8 sigpause

The **sigpause** function assigns *mask* to the current set of masked signals and then waits for a signal.

The syntax of the function is as follows:

```
#include signal

void sigpause  (int mask);
```

Arguments

See **sigblock** in Section 8.7.6 for information concerning the argument *mask*.

Additional Information

When control returns to **sigpause**, the function restores the previous set of masked signals and then returns EINTR, for "interrupt." The value EINTR is defined in the *errno* definition module.

Usually, a signal is blocked using **sigblock** which examines variables modified on the occurrence of the signal, determining if there is further work to be done. The process pauses using **sigpause** with the mask returned by **sigblock** as its argument.

8.7.9 sigsetmask

The **sigsetmask** function establishes those signals which are blocked from delivery.

The syntax of the function is as follows:

```
#include signal

int sigsetmask (int mask);
```

Arguments

See **sigblock** in Section 8.7.6 for information concerning the argument *mask*.

You can express signals in mnemonics (such as SIGBUS for a protection violation) or numbers as defined in the *signal* definition module, and you can express combinations of signals using the bitwise OR operator (|). The **sigsetmask** function returns the previous set of masked signals.

8.7.10 sigstack

The **sigstack** function defines an alternate stack on which to process signals. This allows the processing of signals in a separate environment from that of the current process.

The syntax of the function is as follows:

```
#include signal

int sigstack   (struct sigstack *ss, struct sigstack *oss);
```

The structure sigstack is defined in the standard include module *signal* as follows:

```
struct sigstack
  {
     char     *ss_sp;
     int      ss_onstack;
  };
```

Arguments

The arguments to the **sigstack** function are as follows:

ss — If the argument ss is nonzero, it specifies the address of a structure that holds a pointer to a designated section of memory as a signal stack on which to deliver signals.

oss — If the argument oss is nonzero, it specifies the address of a structure which will be stored to the current state of the signal stack.

Additional Information

If the **sigvec** function specifies that the signal handler execute on the signal stack, the system checks to see if the process is executing currently on that stack. If the process is not executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution. If the argument oss is nonzero, the current state of the signal stack is returned.

Signal stacks must be allocated an adequate amount of storage; they do not "expand" like the run-time stack. If the stack overflows, an error occurs.

The structure sigstack is defined in the *signal* definition module.

Upon successful completion, the function returns 0. Otherwise, the function returns −1.

8.7.11 sigvec

The **sigvec** function assigns a handler for a specific signal.

The syntax of the function is as follows:

```
#include signal

int sigvec     (int sigint, struct sigvec *sv, struct sigvec *osv);
```

The structure sigvec is defined in the standard include module *signal* as follows:

```
struct sigvec
  {
     int   (*handler)();
     int   mask;
     int   onstack;
  };
```

Arguments

The arguments to the **sigvec** function are as follows:

sv If *sv* is nonzero, it specifies the address of a structure containing a pointer to a handler routine and mask to be used when delivering the specified signal and a flag indicating whether the signal is to be delivered to an alternative stack. If the argument sv.onstack has a value of 1, the system delivers the signal to the process on a signal stack specified with **sigstack**.

osv If *osv* is nonzero, the previous handling information for the signal is returned to the user.

Additional Information

The **sigvec** function returns 0 if the call succeeded and returns –1 if an error occurred. Upon error, the variable errno contains the value explaining the error. See Table 8–1 for more information.

8.7.12 sleep

The **sleep** function suspends the execution of the current process for at least the number of seconds indicated by its argument.

The syntax of the function is as follows:

```
#include signal

int sleep (unsigned seconds);
```

Additional Information

On success, **sleep** returns the number of seconds that the process slept. On error, **sleep** returns –1.

8.7.13 ssignal

The **ssignal** function allows you to specify the action to be taken when a particular signal is raised.

The syntax of the function is as follows:

```
#include signal

void (*ssignal (int sig, void (*func) (int, ...))) (int, ...);
```

Arguments

The arguments to the **ssignal** function are as follows:

sig — A number or mnemonic associated with a signal. The symbolic constants for signal values are defined in the *signal* definition module (see Table 8–2).

func — Represents the action to be taken when the signal is raised, or the address of a function that is executed when the signal is raised.

Additional Information

The **ssignal** function returns the address of the function previously established as the action for the signal. Note that the address may contain the value SIG_DFL (0) or SIG_IGN (1).

The **ssignal** function calls **signal** with the same arguments; the only difference between the two is in their return value on detecting an error (usually an invalid signal argument). The function **ssignal** returns zero to indicate errors. For this reason, there is no way to know whether a return status of zero indicates failure or whether it indicates that a previous action was SIG_DFL (0). The **signal** function returns -1 on error.

The **ssignal** function is VAX C specific and is not portable.

See also **sigvec** in this section.

8.7.14 VAXC$ESTABLISH

If you want to establish a VMS exception handler, it *must* be done through a call to the VAX C RTL function **VAXC$ESTABLISH**. This function establishes a special VAX C RTL exception handler that catches all RTL related exceptions and passes on all others to your handler.

The syntax of the function is as follows:

```
#include signal

void VAXC$ESTABLISH (int (*exception_handler)(void *mecharr, void *sigarr));
```

Arguments

The argument *exception_handler* is the name of the function that is to be established as a VMS condition handler. You pass the address of a function as an argument to **VAXC$ESTABLISH**.

Additional Information

The **VAXC$ESTABLISH** function can *only* be invoked from a VAX C function, as it relies on the allocation of data space on the run-time stack by the VAX C compiler. Calling the VMS system library routine LIB$ESTABLISH directly from a VAX C function will result in undefined results by the **setjmp** and **longjmp** functions.

8.8 Program Examples

Example 8–1 illustrates the functionality of **signal**, **alarm**, and **pause**.

Example 8–1: Suspending and Resuming Programs

```
/*  This program shows how to alternately suspend and resume   *
 *  a program using the signal, alarm, and pause functions.     */

#define SECONDS 5

#include stdio
#include signal

int number_of_alarms = 5;           /*  Set alarm counter       */

main()
{
   int alarm_action();
                                    /*  Pass signal and         *
                                     *   function to SIGNAL     */
   signal(SIGALRM, alarm_action);

                                    /*  Set alarm clock for 5   *
                                     *   seconds                */
   alarm(SECONDS);

                                    /*  Suspend the process     *
                                     *   until the signal is    *
                                     *   received               */
   pause();
}

alarm_action()
{
                                    /*  Print the value of      *
                                     *   alarm counter          */
   printf("\t<%d\007>", number_of_alarms);

                                    /*  Pass signal and the     *
                                     *   function to SIGNAL     */
   signal(SIGALRM, alarm_action);

   alarm(SECONDS);                  /*  Set alarm clock         */

   if (--number_of_alarms)          /* Decrement alarm counter */
      pause();
}
```

Sample output from the previous example is as follows:

```
$ RUN  EXAMPLE [RETURN]
         <5> <4>   <3>    <2>   <1>

%SYSTEM-W-ASTFLT, AST fault, SP=FFFFFFFE, param=00001665, PC=03C00000,
                       PSL=7FF2C10C, target PC=00000000, PSL=00000000

%TRACE-W-TRACEBACK, symbolic stack dump follows
module name      routine name              line        rel PC    abs PC
C$SIGNAL         gsignal                   1728      000000C2  00001665
                                                     00001307  00001307
C$SETJMP         LONGJMP                             8000254D  80009E5E
                                                     00001699  00001699
TEMP             main                       146      0000002A  0000122A
```

Chapter 9

Memory Allocation Functions

All of the VAX C Run-Time Library functions that require additional storage from the heap get that storage using the VAX C memory allocation functions **malloc**, **calloc**, **realloc**, **free**, and **cfree**. These functions use the LIB$GET_VM and LIB$FREE_VM routines to acquire the additional virtual memory. The routines LIB$GET_VM and LIB$FREE_VM take a fair amount of time to supply the virtual memory and, thus, the VAX C Run-Time Library attempts to reduce the number of calls to these functions, in the following manner.

The VAX C Run-Time Library maintains a pointer to the memory block that was most recently freed by either **free** or **cfree**. The last freed block is not returned to VMS by LIB$FREE_VM. Instead, the VAX C Run-Time Library attempts to satisfy the next request with this saved block.

If the saved block is large enough to satisfy the request, it is used. Any unused portion of this block is retained for future allocation requests, provided that it is larger than the predefined minimum size. The size constraint prevents over-fragmentation of memory. If the saved block is too small to satisfy a request, it is retained and the requested memory is allocated by LIB$GET_VM.

The freeing of a second block causes the saved block, if any, to be returned to VMS through LIB$FREE_VM. The new block is then saved to be used, if possible, for the next request.

Since the VAX C Run-Time Library saves the last freed block of storage, there is not a one-to-one correspondence between calls to **malloc** or **calloc** and LIB$GET_VM, or between calls to **free** or **cfree** and LIB$FREE_VM. VAX C RTL functions use LIB$GET_VM and LIB$FREE_VM to acquire and return dynamic memory. However, the address given to the VAX C RTL routines by LIB$GET_VM is not the same as the address given to the user by the VAX C RTL routines. Therefore, any memory allocated by a VAX C RTL routine must be deallocated by a VAX C RTL routine. Similarly, any memory allocated by LIB$GET_VM must be deallocated by LIB$FREE_VM.

The **brk** and **sbrk** functions assume memory can be allocated contiguously from the top of the user's address space. However, the **malloc** function and RMS may allocate space from this same address space. Therefore, it is not recommended that you use the **brk** and **sbrk** functions in conjunction with RMS and VAX C Run-Time Library routines that use **malloc**.

The following sections describe the memory allocation functions.

9.1 brk, sbrk

The **brk** and **sbrk** functions determine the lowest virtual address that is not used with the program.

The syntax descriptions of the functions are as follows:

```
#include stdlib

void *brk  (unsigned long int addr);
void *sbrk (unsigned long int incr);
```

Arguments

The arguments to the **brk** and **sbrk** functions are as follows:

addr Specifies the lowest address to the **brk** function, which the function rounds up to the next 512-byte multiple. This rounded address is called the *break* address.

incr Specifies, to the **sbrk** function, the number of bytes to add to the current break address.

Additional Information

The **brk** function returns the break address (the address of an object of type **char**). An address that is greater than or equal to the break address and less than the stack pointer is considered to be outside the program's address space. Attempts to reference it will cause access violations.

The **sbrk** function adds the number of bytes specified by its argument to the current break address and returns the old break address.

When a program is executed, the break address is set to the highest location defined by the program and data storage areas. Consequently, **brk** and **sbrk** are needed only by programs that have growing data areas.

The **brk** and **sbrk** functions return –1 if the program requests too much memory.

9.2 calloc, malloc (Memory Allocation)

The **calloc** and **malloc** functions allocate an area of memory.

The syntax descriptions of the functions are as follows:

```
#include stdlib

void *calloc (size_t number, size_t size);
void *malloc (size_t size);
```

Arguments

The arguments to the **calloc** and **malloc** functions are as follows:

number Specifies the number of items to be allocated.

size The size of each item.

Additional Information

The **calloc** function initializes the items to zero. If unable to allocate the space, **calloc** returns zero.

The **malloc** function allocates a contiguous area of memory whose size in bytes is supplied as an argument. It returns zero if it is unable to allocate enough memory.

Both functions return the address of the first byte, which is aligned on an octaword boundary.

9.3 cfree, free (Memory Deallocation)

The **free** and **cfree** functions make available for reallocation the area allocated by a previous **calloc**, **malloc**, or **realloc** call.

The syntax of the functions is as follows:

```
#include stdlib

int cfree (void *ptr);
int free (void *ptr);
```

Arguments

The argument *ptr* is the address returned by a previous call to **malloc**, **calloc**, or **realloc**.

Additional Information

The contents of the deallocated area are unchanged. The functions return zero if the area is successfully freed, –1 if an error occurs.

In VAX C, **free** and **cfree** are the same function. However, for compatibility with other C implementations, you should use **free** with **malloc** or **realloc**, and **cfree** with **calloc**.

9.4 realloc (Memory Reallocation)

The **realloc** function changes the size of the area pointed to by the first argument to the number of bytes given by the second argument.

The syntax of the function is as follows:

```
#include stdlib

void *realloc (void *ptr, size_t size);
```

Arguments

The arguments to the **realloc** function are as follows:

ptr	May point to an allocated area or, unless other allocations have been made, to the area most recently freed by **free** or **cfree**.
size	Specifies the new size of the allocated area.

Additional Information

If *ptr* is the null pointer constant (NULL), the behavior of the **realloc** function is equivalent to that of the **malloc** function.

The **realloc** function returns the address of the area, since the area may have to be moved to a new address in order to reallocate enough space. If the area was moved, the space previously occupied is freed. If **realloc** is unable to reallocate the space (for example, if there is not enough room), it returns zero.

The contents of the area are unchanged up to the lesser of the old and new sizes. New space in the reallocated area is initialized with zero.

9.5 Program Example

Example 9–1 illustrates the use of the **malloc, calloc, free, and cfree** functions.

Example 9–1: Allocating and Deallocating Memory for Structures

```
/*  This example takes lines of input from the terminal until  *
 *  it encounters a CTRL/Z. It places the strings into an      *
 *  allocated buffer, copies the strings to memory allocated   *
 *  for structures, prints the lines back to the screen, and   *
 *  then deallocates all memory used for the structures.       */

#include  stdio
#define   MAX_LINE_LENGTH  80

struct line_rec                   /*  Declare the structure    */
   {
      struct line_rec *next;      /*  Pointer to next line     */
      char    *data;              /*  A line from terminal     */
   };

main ()
{
   char    *buffer;
                                  /* Define pointers to        *
                                   *  structure (input lines) */
   struct line_rec  *first_line, *next_line, *last_line = NULL;

                                  /* buffer points to memory   */
   buffer = malloc(MAX_LINE_LENGTH);

   if (buffer == 0)               /* If error ...              */
      {
         perror("malloc");
         exit();
      }

   while (gets(buffer) != NULL)   /* While not CTRL/Z ...      */
      {
                                  /* Allocate for input line   */
         next_line = calloc(1, sizeof (struct line_rec));

         if (next_line == NULL)
            {
               perror("calloc");
               exit();
            }
```

(Continued on next page)

Example 9–1 (Cont.): Allocating and Deallocating Memory for Structures

```
                                   /* Put line in data area    */
         next_line-> data = buffer;

         if (last_line == NULL)    /* Reset pointers           */
            first_line = next_line;
         else
            last_line-> next = next_line;

         last_line = next_line;
                                  /* Allocate space for the   *
                                   *  next input line          */
         buffer = malloc(MAX_LINE_LENGTH);

         if (buffer == 0)
            {
               perror("malloc");
               exit();
            }
      }

   free(buffer);                  /* Last buffer always unused */
   next_line = first_line;        /* Pointer to beginning      */

   do
      {
         puts(next_line -> data); /* Write line to screen      */
         free(next_line -> data); /* Deallocate a line         */
         last_line = next_line;
         next_line = next_line-> next;
         cfree(last_line);
      }
   while (next_line != NULL);
}
```

Sample input and output from the previous example is as follows:

```
$ RUN  EXAMPLE [RETURN]
line one
line two
[CTRL/Z]
EXIT
line one
line two
$
```

Chapter 10

Subprocess Functions

The VAX C Run-Time Library provides functions that allow the programmer to create subprocesses from a VAX C program. The creating process is called the "parent" and the created subprocess is called the "child."

The creation of a child process is done within the parent process with the exec functions (**execl**, **execle**, **execv**, **execve**, **execlp**, and **execvp**) and the **vfork** function. Other functions are available to allow the parent and child to read and write data across processes (**pipe**) and to allow for synchronization of the two processes (**wait**). This chapter describes the implementation and use of these functions.

The parent process can execute VAX C programs in its children, either synchronously or asynchronously. The number of children that can run simultaneously is determined by the /PRCLM user authorization quota that has been established for each user on your system. Other quotas that may affect the use of subprocesses are /ENQLM (Queue Entry Limit), /ASTLM (AST Waits Limit), and /FILLM (Open File Limit).

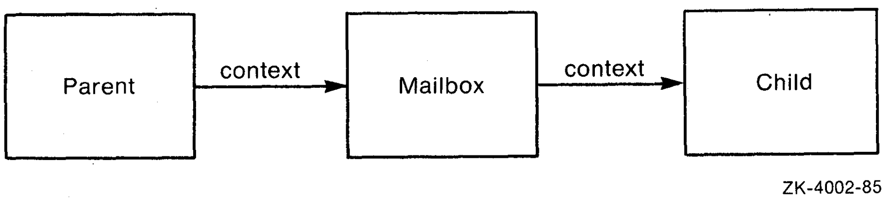
10.1 The Implementation of Child Processes in VAX C

Child processes are created by VAX C functions with the VMS LIB$SPAWN Run-Time Library routine. (See the *VAX/VMS Run-Time Library Routines Reference Manual* for information on LIB$SPAWN.) The use of LIB$SPAWN allows you to create multiple levels of child processes; that is, the parent's children can also spawn children, and so on, up to the limits allowed by the user authorization quotas previously noted.

Child processes are restricted in that they can execute only other VAX C programs. Other native-mode VMS languages do not share VAX C's ability to communicate between processes, or, if they do, they do not use the same mechanisms. In addition, the parent process must be run under a DIGITAL-supported command language interpreter (CLI), such as the DIGITAL Command Language (DCL) or the DEC/Shell. The parent may not be run as a detached process or under control of a user-supplied CLI.

Parent and child processes communicate through a mailbox as shown in Figure 10–1. This mailbox transfers the context in which the child will run. The context mailbox, as it is called, passes to the child the information it inherits from the parent, such as the names and file descriptors of all the files that have been opened by the parent and the current location within those files. The mailbox is deleted by the parent when the child image exits.

Figure 10–1: Communications Links Between Parent and Child Processes



NOTE

> The mailbox created by the **vfork** and exec functions is temporary. The logical name of this mailbox is VAXC$EXECMBX and is reserved for the use of the VAX C Run-Time Library (RTL).

The mailbox is created with a maximum message size and a buffer quota of 512 bytes each. You need the TMPMBX privilege to create a mailbox with these VAX C RTL functions. Since TMPMBX is the privilege required by the PRINT and SUBMIT DCL commands, most users on a system have this privilege. If you are not sure, type SHOW PROCESS/PRIVILEGES; it will show which system privileges you have.

You cannot change the characteristics of these mailboxes. For more information on mailboxes, see the *VAX/VMS I/O Reference Volume*.

VMS does not permit two processes to use the same physical terminal for input, and the VAX C Run-Time Library does not support file sharing or the default C stream file type. Therefore, if stdir is connected to a terminal or if stdout or stderr is connected to stream files, these standard streams will be redirected to the NUL device _NLA0:.

10.1.1 system

The **system** function passes a given string to the host environment to be executed by a command processor.

The syntax of the **system** function is as follows:

```
#include processes

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

Arguments

The argument *string* is a pointer to the string to be executed.

Additional Information

If the argument is a NUL pointer, the **system** function returns a nonzero value to indicate that the **system** function is supported. The **system** function spawns a subprocess and executes the command specified by *string* in that subprocess. The **system** function will wait for the subprocess to complete before returning the subprocess status as the return value of the function.

10.1.2 vfork

The **vfork** function creates an independent child process.

The syntax of the function is as follows:

```
#include processes

int vfork (void);
```

Additional Information

The **vfork** function provided by VAX C differs from the **fork** function provided by other C implementations. The two major differences are as follows:

The vfork Function	The fork Function
Used with the exec functions.	Can be used without exec for asynchronous processing.
Creates an independent child process that shares some of the parent's characteristics.	Creates an exact duplicate of the parent process that branches at the point where **vfork** is called, as if the parent and the child are the same process at different stages of execution.

The **vfork** function provides the setup necessary for a subsequent call to an exec function. Although no process is actually created by **vfork**, it performs the following steps:

- It saves the return address (the address of the **vfork** call) to be used later as the return address for the call to an exec function.
- It duplicates the parent's stack frame.
- It returns the integer 0 the first time it is called; that is, before the call to an exec function has been made. After the corresponding exec function call has been made, the exec function returns control to the parent process, at the point of the **vfork** call, and it returns the process id of the child as the return value. Thus, unless the exec function fails, control will seem to return twice from **vfork** even though one call was made to **vfork** and one call was made to the exec function.

The behavior of the **vfork** function is similar to the behavior of the **setjmp** function. Both **vfork** and **setjmp** establish a return address for later use, both return the integer 0 when they are first called to set up this address, and both pass back the second return value as though it were returned by them rather than by their corresponding exec or **longjmp** function calls.

10.2 The exec Functions

There are six exec functions that can be called to execute a VAX C image in the child process. These functions expect that **vfork** has been called to set up a return address. However, the exec functions call **vfork** if the parent process did not.

When **vfork** is called by the parent, exec returns to the parent process. When **vfork** has been called by an exec function, the exec returns to itself, waits for the child to exit, and then exits the parent process. Thus, exec does not return to the parent process unless the parent calls **vfork** to save the return address.

Unlike UNIX based systems, the exec functions in the VAX C Run-Time Library cannot determine if the specified program image exists. Therefore, the exec functions will appear to succeed even though the image does not exist. The status of the child process, returned to the parent process, will indicate that this error occurred. You can retrieve this error code by using the **wait** function.

10.2.1 execl, execle, execlp, execv, execve, execvp

The exec functions pass the name of an image to be activated in a child process.

The syntax descriptions of the functions are as follows:

```
#include processes

int execl(char *file_spec, char *argn,...);

int execle(char *file_spec, char *argn,...,
           char *envp[]);

int execlp(char *file_name, char *argn,...);

int execv(char *file_spec, char *argv[]);

int execve(char *file_spec, char *argv[],
           char *envp[]);

int execvp(char *file_name, char *argv[]);
```

Arguments

The arguments to the exec functions are as follows:

file_spec The file specification (full) of a new image to be activated in the child process.

file_name The file name of a new image to be activated in the child process. The device and directory specification for the file is obtained by a search of the environment name VAXC$PATH.

argn Represents a sequence of pointers to null-terminated character strings. By convention, at least one argument must be present and must point to a string that is the same as the new process file name (or its last component).

envp An array of strings that specifies the program's environment. Each string in argument *envp* has the form:

```
name = value
```

The name can be one of the names listed in the following table and the value is a NUL-terminated string to be associated with the name.

- HOME—The user's login directory
- TERM—The type of terminal being used
- PATH—The default device and directory
- USER—The name of the user who initiated the process

The last element in envp must be the null pointer NULL.

When the operating system executes the program, it places a copy of the current environment vector (envp) in the external variable environ.

argv An array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, *argv[0]* must point to a string that is the same as the new process file name (or its last component). *Argv* is terminated by a null pointer.

. . . Represents a sequence of pointers to strings. At least one pointer must exist to terminate the list. This pointer may be the NULL pointer.

Additional Information

In order to understand how the exec functions operate, consider how VMS calls any VAX C program as shown in the following syntax:

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

The identifier *argc* is the argument count; *argv* is an array of argument strings. The first member of the array (argv[0]) always contains the name of the image. The actual arguments are placed in subsequent elements of the array. The last element of the array is always the null pointer.

An exec function calls a child process in the same way that the Run-Time system calls any other VAX C program. The exec functions pass the name of the image to be activated in the child; this value is placed in argv[0]. However, the functions differ in the way they pass arguments and environment information to the child:

- Arguments can be passed in separate character strings (**execl**, **execle**, and **execlp**) or in an array of character strings (**execv** and **execve**).
- The environment can be explicitly passed in an array (**execle**, **execve**) or taken from the parent's environment variable (**execl** and **execv**).

10.2.1.1 Exec Processing

The exec functions use the LIB$SPAWN routine to create the subprocess and activate the child image within the subprocess. This child process inherits the parent's environment, including all defined logical names and command line interpreter symbols. The exec functions use the logical name VAXC$EXECMBX to communicate between parent and child; this logical name must not exist outside the context of the parent image.

The exec functions pass the following information to the child:

- The parent's **umask** value, which specifies whether any access is to be disallowed when a new file is created. For more information concerning the **umask** function, refer to Chapter 11, System Functions.
- The file name string associated with each file descriptor and the current position within each file. The child opens the file and calls **lseek** to position the file to the same location as the parent. Note that if the file is a record file, the child is positioned on a record boundary, regardless of the parent's position within the record. For more information concerning file descriptors and the **lseek** function, refer to Chapter 2, Standard I/O Functions and Macros.

This information is sent to the child for all descriptors known to the parent including all descriptors for open files, null descriptors, and duplicate descriptors.

File pointers are not transferred to the child. For files opened by a file pointer in the parent, only their corresponding file descriptors are passed to the child. Therefore, the **fdopen** function must be called to associate a file pointer with the file descriptor if the child will access the file by file pointer. For more information concerning the **fdopen** function, refer to Chapter 2, Standard I/O Functions and Macros.

Process permanent input files are not inherited by the child process. Rather, they are replaced with the null device NLA0. See Section 10.1 for restrictions on the use of the parent's process permanent files by the child process.

- The signal data base. Only SIG_IGN (ignore) actions are inherited. Actions specified as routines are changed to SIG_DFL (default) because the parent's signal-handling routines are inaccessible to the child.
- The environment and argument vectors.

When everything has been transmitted to the child, exec processing is complete. Control in the parent process then returns to the address saved by **vfork** and the child's process id is returned to the parent.

10.2.1.2 Exec Error Conditions

The exec functions can only fail if LIB$SPAWN is unable to create the subprocess. Conditions that can cause a failure include exceeding the subprocess quota or finding the communications by the context mailbox between the parent and child to be broken. Exceeding some quotas will not cause LIB$SPAWN to fail, but rather to be put into a wait state that can cause the parent process to "hang." An example of such a quota is the Open File Limit quota.

You will need an Open File Limit quota of at least 20 files, with an average of three times the number of concurrent processes that your program will run. If you use more than one open pipe at a time, or perform I/O on several files at one time, this quota may need to be even higher. See your system manager if this quota needs to be increased.

When an exec fails, a value of −1 is returned. After such a failure, the parent is expected to call either the **exit** or **_exit** function. Both functions then return to the parent's **vfork** call, returning the child's process id. In this case, the child process id returned by exec is less than zero. For more information concerning the **exit** function, refer to Chapter 8, Error-Handling Functions.

10.3 Synchronizing Processes

A child process is terminated when the parent process terminates. Therefore, the parent process must check the status of its child processes before exiting. This is done with the VAX C RTL function **wait**.

10.3.1 wait

The syntax of the function is as follows:

```
#include processes

int wait (int *status);
```

Arguments

The argument *status* is the address of a location to receive the final status of the terminated child. Thus, the child can set the status with the **exit** function and the parent can retrieve this value by specifying *status*.

Additional Information

The **wait** function suspends the parent process until a value is returned from the child. This value is the final status of the child.

The return value from **wait** is the process id of the terminated child. If more than one child process was created, **wait** will return the process id of the terminated child that was most recently created. Subsequent calls to **wait** will return the process id of the next most recently created, but terminated, child.

10.4 Reading and Writing Data

You must use a mailbox for reading and writing data between the parent and child. The channels through which the processes communicate are called a *pipe*. You use the **pipe** function to create a temporary mailbox.

10.4.1 pipe

The syntax of the function is as follows:

```
#include processes

int pipe (int array_fdscptr[2], ...);
```

Arguments

The arguments to the **pipe** function are as follows:

array_fdscptr — An array of file descriptors. A pipe is implemented as an array of file descriptors associated with a mailbox. The file descriptors are allocated as follows:

- The first available file descriptor is assigned to writing, and the next available file descriptor is assigned to reading.
- The file descriptors are then placed in the array in reverse order; element 0 contains the file descriptor for reading, and element 1 contains the file descriptor for writing, as shown in Figure 10–2.

... — Represents two optional addtional arguments as follows:

flags

An optional argument and is identical to the same argument in the function **open**. The values for the argument are defined in the file definition module and have the following meanings:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_NDELAY	Ignored; not supported by VAX C.
O_APPEND	Append on each write.
O_CREAT	Create a file if it does not exist.
O_TRUNC	Create a new version of this file.
O_EXECL	Error if attempting to create existing file.

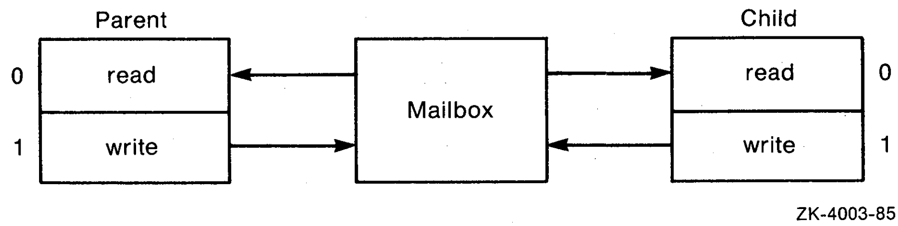
These flags are set using the bitwise OR operator (|) to separate specified flags. Opening a file with O_APPEND causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, **open** creates a new file by incrementing the version number by one, leaving the old version in existence. If O_CREAT is set and the named file does not exist, the VAX C RTL creates it with any attributes specified in the fourth and subsequent arguments, file_attribute. If O_EXECL is set with O_CREAT, then if the file already exists, the attempted open returns an error.

O_CREAT, O_EXECL, and O_TRUNC should not be used with pipes. O_APPEND is ignored with pipes.

bufsize

Optional and specifies the size of the mailbox in bytes. If you do not specify this argument, VAX C creates a mailbox with a default size of 512 bytes.

Figure 10–2: Implementation of a Pipe



Additional Information

The mailbox that is used for the pipe is a temporary mailbox. The mailbox is not deleted until all processes that have open channels to that mailbox close those channels. Each process that closes a previously active channel to the mailbox writes a message to the mailbox, indicating end-of-file.

The mailbox is created with the $CREMBX system service, specifying the following characteristics:

- Maximum message length of 512 characters
- Buffer quota of 512 characters
- A protection mask granting all privileges to USER and GROUP and no privileges to SYSTEM or WORLD

The buffer quota of 512 characters implies that no more than 512 characters can be written to the mailbox before all or part of the mailbox is read. Since a mailbox record is slightly larger than the data part of the message that it contains, not all of the 512 characters can be used for message data. The size of the buffer can be increased by specifying an alternative size using the optional third argument to the **pipe** function. A mailbox under VMS is a record oriented file with no carriage control attributes. It is fully buffered by default in the VAX C Run-Time Library.

The pipe is created by the parent process before **vfork** and exec are called. By calling **pipe** first, the child inherits the open file descriptors for the pipe. The **getname** function can then be used to return the name of the mailbox associated with the pipe, if this information is desired. The mailbox name returned by **getname** always has the format _MBA*nnnn*:, where *nnnn* is a unique number.

Both the parent and the child need to know in advance which file descriptors will be allocated for the pipe. This information cannot be retrieved at run time. Therefore, it is important to understand how file descriptors are used in any VAX C program. For more information concerning file descriptors, refer to Chapter 4, UNIX System I/O Functions.

File descriptors 0, 1, and 2 are always open in a VAX C program for stdin (SYS$INPUT), stdout (SYS$OUTPUT), and stderr (SYS$ERROR), respectively. Therefore, if no other files have been opened when **pipe** is called, **pipe** assigns file descriptor 3 for writing and file descriptor 4 for reading. In the array returned by **pipe**, 4 is placed in element 0 and 3 is placed in element 1.

If other files have been opened, **pipe** assigns the first available file descriptor for writing and the next available file descriptor for reading. Note that in this case, the pipe does not necessarily use adjacent file descriptors. For example, assume two files have been opened and assigned to file descriptors 3 and 4 and the first file is then closed. If **pipe** is called at this point, file descriptor 3 will be assigned for writing and file descriptor 5 will be assigned for reading. Element 0 of the array will contain 5 and element 1 will contain 3.

In large applications that do large amounts of I/O, it gets increasingly more difficult to predict which file descriptors are going to be assigned to a pipe; and, unless the child knows which file descriptors are being used, it will not be able to read and write successfully from and to the pipe.

One way to be sure that the correct file descriptors are being used is to use the following procedure:

1. Choose two descriptor numbers that will be known in both the parent and the child. The numbers should be high enough to account for any I/O that may be done before the pipe is created.
2. Call **pipe** in the parent at some point before calling exec.
3. In the parent, use **dup2** to assign the file descriptors returned by **pipe** to the file descriptors you chose. This now reserves those file descriptors for the pipe; any subsequent I/O will not interfere with the pipe.

Reading and writing through the pipe can be done with the UNIX I/O functions **read** and **write**, specifying the appropriate file descriptors. As an alternative, you can issue **fdopen** calls to associate file pointers with these file descriptors so that you can use the standard I/O functions (**fread** and **fwrite**).

NOTE

> If you use the UNIX I/O function **write** to write to a mailbox, and the third argument specifies a length of zero, then an end-of-file message is written to the mailbox.

Although two separate file descriptors are used for reading from and writing to the pipe, only one mailbox channel is used. Thus, some I/O synchronization is required. For example, assume that the parent writes a message to the pipe. If the parent is the first process to read from the pipe, then it will read its own message back. In the final example in Section 10.5, the required synchronization is achieved by means of a **wait** function call, whereby the parent waits until the child terminates before reading from the pipe. This form of synchronization is limited in its scope, and other alternative methods should be investigated.

10.5 Program Examples

Example 10–1 shows the basic procedures for executing an image in a child process. Since the first program is crucial to understanding the implementation of subprocesses in VAX C, important lines of source code are explained in the list that follows the example.

The child process in this first example prints a message 10 times.

Example 10–1: Creating the Child Process

```
/*  This example creates the child process.  The only     *
 *  functionality given to the child is the ability to    *
 *  print a message 10 times.                              *
 *                                                         *
 *    PARENT:                                              */

#include climsgdef           /* CLI status values          */
#include stdio
#include perror
#include processes

main()
{
   int status, cstatus;

❶  if ((status = vfork()) != 0)
      {
         ❸        if (status < 0)
         printf("Parent - Child process failed\n");
      else
         {
            printf("Parent - Waiting for Child\n");

         ❹              if ((status = wait(&cstatus)) == -1)
               perror("Parent - Wait failed");
            else

         ❺                 if (cstatus == CLI$_IMAGEFNF)
                  printf("Parent - Child does not \
exist\n");

            else
                  printf("Parent - Child final \
status: %d\n", cstatus);

         }
      }
         ❷     else
      {
         printf("Parent - Starting Child\n");
         if ((status = execl("child", 0)) == -1)
            {
               perror("Parent - Execl failed");
               _exit();
            }
      }
}
```

(Continued on next page)

Example 10–1 (Cont.): Creating the Child Process

```
/*  This is a program separate from the parent process.     *
 *                                                           *
 *  CHILD:                                                   *
 *                                                           */

main()
{
   int i;

   for (i=0; i < 10; i++)
      printf("Child - executing\n");
}
```

The following numbers correspond to the numbers in the previous example:

❶ The **vfork** function is called to set up the return address for the exec call.

Typically, **vfork** is used in the expression of an **if** statement. This construct allows you to take advantage of the double return aspect of **vfork**, since one return value is zero and the other nonzero.

❷ A zero return value is returned the first time **vfork** is called and the parent executes the **else** clause associated with the **vfork** call, which calls **execl**.

❸ A negative child process id is returned when an exec function fails. Therefore, the return value is checked for these conditions.

❹ The **wait** function is used to synchronize the parent and child processes.

❺ Since the exec functions can indicate success up to this point even if the image to be activated in child does not exist, the parent checks the child's return status for the predefined status, CLI$_IMAGEFNF (file not found).

In Example 10–2, the parent passes arguments to the child process.

Example 10–2: Passing Arguments to the Child Process

```
/*  In this example, the arguments are placed in an array,     *
 *  gargv, but they could also be passed to the child          *
 *  explicitly as a zero-terminated series of character        *
 *  strings. The child program in this example simply writes   *
 *  to stdout the arguments that have been passed to it.       *
 *                                                             *
 *     PARENT:                                                 *
 *                                                             */

#include climsgdef
#include stdio
#include perror
#include processes

main()
{
   int status, cstatus;
   char *gargv[] = { "Child", "ARGC1", "ARGC2", "Parent", 0 };

   if ((status = vfork()) != 0)
      {
         if (status < -1)
            printf("Parent - Child process failed\n");
         else
            {
               printf("Parent - waiting for Child\n");
               if ((status = wait(&cstatus)) == -1)
                  perror("Parent - Wait failed");
               else
                  if (cstatus == CLI$_IMAGEFNF)
                     printf("Parent - Child does not exist\n");
                  else
                     printf("Parent - Child final status: %x\n",
                             cstatus);
            }
      }
   else
      {
         printf("Parent - Starting Child\n");
         if ((status = execv("Child", gargv)) == -1)
            {
               perror("Parent - Exec failed");
               _exit();
            }
      }
}
```

(Continued on next page)

Example 10–2 (Cont.): Passing Arguments to the Child Process

```
/*  This is a program separate from the parent process.        *
 *                                                              *
 *  CHILD:                                                      *
 *                                                              */

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

   printf("Program name: %s\n", argv[0]);

   for (i = 1; i < argc; i++)
      printf("Argument %d: %s\n", i, argv[i]);
}
```

Example 10–3 shows how the **wait** function can be used to check the final status of multiple children being run simultaneously.

Example 10–3: Checking the Status of Child Processes

```
/*  In this example, the wait function is placed in a separate *
 *  for loop so that it is called once for each child. If      *
 *  wait were called within the first for loop, the parent     *
 *  would wait for one child to terminate before executing the *
 *  next child.  If there were only one wait request, any      *
 *  child still running when the parent exits would terminate  *
 *  prematurely.                                               *
 *                                                             *
 *     PARENT:                                                 *
 *                                                             */

#include climsgdef
#include stdio
#include perror
#include processes

main()
{
   int status, cstatus, mode, i;

   for (i = 0; i < 5; i++)
      {
         if ((status = vfork()) == 0)
            {
               printf("Parent - Starting Child %d\n", i);
               if ((status = execl("child", 0)) == -1)
                  {
                     perror("Parent - Exec failed");
                     _exit();
                  }
            }
         else
            if (status < -1)
               printf("Parent - Child process failed\n");
      }

   printf("Parent - Waiting for children\n");

   for (i = 0; i < 5; i++)
      {
         if ((status = wait(&cstatus)) == -1)
            perror("Parent - Wait failed");
         else
            if (cstatus == CLI$_IMAGEFNF)
               printf("Parent - Child does not exist\n");
            else
               printf("Parent - Child %X final status: %d\n",
                       status, cstatus);
      }
}
```

(Continued on next page)

Example 10–3 (Cont.): Checking the Status of Child Processes

```
/*  This is a program separate from the parent process.     *
 *                                                          *
 *  CHILD:                                                  *
 *                                                          */

main()
{
   int pid, i;

   printf("Child %0X: working...\n", (pid = getpid()));
   sleep(5);
   printf("Child %0X: Finished\n",pid);
}
```

Example 10–4 shows the use of **pipe** and **dup2** to communicate between a parent and child process through specific file descriptors. The **#define** preprocessor directive defines the preprocessor constants inpipe and outpipe as the names of file descriptors 11 and 12.

Since there is only one child being executed from the parent, the status value of the exec call is tested in a **switch** statement. Case 0 is executed the first time **vfork** is called. Case –1 is executed if either the **execl** call or the child process fails. A **switch** statement could not be used where more than one child is being executed, since the process ids for children that fail are assigned in decreasing order, beginning with –1. The default case is executed when the child is successfully executed and **execl** has returned a normal child process id. Note that the default case checks for the file-not-found condition, since an exec call cannot detect this condition.

Example 10–4: Communicating Through a Pipe

```
/*  In this example, the parent writes a string to the pipe    *
 *  for the child to read.  The child then writes the string   *
 *  back to the pipe for the parent to read.  The wait         *
 *  function is called before the parent reads the string that *
 *  the child has passed back through the pipe.  Otherwise,    *
 *  the reads and writes would not be synchronized.            *
 *                                                             *
 *     PARENT:                                                 *
 *                                                             */

#include perror
#include climsgdef
#include stdio
#define inpipe 11
#define outpipe 12
#include processes

main()
{
   int pipes[2];
   int mode, status, cstatus, len;
   char *outbuf, *inbuf;

   if ((outbuf = malloc(512)) == 0)
      {
         printf("Parent - Outbuf allocation failed\n");
         exit();
      }

   if ((inbuf = malloc(512)) == 0)
      {
         printf("Parent - Inbuf allocation failed\n");
         exit();
      }

   if (pipe(pipes) == -1)
      {
         printf("Parent - Pipe allocation failed\n");
         exit();
      }

   dup2(pipes[0], inpipe);
   dup2(pipes[1], outpipe);
   strcpy(outbuf, "This is a test of two-way pipes.\n");

   status = vfork();
```

(Continued on next page)

Example 10–4 (Cont.): Communicating Through a Pipe

```
switch (status)
   {
      case 0:
         printf("Parent - Starting child\n");
         if ((status = execl("child", 0)) == -1)
            {
                printf("Parent - Exec failed");
                _exit();
            }
         break;

      case -1:
         printf("Parent - Child process failed\n");
         break;

      default:
         printf("Parent - Writing to child\n");
         if (write(outpipe, outbuf, strlen(outbuf)+1))
              == -1)
            {
               perror("Parent - Write failed");
               exit();
            }
         else
            {
               if ((status = wait(&cstatus)) == -1)
                  perror("Parent - Wait failed");

               if (cstatus == CLI$_IMAGEFNF)
                  printf("Parent - Child does not exist\n");
               else
                  {
                     printf("Parent - Reading from child\n");
                     if ((len = read(inpipe, inbuf, 512))
                         <= 0)
                        {
                           perror("Parent - Read failed");
                           exit();
                        }
                     else
                        {
```

(Continued on next page)

Example 10–4 (Cont.): Communicating Through a Pipe

```
                              printf("Parent: %s\n", inbuf);
                              printf("Parent - Child final \
status: %d\n", cstatus);
                           }
                     }
               }
            break;
      }
}

/*  This is a program separate from the parent process.          *
 *                                                               *
 *  CHILD:                                                       *
 *                                                               */

#define inpipe 11
#define outpipe 12

main()
{
   char *buffer;
   int len;

   if ((buffer = malloc(512)) == 0)
      {
         perror("Child - Buffer allocation failed\n");
         exit();
      }

   printf("Child - Reading from parent\n");
   if ((len = read(inpipe, buffer, 512)) <=0)
      {
         perror("Child - Read failed");
         exit();
      }
   else
      {
         printf("Child: %s\n", buffer);
         printf("Child - Writing to parent\n");
         if (write(outpipe, buffer, strlen(buffer)+1) == -1)
            {
               perror("Child - Write failed");
               exit();
            }
      }
}
```

Chapter 11

System Functions

The C programming language is a good choice for programmers who wish to write operating systems. For example, much of the UNIX operating system is written in C. When writing system programs, it is sometimes necessary to retrieve or modify the environment in which the program is running. This chapter describes VAX C RTL functions that accomplish this task as well as other miscellaneous functions.

11.1 Searching and Sorting Utilities

The following functions provide a method of searching and sorting array elements.

11.1.1 bsearch

The **bsearch** function performs a binary search. It searches an array of sorted objects for a specified object.

The syntax of the **bsearch** function is as follows:

```
#include stdlib
void *bsearch  (const void *key,
                const void *base,
                size_t nmemb,
                size_t size,
                int (*compar) (const void *, const void *));
```

Arguments

The arguments for the **bsearch** function are as follows:

key	A pointer to the object to be sought in the array.
base	A pointer to the initial member of the array.
nmemb	The number of objects in the array.
size	The size of an object in bytes.
compar	A pointer to the comparison function.

The pointers to the key and the member at the base of the array should be of type pointer-to-object and cast to type pointer-to-character.

Additional Information

The **bsearch** function returns a pointer to the matching member of the array or a NUL pointer if no match is found. The array must be previously sorted in increasing order according to the specified comparison function pointed to by *compar*.

Two arguments are passed to the comparison function pointed to by *compar*. The two arguments point to the objects being compared. Depending on whether the first argument is less than, equal to, or greater than the second argument, the comparison function returns an integer less than, equal to, or greater than zero.

If the key cannot be found in the array, a NUL pointer is returned.

It is not necessary for the comparison function (*compar*) to compare every byte in the array. Accordingly, the objects in the array can contain arbitrary data in addition to the data being compared.

Because it is declared as type "pointer-to-void", the value returned must be cast into type pointer-to-object.

11.1.2 qsort

The **qsort** function sorts an array of objects in place. It implements the "quicker-sort" algorithm. The syntax of the **qsort** function is as follows:

```
#include stdlib

void qsort     (void *base,
                size_t nmemb,
                size_t size,
                int (*compar) const void *, const void *));
```

Arguments

The arguments to the **qsort** function are as follows:

base A pointer to the initial member of the array. The pointer should be of type pointer-to-element and cast to type pointer-to-character.

nmemb The number of objects in the array.

size The size of an object in bytes.

compar A pointer to the comparison function.

Additional Information

Two arguments are passed to the comparison function pointed to by *compar*. The two arguments point to the objects being compared. Depending on whether the first argument is less than, equal to, or greater than the second argument, the comparison function returns an integer less than, equal to, or greater than zero.

The comparison function (*compar*) need not compare every byte, so arbitrary data may be contained in the objects in addition to the values being compared.

The order in the output of two objects that compare as equal is unpredictable.

11.2 Retrieving Process Information

The following sections describe the system functions that return process information.

11.2.1 ctermid

The **ctermid** function returns a character string giving the equivalence string of SYS$COMMAND. This is the name of the controlling terminal.

The syntax of the function is as follows:

```
#include stdlib

char *ctermid (char *str);
```

Arguments

The argument *str* must be a pointer to an array of characters. If this argument is NULL, the file name is stored internally and may be overwritten by the next **ctermid** call. Otherwise, the file name is stored beginning at the location indicated by the argument. The argument must point to a storage area of length L_ctermid (defined by the *stdio* definition module).

11.2.2 cuserid

The **cuserid** function returns a pointer to a character string containing the name of the user who initiated the current process.

The syntax of the function is as follows:

```
#include stdlib

char *cuserid (char *str);
```

Arguments

If the argument *str* is NULL, the user name is stored internally. If the argument is not NULL, it points to a storage area of length L_cuserid (defined by the *stdio* definition module), and the name is written into that storage. If the user name is NULL, the function returns a pointer to a NULL string.

11.2.3 getcwd

The **getcwd** function returns a pointer to the file specification for the current working directory.

The syntax of the **getcwd** function is as follows:

```
char *getcwd (char *buffer, unsigned int size, ...);
```

Arguments

The arguments to the **getcwd** function are as follows:

buffer — A pointer to a character string that is large enough to hold the directory specification.

If *buffer* is a NUL pointer, **getcwd** will obtain *size* bytes of space using **malloc**. In this case, the pointer returned by **getcwd** can be used as the argument in a subsequent call to **free**.

size — The length of the directory specification to be returned.

... — An optional argument that can be either 1 or 0. If you specify 1, the function **getcwd** returns the directory specification in VMS format. If you specify 0, the function **getcwd** returns the directory specification (pathname) in DEC/Shell format. If you do not specify this argument, this function returns the file name according to your current command language interpreter. For more information concerning DEC/Shell directory specifications, refer to Chapter 1, VAX C Run-Time Library Information.

Additional Information

If an error occurs, the **getcwd** function returns NULL with errno set to:

- ERANGE if *size* is not large enough.
- EINVAL if *size* is zero.
- ENOMEM if space for the returned string is not available for allocation.

11.2.4 getegid, geteuid, getgid, getuid

The get functions return, in VMS terms, group and member numbers from the user identification code (UIC). For example, if the UIC is [313,031], 313 is the group number, and 031 is the member number.

The syntax descriptions of the functions are as follows:

```
#include stdlib
unsigned int getgid  (void);
unsigned int getegid (void);
unsigned int getuid  (void);
unsigned int geteuid (void);
```

Additional Information

In VAX C, there is no difference between **getgid** and **getegid**. Both return the group number from the current UIC. Similarly, **getuid** and **geteuid** both return the member number from the current UIC.

11.2.5 getenv

The **getenv** function searches the environment array for the current process and returns the value associated with a specified environment name.

The syntax of the function is as follows:

```
#include stdlib

char *getenv (const char *name);
```

Arguments

The argument *name* can be one of the following:

- HOME—The user's login directory
- TERM—The type of terminal being used
- PATH—The default device and directory
- USER—The name of the user who initiated the process

In certain situations, **getenv** will attempt to perform a logical name translation on the user-specified argument. If the argument to **getenv** does not match any of the environment strings present in the user's environment array, then **getenv** will attempt to translate the user's argument as if it were a logical name. All four logical name tables are searched in the standard order. If no logical names exist, this function will attempt to translate the argument string as a command language interpreter (CLI) symbol; if it succeeds, it will return the translated symbol text. If it fails, the return value is NULL.

If your CLI is the DEC/Shell, the function does not attempt a logical name translation since Shell environment symbols are implemented as DCL symbols.

11.2.6 getpid

The **getpid** function returns the process ID of the current process.

The syntax of the function is as follows:

```
#include stdlib

int  getpid(void);
```

11.2.7 getppid

The **getppid** function returns the parent process ID of the calling process.

The syntax of the **getppid** function is as follows:

```
int getppid (void);
```

Additional Information

If the calling process does not have a parent process, the function returns zero.

11.3 Changing Process Information

The following sections describe the system functions that change information about your current process.

11.3.1 chdir

The **chdir** function changes the default directory.

The syntax of the function is as follows:

```
#include stdlib
int chdir (char *dir_spec);
```

Arguments

The argument *dir_spec* is a NUL-terminated character string naming a directory in either a VMS or DEC/Shell specification.

Additional Information

The **chdir** function returns zero if the directory is successfully changed to the given name, and -1 if the change fails.

If **chdir** is called in USER mode, the default directory change is only temporary. On image exit, the default is set to whatever it was before the execution of the image. If you want the change to be effective across images, you should call **chdir** from SUPERVISOR, EXECUTIVE, or KERNEL mode.

11.3.2 chmod

The **chmod** function changes the file protection of a file.

The syntax of the function is as follows:

```
#include stdlib
int chmod (char *file_spec, unsigned int mode);
```

Arguments

The arguments to the **chmod** function are as follows:

file_spec The name of a VMS or DEC/Shell file specification.

mode A file protection. Modes are constructed by performing a bitwise OR on any of the following values:

Value	Privilege
0400	OWNER:READ
0200	OWNER:WRITE
0100	OWNER:EXECUTE
0040	GROUP:READ
0020	GROUP:WRITE
0010	GROUP:EXECUTE
0004	WORLD:READ
0002	WORLD:WRITE
0001	WORLD:EXECUTE

When you supply a mode argument of zero, **chmod** gives the file the user's default file protection.

The system is always given the same privileges as the owner. A WRITE privilege also implies a DELETE privilege.

Additional Information

You must have a WRITE privilege for the file specified to change the mode. The function returns zero if the change was successful and –1 if unsuccessful.

11.3.3 chown

The **chown** function changes the owner UIC (user identification code) of the file; it returns zero on success and –1 on failure.

The syntax of the function is as follows:

```
#include stdlib

int chown (char *file_spec, unsigned int owner, unsigned int group);
```

Arguments

The arguments to the **chown** function are as follows:

file_spec	The address of an ASCII file name.
owner	The owner name.
group	The group names.

11.3.4 mkdir

The **mkdir** function creates a directory.

The syntax of the function is as follows:

```
#include stdlib
int mkdir (char *dir_spec, unsigned mode, ...);
```

Arguments

The arguments to the **mkdir** function are as follows:

dir_spec — A valid VMS or DEC/Shell directory specification that may contain a device name, as in the following:

```
DBAO:[BAY.WINDOWS]      /*   VMS         */
/dba0/bay/windows       /*  DEC/Shell    */
```

This specification cannot contain a node name, file name, file extension, file version, or a wildcard character. The same restriction applies to the DEC/Shell directory specifications. For more information concerning the restrictions on the DEC/Shell specifications, refer to Chapter 1, VAX C Run-Time Library Information.

mode — A file protection. See **chmod** in Section 11.3.2 for information concerning the specific file protections. All parent-directory defaults are applied to the new directory unless you override them.

... — Represents optional additional arguments as follows:

uic — The user identification code that identifies the owner of the created directory. If this argument is zero, VAX C gives the created directory the UIC of the parent directory. This optional argument is VAX C specific and is not portable.

max_versions — The maximum number of file versions to be retained in the created directory. The system automatically purges the directory keeping, at most, max_versions number of every file. If this argument is zero, VAX C does not place a limit on the maximum number of file versions. This optional argument is VAX C specific and is not portable.

r_v_number — Specifies on which volume (device) to place the created directory if the device is part of a volume set. If this argument is zero, VAX C arbitrarily places the created directory within the volume set. This optional argument is VAX C specific and is not portable.

Additional Information

This function returns zero to indicate success and a value of –1 to indicate failure.

If *dir_spec* specifies a path that includes directories, which do not exist, intermediate directories are also created. This differs from the behavior of the UNIX system wherein these intermediate directories must already exist and will not be created.

VAX C implements this function using the VMS Run-Time routine LIB$CREATE_DIR. For more information, refer to the *VAX/VMS Run-Time Library Routines Reference Manual.*

11.3.5 nice

The **nice** function increases or decreases process priority relative to the process base priority by the amount of the argument.

The syntax of the function is as follows:

```
#include stdlib

int nice (int increment);
```

Arguments

A positive argument (*increment*) decreases priority, and a negative argument increases priority. The resulting priority cannot be less than one or greater than the process's base priority.

Additional Information

The **nice** function returns zero on success and –1 on failure.

When a process calls **vfork**, the resulting child inherits the parent's priority.

11.3.6 setgid, setuid

The set functions are implemented for program portability and have no functionality. They always return zero (to indicate success).

The syntax descriptions of the functions are as follows:

```
#include stdlib

int setgid (unsigned int group_number);
int setuid (unsigned int member_number);
```

11.3.7 umask

The **umask** function creates a file protection mask that is used whenever a new file is created, and returns the old mask value.

The syntax of the function is as follows:

```
#include stdlib

int umask (unsigned int mode_complement);
```

Arguments

The argument *mode_complement* shows which bits to turn off when a new file is created.

Additional Information

The actual file protection of a newly created file is the bitwise AND of the mode with the complement of the **umask** argument. The mode is supplied when the file is opened. Initially, the mask is set from the current process default file protection.

See also Section 11.3.2.

11.4 Retrieving Time Information

The following sections describe system functions that return various time values.

11.4.1 asctime

The **asctime** function converts a broken-down time (see Section 11.4.7) into a 26-character string in the following form:

```
Sun Sep 16 01:03:52 1984\n\0
```

All of the fields have constant width.

The syntax of the **asctime** function is as follows:

```
#include time

char *asctime (const tm_t *timeptr);
```

Arguments

The argument *timeptr* is a pointer to the structure tm, which contains the broken-down time. The type tm_t is defined in the standard include module *time.h,* as follows:

```
typedef struct tm
      {
        short tm_sec, tm_min, tm_hour;
        short tm_mday, tm_mon, tm_year;
        short tm_wday, tm_yday, tm_isdst;
      }tm_t;
```

Additional Information

The **asctime** function converts the contents of tm into a 26-character string, as shown in the preceding example, and returns a pointer to the string. Subsequent calls to **asctime** or **ctime** may point to the same static string, which is overwritten by each call.

See Section 11.4.7 for a list of the members in tm.

11.4.2 clock

The **clock** function determines the CPU time (in microseconds) used since the beginning of the program execution. The time reported is the sum of the user and system times of the calling process and any terminated child processes for which the calling process has executed **wait** or **system**.

The syntax of the **clock** function is as follows:

```
#include time

clock_t clock (void);
```

Additional Information

The value returned by the **clock** function must be divided by the value of the macro CLK_TCK, as defined in the standard include module *time.h,* to obtain the time in seconds. The value (clock_t)–1 is returned if the processor time used is not available.

11.4.3 ctime

The **ctime** function converts a time in seconds, since 00:00:00 January 1, 1970, to an ASCII string to the form generated by the **asctime** function.

The syntax of the function is as follows:

```
#include time

char *ctime (const time_t *bintim);
```

Arguments

The argument *bintim* is a pointer to the time value to be converted.

Additional Information

The **ctime** function returns a pointer to the 26-character ASCII string. Successive calls to **ctime** overwrite any previous time values. The type time_t is defined in the standard include module *time.h* as follows:

```
typedef long int time_t
```

11.4.4 difftime

The **difftime** function computes the difference in seconds between the two times specified by its arguments; that is, *time2–time1*.

The syntax of the **difftime** function is as follows:

```
#include time

double difftime (time_t time1, time_t time2);
```

Arguments

Both *time2* and *time1* are of type time_t, which is defined in the standard include module time.h.

Additional Information

The **difftime** function returns the difference in seconds expressed as a double.

11.4.5 ftime

The **ftime** function returns the elapsed time since 00:00:00, January 1, 1970, in the structure timeb.

The syntax of the function is as follows:

```
#include time

void ftime (timeb_t *timeptr);
```

Arguments

The structure timeb_t is defined in the standard include module *time.h* as follows:

```
typedef struct timeb
   {
      time_t          time;
      unsigned short millitm;
      short           timezone;
      short           dstflag;
   }timeb_t;
```

The member time_t gives the time in seconds; the member millitm gives the fractional time in milliseconds; the members timezone and dstflag (daylight savings time flag) are always zero.

11.4.6 gmtime

The **gmtime** function converts a given calendar time into a broken-down time, expressed as Greenwich Mean Time (GMT).

The syntax of the **gmtime** function is as follows:

```
#include time

struct tm *gmtime (const time_t *timer);
```

Arguments

The argument *timer* is a pointer to an object of type time_t, which contains the calendar time.

Additional Information

The **gmtime** function returns a null pointer because GMT is not available under VMS.

This function is provided for reasons of conformance to the draft proposed ANSI standard for the C language.

11.4.7 localtime

The **localtime** function converts a time (expressed as the number of seconds elapsed since 00:00:00 January 1, 1970) into hours, minutes, seconds, and so on.

```
#include time

struct tm *localtime (const time_t *bintim);
```

Arguments

The argument *bintim* is a pointer to the time in seconds relative to 00:00:00 January 1, 1970. This time can be generated by the **time** function or supplied by the user.

Additional Information

The converted time value is placed in a time structure defined in the *time* definition module with the tag tm. The following member names are offsets into the structure:

tm_sec	—	time in seconds
tm_min	—	minutes
tm_hour	—	hours (24)
tm_mday	—	day of the month (1–31)
tm_mon	—	month (0–11)
tm_year	—	year (last two digits)
tm_wday	—	day of the week (0–6)
tm_yday	—	day of the year (0–365)
tm_isdst	—	daylight savings time (always 0)

The member names are integers.

The **localtime** function returns a pointer to the time structure. Successive calls to **localtime** overwrite the structure.

11.4.8 time

The **time** function returns the time elapsed since 00:00:00, January 1, 1970, in seconds.

The syntax of the function is as follows:

```
#include time

time_t time (time_t *time_location);
```

Arguments

The argument *time_location* is either null or a pointer to the place where the returned time is also stored.

11.4.9 times

The **times** function returns the accumulated times of the current process and of its terminated child processes.

The syntax of the function is as follows:

```
#include time

void times (tbuffer_t *buffer);
```

Arguments

The type tbuffer_t is defined in the standard include module *time.h* as follows:

```
typedef struct tbuffer
   {
      int proc_user_time;
      int proc_system_time;
      int child_user_time;
      int child_system_time;
   }tbuffer_t;
```

Additional Information

For both process and children times, the structure breaks down the time by user and system time. Since VMS does not differentiate between system and user time, all system times are returned as zero. Accumulated CPU times are returned in 10-millisecond units.

11.5 VAXC$CRTL_INIT

The **VAXC$CRTL_INIT** function allows you to call the VAX C RTL from other languages. It initializes the run-time environment and establishes both an exit and condition handler.

The following example shows a Pascal program that calls the VAX C RTL using the **VAXC$CRTL_INIT** function:

```
PROGRAM TESTC (input,output);
PROCEDURE VAXC$CRTL_INIT; extern;
BEGIN
   VAXC$CRTL_INIT;
END.
```

11.6 Program Examples

Example 11–1 and Example 11–2 illustrate the use of the **cuserid** function.

Example 11–1: Accessing the User Name

```
/*  Using cuserid, this program returns the user name.          */

#include stdio
#include perror

main()
{
   static char string[L_cuserid] = "";
   cuserid(string);
   printf("Initiating user: %s\n", string);
}
```

Given that a user named TOLLIVER is running the program, the output to stdout is as follows:

```
$ RUN  EXAMPLE1 [RETURN]
Initiating user: TOLLIVER
```

Example 11–2 produces the same output.

Example 11–2: A Second Way to Access the User Name

```
/*  Using cuserid, this program returns the user name.          */

#include stdio

main()
{
                                  /*  Zero: a null argument.     */
    printf("Initiating user: %s\n", cuserid(0));
}
```

Example 11–3 illustrates the **getenv** function.

Example 11–3: Accessing Terminal Information

```
cfunc()
{
    printf("Terminal type: %s\n", getenv("TERM"));
}
```

Given that the terminal in use is a DIGITAL VT100 in 132-column mode, sample output from the previous program is as follows:

```
$ RUN  EXAMPLE3 RETURN
Terminal type: vt100-132
```

Example 11–4 illustrates how to use **getenv** to find the user's default login directory and **chdir** to change to that directory.

Example 11–4: Manipulating the Default Directory

```
/*  This program performs the equivalent to the DCL command    *
 *  SET DEFAULT SYS$LOGIN. Once the program exits, however,    *
 *  the directory is reset to the directory from which the     *
 *  program was run.                                           */

#include stdio

main()
{
   char *dir;
   int i;

   dir = getenv("HOME");
   if ((i = chdir(dir)) != 0)
      {
         perror("Cannot set directory");
         exit();
      }

   printf("Current directory: %s\n", dir);
}
```

Sample output from the previous program is as follows:

```
$ RUN EXAMPLE4 RETURN
Current directory: dba0:[tolliver]
$
```

Example 11–5 illustrates how to use the **time** and **localtime** functions to print the correct date and time at the terminal.

Example 11–5: Printing the Date and Time

```
/*  The time function returns the time in seconds; the         *
 *  localtime function converts  the time to hours, minutes,   *
 *  and so on.                                                 */

#include time

main()
{
   struct  tm  *time_structure;
   int time_val, i;

   static char *weekday[7] = {"Sunday", "Monday", "Tuesday",
                              "Wednesday", "Thursday", "Friday",
                              "Saturday"};

   static char *month[12] = {"January","February","March",
                             "April","May","June","July",
                             "August","September",
                             "October","November","December"};

   static char *hour[2] = {"AM","PM"};

   time(&time_val);
   time_structure = localtime(&time_val);

                                  /* Print the date            */
   printf("Today is %s, %s %d, 19%d\n",
           weekday[time_structure->tm_wday],
           month[time_structure->tm_mon],
           time_structure->tm_mday,
           time_structure->tm_year);

/* Time conversion and print using 12 hour clock               */

   if(time_structure->tm_hour > 12)
      {
         time_structure->tm_hour = (time_structure->tm_hour)-12;
         i = 1;
      }
   else
      i = 0;

   printf("The time is %d:%02d %s\n",
           time_structure->tm_hour,
           time_structure->tm_min,
           hour[i]);
}
```

Sample output from the previous example is as follows:

```
$ RUN  EXAMPLE5 RETURN
Today is Thursday, February 7, 1985
The time is 10:18 AM
$
```

Chapter 12

Curses Screen Management Functions and Macros

Curses, the VAX C Screen Management Package, is composed of VAX C RTL functions and macros that create and modify defined sections of the terminal screen and optimize cursor movement. Using the screen management package, you can develop a user interface that is both visually attractive and user-friendly. Curses is terminal-independent and provides simplified terminal screen formatting and efficient cursor movement.

Curses is implemented using the terminal-independent Screen Management Software, which is a part of the VMS Run-Time Library. For portability purposes, most functions and macros are designed to perform in much the same way as those in other C implementations. However, VAX C Curses depends upon VMS and its Screen Management Software, so performance of some functions and macros may differ slightly from those of other implementations. Some functions and macros available on other systems are not available with VAX C Curses. The functions and macros **[w]clrattr**, **[w]insstr**, **mv[w]insstr**, and **[w]setattr** are VAX C specific and are not portable.

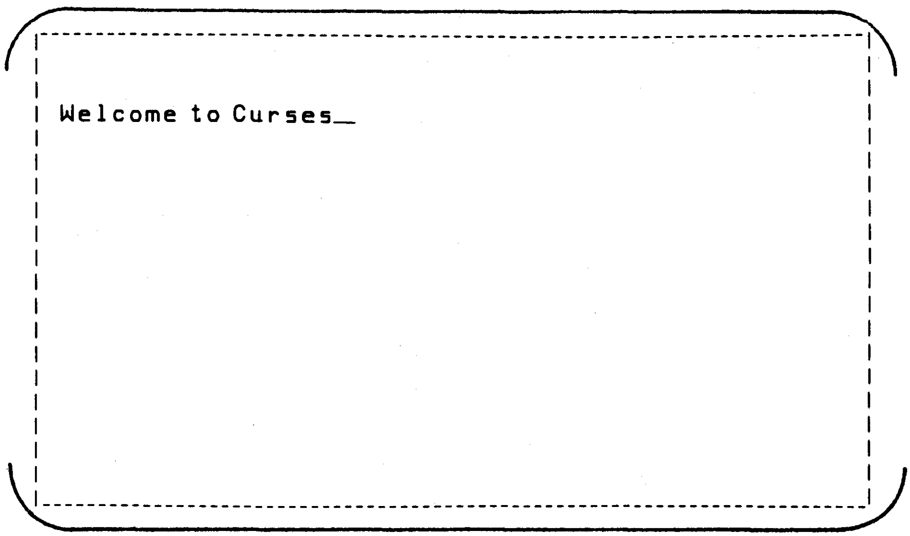
12.1 Curses Terminology

The purpose of this section is to explain some of the Curses terminology and to show you how Curses looks on the terminal screen.

You can imagine a Curses application as being a series of overlapping windows. Window overlapping is called *occlusion*. To distinguish the boundaries of these occluding windows, you can outline the rectangular windows with specified characters, or you can turn on the reverse video option (make the window a light background with dark writing).

Initially, there are two windows the size of the terminal screen that are predefined by Curses. These windows are called *stdscr* and *curscr*. The stdscr window is specifically defined for your use. Many Curses macros default to this window. For example, if you draw a box around stdscr, move the cursor to the left-corner area of the screen, write a string to stdscr, and then display stdscr on the terminal screen, your display would look like that in Figure 12–1.

Figure 12–1: Example of the stdscr Window



ZK-5752-86

The second predefined window, curscr, is designed for internal Curses work; it is an image of what is currently displayed on the terminal screen. The only VAX C Curses function that will accept this window as an argument is **clearok**. Do not write to or read from curscr. Use stdscr and user-defined windows for all of your Curses applications.

12.1.1 User-Defined Windows

You may choose to occlude stdscr with your own windows. The size and location of each window is given in terms of the number of lines, the number of columns, and the starting position. The lines and columns of the terminal screen form a coordinate system, or grid, on which the windows are formed. You specify the starting position of a window with the (y, x) coordinates on the terminal screen where the upper left corner of the window is located. The coordinates (0, 0) on the terminal screen, for example, are the upper left corner of the screen. The entire area of the window must be within the terminal screen borders, windows being as small as a single character or as large as the entire terminal screen. You may create as many windows as memory allows.

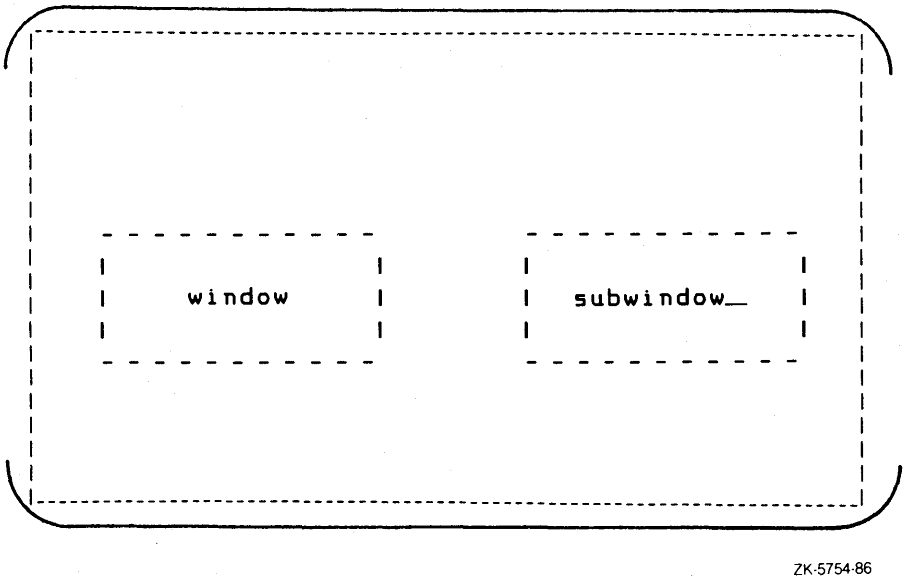
When writing to or deleting from windows, changes do not appear on the terminal screen until the window is *refreshed*. When refreshing a window, you place the updated window onto the terminal screen, leaving the rest of the screen unaltered.

All user-defined windows, by default, occlude stdscr. You can create two or more windows that occlude each other as well as stdscr. When writing data to one occluding window, the data is *not* written to the underlying window.

You can create overlapping windows (called *subwindows*); a declared window must contain the entire area of its subwindow. When writing data to a subwindow or to the portion of the window overlapped by the subwindow, both windows contain the new data. For instance, if you write data to a subwindow and then delete that subwindow, the data is still present on the underlying window.

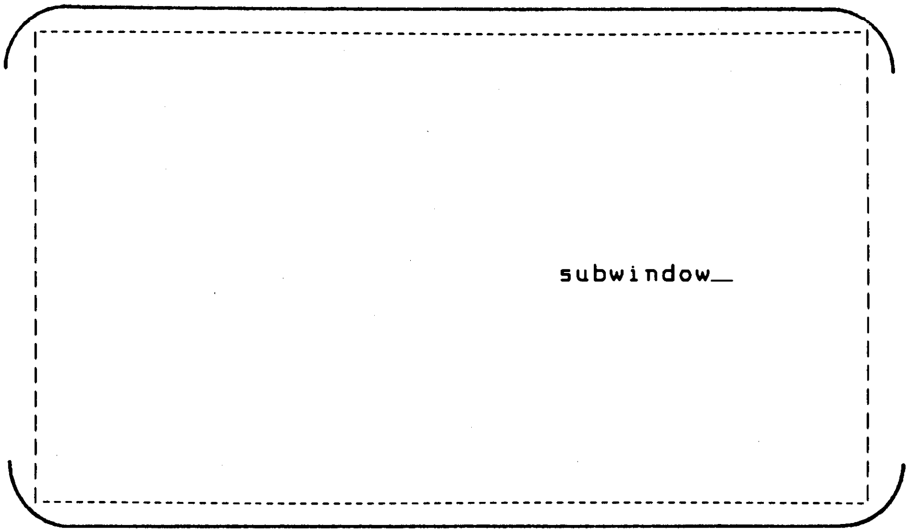
If creating both a window that occludes stdscr and a subwindow of stdscr, your terminal screen will look similar to Figure 12–2.

Figure 12–2: Diplaying Windows and Subwindows



If you delete both the user-defined window and the subwindow, and then update the terminal screen with the new image, the screen would appear like that in Figure 12–3.

Figure 12–3: Illustration of an Updated Terminal Screen



ZK-5753-86

Notice that the string written on the window is deleted, but the string written on the subwindow remains on stdscr.

12.2 Getting Started with Curses

There are commands which you must use to initialize and restore the terminal screen when using Curses Screen Management functions and macros. Also, there are predefined variables and constants on which Curses depends. Example 12–1 shows how to set up a program using Curses.

Example 12–1: A Curses Program

```
❶ #include curses
❷ WINDOW  *win1, *win2, *win3;
  main()
  {
❸    initscr();
         .
         .
         .
     endwin();
  }
```

The following numbers correspond to the numbers in the previous example:

❶ The preprocessor directive includes the *curses* definition module which defines the data structures and variables used to implement Curses. The module *curses* includes the module *stdio,* so it is not necessary to duplicate this action by including *stdio* again in the program source code. You must include *curses* to use any of the Curses functions or macros.

❷ In the example, WINDOW is a data structure defined in *curses.* You must declare each user-specified window in this manner. In the previous example, the three defined windows are win1, win2, and win3.

❸ The **initscr** and **endwin** functions begin and end the window editing session. The **initscr** function clears the terminal screen and allocates space for the windows stdscr and curscr. The **endwin** function deletes all windows and clears the terminal screen.

Most Curses users wish to define and modify windows. Example 12–2 shows you how to define and write to a single window.

Example 12–2: Manipulating Windows

```
   #include curses

   WINDOW  *win1, *win2, *win3;

   main()
   {
      initscr();
❶     win1 = newwin(24, 80, 0, 0);
❷     mvwaddstr(win1, 2, 2, "HELLO");
         .
         .
         .
      endwin();
   }
```

The following numbers correspond to the numbers in the previous example:

❶ The **newwin** function defines a window 24 rows high, 80 columns wide, and a starting position at coordinates (0, 0), the upper left corner of the terminal screen. The program assigns these attributes to win1. Note that the coordinates are specified as follows: (lines, columns) or (y, x).

❷ The **mvaddstr** macro performs the same task as a call to the separate macros **move** and **addstr**. The **mvaddstr** macro moves the cursor to the specified coordinates and writes a string onto stdscr.

Most Curses macros update stdscr by default. Curses functions that update other windows have the same name as the macros but with the added prefix "w". For example, the **addstr** macro adds a given string to stdscr at the current cursor position. So, the **waddstr** function adds a given string to a specified window at the current cursor position.

When updating a window, specify the cursor position relative to the origin of the window, not the origin of the terminal screen. For example, if a window has a starting position of (10, 10) and you wanted to add a character to the window at its starting position, you specify the coordinates (0, 0), not (10, 10).

The string HELLO in the preceding example does not appear on the terminal screen until you refresh the screen. You accomplish this by using the **wrefresh** function. Example 12–3 illustrates how to display the contents of win1 on the terminal screen:

Example 12–3: Refreshing the Terminal Screen

```
#include curses

WINDOW  *win1, *win2, *win3;

main()
{
   initscr();

   win1 = newwin(22, 60, 0, 0);
   mvwaddstr(win1, 2, 2, "HELLO");
   wrefresh(win1);
       .
       .
       .
   endwin();
}
```

The **wrefresh** function updates just the region of the specified window on the terminal screen. When the program is executed, the string HELLO appears on the terminal screen until the program executes the **endwin** function. The **wrefresh** function only refreshes the part of the window on the terminal screen that is not overlapped by another window. If win1 was overlapped by another window and you wanted all of win1 to be displayed on the terminal screen, you call the **touchwin** function.

12.3 Predefined Variables and Constants

There is a group of variables, defined in the *curses* definition module, that will be useful when using Curses. Also, there is a group of constants defined in *curses*, using the **#define** preprocessor directive, that will be useful. Table 12–1 describes the variables and constants defined in the *curses* definition module.

Table 12–1: Curses Predefined Variables and #define Constants

Name	Type	Description
curscr	WINDOW *	VAR: Window of current screen
stdscr	WINDOW *	VAR: Default window
LINES	int	VAR: Number of lines on terminal screen
COLS	int	VAR: Number of columns on terminal screen
ERR	—	CON: Flag (0) for failed routines
OK	—	CON: Flag (1) for successful routines
TRUE	—	CON: Boolean true flag (1)
FALSE	—	CON: Boolean false flag (0)
_BLINK	—	CON: Parameter for **setattr** and **clrattr**
_BOLD	—	CON: Parameter for **setattr** and **clrattr**
_REVERSE	—	CON: Parameter for **setattr** and **clrattr**
_UNDERLINE	—	CON: Parameter for **setattr** and **clrattr**

For example, you can use the predefined variable ERR to test the success or failure of a Curses function. Example 12–4 shows how to perform such a test.

Example 12–4: Curses Predefined Variables

```
#include curses

WINDOW  *win1, *win2, *win3;

main()
{
   initscr();
   win1 = newwin(10, 10, 1, 5);
       .
       .
       .
   if (mvwin(win1, 1, 10) == ERR)
      addstr("The MVWIN function failed.");
       .
       .
       .
   endwin();
}
```

In the example, if the **mvwin** function fails, then the program adds a string to stdscr explaining the outcome. The Curses function **mvwin** moves the starting position of a window.

12.4 Cursor Movement

In the UNIX system environment, you can use Curses functions to move the cursor across the terminal screen. With other implementations, you can either allow Curses to move the cursor using the **move** function, or you can specify the origin and the destination of the cursor to the **mvcur** function, so as to move the cursor in a more efficient fashion.

In VAX C, the two functions are functionally equivalent and move the cursor with the same efficiency.

Example 12–5 illustrates the use of the **move** and **mvcur** functions:

Example 12–5: The Cursor Movement Functions

```
#include curses

main()
{
   initscr();
        .
        .
        .
❶  clear();
❷  move(10, 10);
❸  move(LINES/2, COLS/2);
❹  mvcur(0, COLS-1, LINES-1, 0);
        .
        .
        .
   endwin();
}
```

The following numbers correspond to the numbers in the previous example:

❶ The **clear** macro erases stdscr and then positions the cursor at coordinates (0,0).

❷ The first occurrence of **move** moves the cursor to coordinates (10, 10).

❸ The second occurrence of **move** uses the predefined variables LINES and COLS to calculate the center of the screen (by calculating the value of half the number of LINES and COLS on the screen).

❹ The occurrence of **mvcur** forces absolute addressing. The function **mvcur** can absolutely address the lower left corner of the screen by claiming that the cursor is presently in the upper right corner. You may use this method when unsure of the current position of the cursor, although **move** is just as applicable.

12.5 The Curses Functions and Macros

Most Curses functions and macros are listed in pairs where the first is a macro and the second is a function beginning with the prefix "w," for "window." These prefixes are delimited by brackets ([ ]). For example, **[w]addstr** designates the **addstr** macro and the **waddstr** function. The macros default to the window stdscr; the functions accept as an argument a specified window. When working with macros, take care in specifying arguments that may cause side effects, such as those that use the increment and decrement operators. For an explanation of passing arguments to macros, refer to the *Guide to VAX C*.

All argument names given in the following syntax descriptions show their order and their type. Argument names are only suggestions.

12.5.1 [w]addch

The **addch** macro and the **waddch** function add the character ch to the window at the current position of the cursor.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

addch(ch)
int waddch (WINDOW *win, char ch);
```

Arguments

The argument *ch* is an object of type char. If the character is a newline (\n), the macro and function clear the line to the end, and move the current (y, x) coordinates to the next line at the same x coordinate. A return (\r) moves the character to the beginning of the line on the window. Tabs (\t) expand into spaces in the normal tabstop positions of every eight characters.

Additional Information

When **waddch** is used on a subwindow, it writes the character onto the underlying window as well. The **waddch** function returns ERR if it would cause the screen to scroll illegally (see Section 12.5.46).

12.5.2 [w]addstr

The **addstr** macro and the **waddstr** function add the string pointed to by *str* to the window at the current position of the cursor.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

addstr(str)
int waddstr (WINDOW *win, char *str);
```

Arguments

The argument *str* is a pointer to a character string.

Additional Information

When **waddstr** is used on a subwindow, the string is written onto the underlying window as well. The **waddstr** function returns ERR if it would cause the screen to scroll illegally (see Section 12.5.46), but it places as much of the string onto the window as possible.

12.5.3 box

The **box** function draws a box around the window using the character *vert* as the character for drawing the vertical lines of the rectangle, and *hor* for drawing the horizontal lines of the rectangle.

The syntax of the function is as follows:

```
#include curses
#define bool int

int box (WINDOW *win, char vert, char hor);
```

Additional Information

The **box** function copies boxes drawn on subwindows onto the underlying window. Use caution when using functions such as **overlay** and **overwrite** with boxed subwindows. Such functions copy the box onto the underlying window.

12.5.4 [w]clear

The **clear** macro and the **wclear** function erase the contents of the specified window and reset the cursor to coordinates (0, 0).

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

clear()
int wclear (WINDOW *win);
```

12.5.5 clearok

The **clearok** macro sets the clear flag for the window *win*.

The syntax of the macro is as follows:

```
#include curses
#define bool int

clearok (WINDOW *win, bool boolf);
```

Arguments

The arguments to the **clearok** macro are as follows:

win — The entire size of the terminal screen. You can use the windows stdscr and curscr with **clearok**.

boolf — A Boolean value of TRUE or FALSE. If the argument, boolf, is TRUE, this forces a clearscreen to be printed on the next call to **refresh**, or stops the screen from being cleared if boolf is FALSE. The constant boolf is defined in the *curses* definition module.

Additional Information

Unlike **clear**, this macro does not alter the contents of the window. If the argument, win, is curscr, the next call to **refresh** causes a clearscreen, even if the window passed to **refresh** is not a window the size of the entire terminal screen.

12.5.6 [w]clrattr

The **clrattr** macro and the **wclrattr** function deactivate the video display attribute attr within the window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

clrattr(attr)
int wclrattr (WINDOW *win, int attr);
```

Arguments

The video display attributes, specified by the argument *attr*, are blinking, boldface, reverse video, and underlining, and are represented by the defined constants _BLINK, _BOLD, _REVERSE, and _UNDERLINE. You can clear multiple attributes by separating them with a bitwise OR operator (|) as follows:

```
clrattr(_BLINK | _UNDERLINE);
```

Additional Information

The **clrattr** macro and the **wclrattr** function are VAX C specific and are not portable.

12.5.7 [w]clrtobot

The **clrtobot** macro and the **wclrtobot** function erase the contents of the window from the current position of the cursor to the bottom of the window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

clrtobot()
int wclrtobot (WINDOW *win);
```

12.5.8 [w]clrtoeol

The **clrtoeol** macro and the **wclrtoeol** function erase the contents of the window from the current cursor position to the end of the line on the specified window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

clrtoeol()
int wclrtoeol (WINDOW *win);
```

12.5.9 [no]crmode

In the UNIX system environment, the **crmode** and **nocrmode** macros set and unset the terminal from cbreak mode; they are provided only for UNIX software compatibility and they have no functionality in the VMS environment.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

crmode()
nocrmode()
```

12.5.10 [w]delch

The **delch** macro and the **wdelch** function delete the character on the specified window at the current position of the cursor.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

delch()
int wdelch (WINDOW *win);
```

Additional Information

Each of the following characters on the same line shifts to the left, and Curses appends a blank character to the end of the line.

12.5.11 [w]deleteln

The **deleteln** macro and the **wdeleteln** function delete the line at the current position of the cursor.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

deleteln()
int wdeleteln (WINDOW *win);
```

Additional Information

Every line below the deleted line moves up, and the bottom line becomes blank. The current (y, x) coordinates of the cursor remain unchanged.

12.5.12 delwin

The **delwin** function deletes the specified window from memory.

The syntax of the function is as follows:

```
#include curses
#define bool int

int delwin (WINDOW *win);
```

Additional Information

If the window being deleted contains a subwindow, the subwindow is invalidated. You should delete subwindows before deleting the underlying window. The **delwin** function refreshes all covered windows of the deleted window.

12.5.13 [no]echo

The **echo** and **noecho** macros set the terminal so that characters may or may not be echoed on the terminal screen.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

echo()
noecho()
```

Additional Information

The **noecho** macro may be helpful when accepting input from the terminal screen with **wgetch** and **wgetstr**; it prevents the input characters from being written onto the specified window.

12.5.14 endwin

The **endwin** function clears the terminal screen and frees any virtual memory allocated to Curses data structures.

The syntax of the function is as follows:

```
#include curses
#define bool int

void endwin (void);
```

Additional Information

You must call this function before exiting in order to restore the previous environment of the terminal screen.

12.5.15 [w]erase

The **erase** macro and the **werase** function erase the window by "painting" it with blanks.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

erase()
int werase (WINDOW *win);
```

Additional Information

Both **erase** and **werase** leave the cursor at the current position on the terminal screen after completion; they do not return the cursor to the home coordinates of (0, 0).

12.5.16 [w]getch

The **getch** macro and the **wgetch** function get a character from the terminal screen and echo it on the specified window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

getch()
char wgetch (WINDOW *win);
```

Additional Information

The **getch** macro and the **wgetch** function return ERR if the screen scrolls illegally (see Section 12.5.46); otherwise, they return the character. The macro and function **getch** and **wgetch** refresh the specified window before fetching a character.

12.5.17 [w]getstr

The **getstr** macro and the **wgetstr** function get a string from the terminal screen, store it in the variable str, and echo it on the specified window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

getstr(str)
int wgetstr (WINDOW *win, char *str);
```

Arguments

The argument *str* must be large enough to hold the character string fetched from the window.

Additional Information

The **getstr** macro and the **wgetstr** function refresh the specified window before fetching a string. The newline terminator is stripped from the fetched string. They return ERR if the screen scrolls illegally (see Section 12.5.46).

12.5.18 getyx

The **getyx** macro puts the (y, x) coordinates of the current cursor position on *win* in the variables y and x.

The syntax of the macro is as follows:

```
#include curses
#define bool int

getyx (WINDOW *win, int y, int x);
```

Arguments

The arguments *y* and *x* must be valid VAX C lvalues. For more information concerning lvalues, refer to the *Guide to VAX C*.

12.5.19 [w]inch

The **inch** macro and the **winch** function return the character at the current cursor position on the specified window without making changes to the window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

inch()
char winch (WINDOW *win);
```

12.5.20 initscr

The **initscr** function initializes the terminal-type data and all screen functions. You must call **initscr** before using any of the screen functions or macros.

The syntax of the function is as follows:

```
#include curses
#define bool int

void initscr (void);
```

12.5.21 [w]insch

The **insch** macro and the **winsch** function insert the character ch at the current cursor position in the specified window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

insch(ch)
int winsch (WINDOW *win, char ch);
```

Additional Information

After the character is inserted, each character on the line shifts to the right, and Curses deletes the last character in the line. The macro and function return ERR if the screen scrolls illegally (see Section 12.5.46).

12.5.22 [w]insertln

The **insertln** macro and the **winsertln** function insert a line above the line containing the current cursor position.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

insertln()
int winsertln (WINDOW *win);
```

Additional Information

Every line below the current line shifts down, and the bottom line disappears. The inserted line is blank and the current (y, x) coordinates remain the same. The macro and function return ERR if the screen scrolls illegally (see Section 12.5.46).

12.5.23 [w]insstr

The **insstr** macro and the **winsstr** function insert a string at the current cursor position on the specified window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

insstr(str)
int winsstr (WINDOW *win, char *str);
```

Additional Information

Each character after the string shifts to the right, and the last character disappears. The macro and function return ERR if the screen scrolls illegally (see Section 12.5.46). The macro and function are VAX C specific and are not portable.

12.5.24 longname

The **longname** function assigns the full terminal name to *name* which must be large enough to hold the character string.

The syntax of the function is as follows:

```
#include curses
#define bool int

void longname (char *termbuf, char *name);
```

Arguments

The argument *name* is a character string buffer with a minimum length of 64 characters.

Additional Information

The terminal name is in a readable format so that you can double-check to be sure that Curses has correctly identified your terminal. The dummy argument termbuf is required for UNIX software compatibility and has no functionality in the VMS environment. If portability is a concern, you must write a set of dummy routines to perform the functionality provided by the database *termcap* provided in the UNIX system environment.

12.5.25 leaveok

The **leaveok** macro signals Curses to leave the cursor at the current coordinates after an update to the window.

The syntax of the macro is as follows:

```
#include curses
#define bool int

leaveok (WINDOW *win, bool boolf);
```

Arguments

The argument *boolf* is a Boolean TRUE or FALSE value. If *boolf* is TRUE, the cursor remains in place after the last update and the coordinate setting on *win* changes accordingly. If *boolf* is FALSE, then the cursor moves to the currently specified (y, x) coordinates of *win*. Values for *boolf* are defined in the *curses* definition mode.

Additional Information

The **leaveok** macro defaults to moving the cursor to the current coordinates of *win*.

12.5.26 [w]move

The **move** macro and the **wmove** function change the current cursor position on the specified window to the coordinates (y, x).

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

move(y, x)
int wmove (WINDOW *win, int, y, int, x);
```

Additional Information

The macro and function return ERR if the screen scrolls illegally (see Section 12.5.46).

12.5.27 mv[w]addch

The **mvaddch** and **mvwaddch** macros move the cursor to (y , x) and add the character ch to the specified window.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

mvaddch (int y, int x, char ch);
mvwaddch (WINDOW *win, int y, int x, char ch);
```

Arguments

If the argument *ch* is a newline (\n), the macro and function clear the line to the end, and move the specified (y, x) coordinates to the next line at the same x coordinate. A return (\r) moves the character to the beginning of the specified line. Tabs (\t) are expanded into spaces in the normal tabstop positions of every eight characters.

12.5.28 mv[w]addstr

The **mvaddstr** and **mvwaddstr** macros move the cursor to (y, x) and add the specified string, to which *str* points, to the specified window.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

mvaddstr (int y, int x, char *str);
mvwaddstr (WINDOW *win, int y, int x, char *str);
```

12.5.29 mvcur

The **mvcur** function moves the terminal's cursor from (lasty, lastx) to (newy, newx).

The syntax of the function is as follows:

```
#include curses
#define bool int

int mvcur (int lasty, int lastx, int newy, int newx);
```

Additional Information

This function is functionally equivalent to **move**.

12.5.30 mv[w]delch

The **mvdelch** and **mvwdelch** macros move the cursor to (y, x) and delete the character on the specified window.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

mvdelch (int y, int x);
mvwdelch (WINDOW *win, int y, int x);
```

Additional Information

Each of the following characters on the same line shifts to the left, and the last character becomes blank.

12.5.31 mv[w]getch

The **mvgetch** and **mvwgetch** macros move the cursor to (y, x), get a character from the terminal screen, and echo it on the specified window.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

mvgetch (int y, int x);
mvwgetch (WINDOW *win, int y, int x);
```

12.5.32 mv[w]getstr

The **mvgetstr** and **mvwgetstr** macros move the cursor to (y, x), get a string from the terminal screen, store it in the variable str which must be large enough to contain the string, and echo it on the specified window.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

mvgetstr (int y, int x, char *str);
mvwgetstr (WINDOW *win, int y, int x, char *str);
```

Additional Information

The macros strip the newline terminator (\n) from the string.

12.5.33 mv[w]inch

The **mvinch** and **mvwinch** macros move the cursor to (y, x) and return the character on the specified window without making changes to the window.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

mvinch (int y, int x);
mvwinch (WINDOW *win, int y, int x);
```

12.5.34 mv[w]insch

The **mvinsch** and **mvwinsch** macros move the cursor to (y, x) and insert the character ch in the specified window.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

mvinsch (char ch, int y, int x);
mvwinsch (WINDOW *win, int y, int x, char ch);
```

Additional Information

After the character is inserted, each character on the line shifts to the right, and the last character disappears.

12.5.35 mv[w]insstr

The **mvinsstr** and **mvwinsstr** macros move the cursor to (y, x) and insert a string in the specified window.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

mvinsstr (int y, int x, char *str);
mvwinsstr (WINDOW *win, int y, int x, char *str);
```

Additional Information

Each character after the string shifts to the right, and the last character disappears. The macro and function are VAX C specific and are not portable.

12.5.36 mvwin

The **mvwin** function moves the starting position of the window to the specified (y, x) coordinates.

The syntax of the function is as follows:

```
#include curses
#define bool int

mvwin (WINDOW *win, int y, int x);
```

Additional Information

If moving the window puts part or all of the window off the edge of the terminal screen, the **mvwin** function returns ERR and the terminal screen remains unaltered. When moving subwindows, the function does not rewrite the contents of the subwindow on the underlying window at the new position. If anything is written to the subwindow after the move, the function also writes to the underlying window.

12.5.37 newwin

The **newwin** function creates a new window with *numlines* lines and *numcols* columns starting at the coordinates *(begin_y, begin_x)* on the terminal screen.

The syntax of the function is as follows:

```
#include curses
#define bool int

WINDOW *newwin (int numlines, int numcols, int begin_y, int begin_x);
```

Arguments

If either *numlines* or *numcols* is zero, then the function sets that dimension to (LINES—*begin_y*) or (COLS – *begin_x*) respectively. Thus, to get a new window of dimensions LINES by COLS, use *newwin (0, 0, 0, 0)*.

12.5.38 [no]nl

The **nl** and **nonl** macros are provided only for UNIX software compatibility and have no functionality in the VMS environment.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

nl()
nonl()
```

12.5.39 overlay

The **overlay** function nondestructively superimposes *win1* on *win2*. The function writes the contents of *win1*, insofar as they fit, on *win2* beginning at the starting coordinates of both windows. Blanks on *win1* leave the contents of the corresponding space on *win2* unaltered. The function **overlay** copies as much of a window's box as possible.

The syntax of the function is as follows:

```
#include curses
#define bool int

int overlay (WINDOW *win1, WINDOW *win2);
```

12.5.40 overwrite

The **overwrite** function destructively writes the contents of *win1* on *win2*. The function writes the contents of *win1*, insofar as they fit, on *win2*, beginning at the starting coordinates of both windows. Blanks on *win1* are written on *win2* as blanks. The function **overwrite** copies as much of a window's box as possible.

The syntax of the function is as follows:

```
#include curses
#define bool int

int overwrite (WINDOW *win1, WINDOW *win2);
```

12.5.41 [w]printw

The **printw** and **wprintw** functions perform a **printf** (see Chapter 2, Standard I/O Functions and Macros) on the window starting at the current position of the cursor.

The syntax descriptions of the functions are as follows:

```
#include curses
#define bool int

printw (format_spec [,output_src, . . . ])
int wprintw (WINDOW *win, char *format_spec, ...);
```

Arguments

The formatting specification (*fmt_spec*) and the other arguments are identical to those used with the function **printf**.

Additional Information

The **printw** and **wprintw** functions format and then print the resultant string to the window using **addstr**. The functions return ERR if the screen scrolls illegally (see Section 12.5.46).

12.5.42 [no]raw

The **raw** and **noraw** macros are provided only for UNIX software compatibility and have no functionality in the VMS environment.

The syntax descriptions of the macros are as follows:

```
#include curses
#define bool int

raw()
noraw()
```

12.5.43 [w]refresh

The **refresh** macro and the **wrefresh** function repaint the specified window on the terminal screen.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

refresh()
int wrefresh (WINDOW *win);
```

Additional Information

The result of this process is that the portion of the window which is not occluded by subwindows or other windows appears on the terminal screen. To see the entire occluded window on the terminal screen, call the **touchwin** function instead of **refresh** or **wrefresh**.

12.5.44 [w]scanw

The **scanw** and **wscanw** functions perform a **scanf** (see Chapter 2, Standard I/O Functions and Macros) on the window.

The syntax descriptions of the functions are as follows:

```
#include curses
#define bool int

scanw (format_spec [,input_src, . . . ])
int wscanw (WINDOW *win, char *format_spec, ...);
```

Arguments

The formatting specification (*format_spec*) and the other arguments are identical to those used with the function **scanf**.

Additional Information

The **scanw** macro and **wscanw** function accept, format, and return a line of text from the terminal screen. They return ERR if the screen scrolls illegally (see Section 12.5.46).

12.5.45 scroll

The **scroll** function moves all of the lines on the window up one line. The top line scrolls off the window and the bottom line becomes blank.

The syntax of the function is as follows:

```
#include curses
#define bool int

int scroll (WINDOW *win);
```

12.5.46 scrollok

The **scrollok** macro sets the scroll flag for the specified window.

The syntax of the macro is as follows:

```
#include curses
#define bool int

scrollok (WINDOW *win, bool boolf);
```

Arguments

The argument *boolf* is a Boolean TRUE or FALSE value. If *boolf* is FALSE, scrolling is not allowed. This is the default setting. The argument *boolf* is defined in the *curses* definition module.

12.5.47 [w]setattr

The **setattr** macro and the **wsetattr** function activate the video display attribute *attr* within the window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

setattr(attr)
int wsetattr (WINDOW *win, int attr);
```

Arguments

The argument *attr* is one of a set of video display attributes, which are blinking, boldface, reverse video, and underlining, and are represented by the defined constants _BLINK, _BOLD, _REVERSE, and _UNDERLINE. You can set multiple attributes by separating them with a bitwise OR operator (|) as follows:

```
setattr(_BLINK | _UNDERLINE);
```

Additional Information

The macro and the function are VAX C specific and are not portable.

12.5.48 subwin

The **subwin** function creates a new subwindow with *numlines* lines and *numcols* columns starting at the coordinates *(begin_y, begin_x)* on the terminal screen.

The syntax of the function is as follows:

```
#include curses
#define bool int

WINDOW *subwin (WINDOW *win, int numlines, int numcols,
                        int begin_y, int begin_x);
```

Additional Information

When creating the subwindow, *begin_y* and *begin_x* are relative to the entire terminal screen. If either *numlines* or *numcols* is zero, then the function sets that dimension to (LINES – begin_y) or (COLS—begin_x) respectively.

A declared window must contain the entire area of the subwindow. Any changes made to either window within the coordinates of the subwindow appear on both windows.

12.5.49 [w]standend

The **standend** macro and the **wstandend** function deactivate the boldface attribute for the specified window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

standend()
int wstandend (WINDOW *win);
```

Additional Information

They are equivalent to **clrattr** and **wclrattr** called with the attribute _BOLD.

12.5.50 [w]standout

The **standout** macro and the **wstandout** function activate the boldface attribute of the specified window.

The syntax descriptions of the macro and function are as follows:

```
#include curses
#define bool int

standout()
int wstandout (WINDOW *win);
```

Additional Information

They are equivalent to **setattr** and **wsetattr** called with the attribute _BOLD.

12.5.51 touchwin

The **touchwin** function places the most recently edited version of the specified window on the terminal screen.

The syntax of the function is as follows:

```
#include curses
#define bool int

int touchwin (WINDOW *win);
```

Additional Information

The **touchwin** function usually is only needed to refresh overlapping windows.

12.5.52 wrapok

The **wrapok** macro, in the UNIX system environment, allows the wrapping of a word from the right border of the window to the beginning of the next line. This macro is provided only for UNIX software compatibility and has no functionality in the VMS environment.

The syntax of the macro is as follows:

```
#include curses
#define bool int

wrapok (WINDOW *win, bool boolf);
```

12.6 Program Examples

The following program examples show the effects of many of the Curses macros and functions. The **wgetch** and **wgetstr** functions appear throughout the programs so that the terminal screen may be viewed while the program waits for input. You can find explanations of the individual lines of code, if not self-explanatory, in the comments to the right of the particular line. Detailed discussions of the functions follow the source code listing.

Example 12–6 illustrates the definition and manipulation of one user-defined window and stdscr.

Example 12–6: Stdscr and Occluding Windows

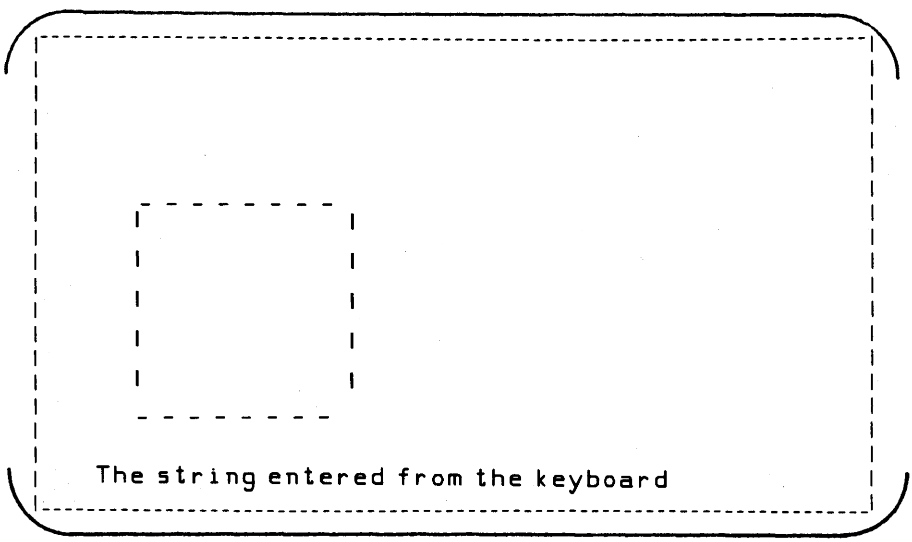
```
/*  The following program defines one window: WIN1.        *
 *  WIN1 is located towards the center of the default       *
 *  window stdscr.  When writing to an occluding window     *
 *  (WIN1) that is later erased, the writing is             *
 *  erased as well.                                         */

#include curses              /* Include module             */

WINDOW   *win1;              /* Define windows             */

main()
{
   char   str[80];           /* Variable declaration       */

   initscr();                /* Set up Curses              */
   noecho();                 /* Turn off echo              */
                             /* Create window              */
   win1 = newwin(10, 20, 10, 10);

   box(stdscr, '|', '-');    /* Draw a box around STDSCR   */
   box(win1, '|', '-');      /* Draw a box around WIN1     */

   refresh();                /* Display STDSCR on screen   */
   wrefresh(win1);           /* Display WIN1 on screen     */

❶  getstr(str);              /* Pause. Type a few words!   */

   mvaddstr(22, 1, str);
❷  getch();
                             /* Add string to WIN1         */
   mvwaddstr(win1, 5, 5, "Hello");
   wrefresh(win1);           /* Add WIN1 to terminal scr   */
   getch();                  /* Pause.  Press RETURN       */

   delwin(win1);             /* Delete WIN1                */
❸  touchwin(stdscr);         /* Refresh all of STDSCR      */

   getch();                  /* Pause.  Press RETURN       */
   endwin();                 /* Ends session.              */
}
```

The following numbers correspond to the numbers in the previous example:

❶ The program waits for input. Since the echo has been disabled using the **noecho** macro, the words that you type do not appear on stdscr. However, the macro stores the words in the variable str for use elsewhere in the program.

❷ The **getch** macro causes the program to pause. When you are finished viewing the screen, press the RETURN key so the program can resume. The **getch** macro refreshes stdscr on the terminal screen without calling **refresh**. The screen appears like Figure 12–4.

Figure 12–4: Example of the getch Macro



ZK-5751-86

❸ The **touchwin** function refreshes the screen so that all of stdscr is visible and the deleted occluding window no longer appears on the screen.

Example 12–7 illustrates **overlay**.

Example 12–7: Subwindows

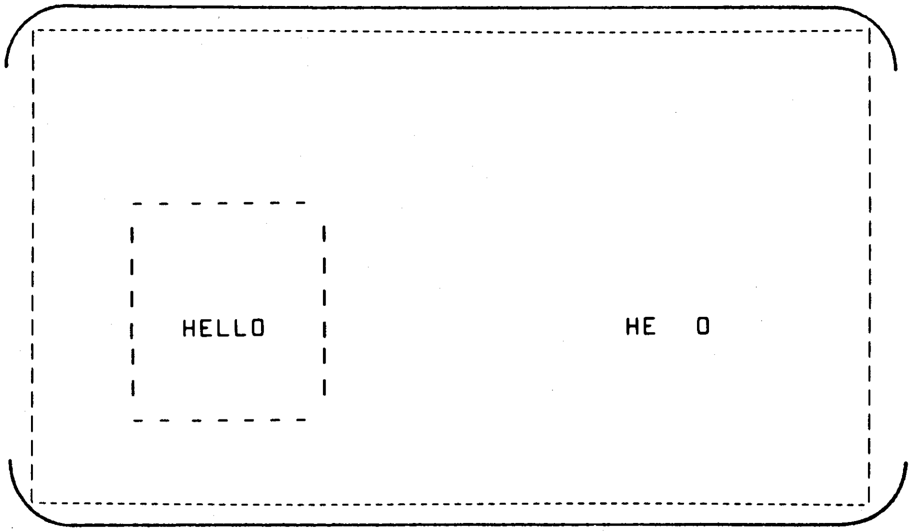
```
/*  The following program creates subwindows ---  WIN1    *
 *  and WIN2 --- and shows the effects of OVERLAY.         */

#include curses               /* Include module            */

WINDOW   *win1, *win2;        /* Define windows            */

main()
{
   initscr();                 /* Set up Curses             */
   noecho();                  /* Turn off echo             */

                              /* Create subwindows         */
   win1 = subwin(stdscr, 10, 20, 10, 10);
   win2 = subwin(stdscr, 10, 20, 10, 30);

   box(stdscr, '|', '-');     /* Draw a box round STDSCR   */
   box(win1, '|', '-');       /* Draw box round WIN1       */
   box(win2, '|','-');        /* Draw a box round WIN2     */

   mvwaddstr(win1, 5, 5, "  LL ");
❶  mvwaddstr(win2, 5, 5, "HE  O");

   overlay(win2, win1);       /* Lay WIN2 on WIN1          */
   wrefresh(win2);            /* Display WIN2 on screen    */

   delwin(win2);
   refresh();                 /* Refresh STDSCR            */
   wrefresh(win1);            /* Refresh WIN1              */

❷  getch();

   endwin();                  /* Ends session.             */
}
```

The following numbers correspond to the numbers in the previous example:

❶ Strings are added to the two subwindows. Anything written to the subwindows is also written to stdscr. These strings are added to the two subwindows at the same coordinates, (5, 5).

❷ The program pauses. When *win2* overlays win1, the word HELLO is formed. If *win2* were to overwrite *win1,* then the string HE O would appear instead of HELLO, the blanks overwriting the letters. The screen appears like that in Figure 12–5.

Figure 12–5: Example of Overwriting Windows



ZK-5750-86

Appendix A

VAX C RTL and RTLs of Other C Implementations

Most implementations of the C programming language provide, in one form or another, the run-time functions and macros listed in this appendix. Some of these functions are VAX C specific. Table A–1 describes possible differences between the VAX C RTL function or macro and other implementations of the functions or macros.

Table A–1: Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
abort	8.1	Not equivalent.
		VMS does not generate a core dump.
abs	7.1	Equivalent functionality.
access	2.6.1	Equivalent functionality.
acct		Not provided.
		Not provided in the VAX C Run-Time Library. The DCL command SET can be used to turn accounting on and off; the VMS system service, SYS$SNDACC, can be used to send messages to an accounting file.
acos	7.2	Equivalent functionality.
[w]addch	12.5.1	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
[w]addstr	12.5.2	Equivalent functionality.
alarm	8.7.1	Equivalent functionality.
asctime	11.4.1	Equivalent functionality.
asin	7.3	Equivalent functionality.
assert	8.2	Equivalent functionality.
atan	7.4	Equivalent functionality.
atan2	7.5	Equivalent functionality.
atexit	8.3	Defined in the ANSI C standard.
atof	6.7	Not equivalent. With VAX C, the string may contain any of the white-space characters (space, horizontal or vertical tab, carriage return, form feed, or newline).
atoi	6.9	See **atof**.
atol	6.9	See **atof**.
box	12.5.3	Equivalent functionality.
brk	9.1	See **sbrk**.
cabs	7.6	Equivalent functionality.
calloc	9.2	Equivalent functionality.
ceil	7.7	Equivalent functionality.
cfree	9.3	Equivalent functionality.
chdir	11.3.1	Not equivalent. The VAX C version changes the default directory for the user's program only. The user at a terminal will still have the same default directory as before the call. On VMS, use the DCL SET DEFAULT command.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
chmod	11.3.2	Not equivalent. VMS has no equivalent to the "set user id", "set group id" or "save text" file attributes. You can specify group and system read, write, and execute protection individually. **chmod** to 1000 ("save text") is done on VMS using the INSTALL utility.
chown	11.3.3	Equivalent functionality.
circle		Not provided.
[w]clear	12.5.4	Equivalent functionality.
clearerr	2.6.2	Equivalent functionality.
clearok	12.5.5	Equivalent functionality.
clock	11.4.2	Equivalent functionality.
close	4.1.1	Equivalent functionality.
closepl		Not provided.
[w]clrattr	12.5.6	VAX C specific.
[w]clrtobot	12.5.7	Equivalent functionality.
[w]clrtoeol	12.5.8	Equivalent functionality.
cont		Not provided.
cos	7.8	Equivalent functionality.
cosh	7.9	Equivalent functionality.
creat	4.1.2	Not equivalent. VAX C adds optional file attributes to allow the creation of files with RMS formats other than stream.
[no]crmode	12.5.9	Provided without functionality.
crypt		Not provided.
ctermid	11.2.1	Equivalent functionality.
ctime	11.4.3	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
cuserid	11.2.2	Equivalent functionality.
dbm		Not provided.
[w]delch	12.5.10	Equivalent functionality.
delete	2.6.7	VAX C specific.
[w]deleteln	12.5.11	Equivalent functionality.
delwin	12.5.12	Equivalent functionality.
difftime	11.4.4	Defined in the ANSI C standard.
div	7.15	Equivalent functionality.
dup	4.1.3	Equivalent functionality.
dup2	4.1.3	Equivalent functionality.
[no]echo	12.5.13	Equivalent functionality.
ecvt	5.2.1	Equivalent functionality.
endfsent		Not provided.
endgrent		Not provided.
endpwent		Not provided.
endwin	12.5.14	Equivalent functionality.
[w]erase	12.5.15	Equivalent functionality.
exec	10.2.1	See **execve**.
execl	10.2.1	See **execve**.
execlp	10.2.1	See **execve**.
execle	10.2.1	See **execve**.
execv	10.2.1	See **execve**.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
execve	10.2.1	Not equivalent. The principle of process overlaying is not used in VMS. On VAX C, you can **exec** programs only. When specifying the environment array, use the DCL syntax. The functions **execl** and **execle** contain separate character strings; the functions **execv** and **execve** contain arrays of character strings.
execvp	10.2.1	See **execve**.
exit	8.4	Not equivalent. If the process was invoked by the DCL command interpreter, then VMS interprets the return value and prints a DCL message.
exp	7.10	Equivalent functionality.
fabs	7.1	Equivalent functionality.
fclose	2.2.1	Equivalent functionality.
fcvt	5.2.1	Equivalent functionality.
fdopen	2.2.2	Equivalent functionality.
feof	2.6.3	Equivalent functionality.
ferror	2.6.4	Equivalent functionality.
fflush	2.5.1	Equivalent functionality.
fgetc	2.3.1	Equivalent functionality.
fgetname	2.6.5	Not equivalent. VAX C returns either the VMS file specification or the DEC/Shell file specification.
fgets	2.3.2	Equivalent functionality.
fileno	4.4.1	Equivalent functionality.
floor	7.11	Equivalent functionality.
fmod	7.12	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
fopen	2.2.3	Not equivalent. VAX C adds optional file attributes to allow the creation of files with RMS formats other than stream.
fork	10.1.2	Not provided (see **vfork**).
fprintf	2.4.1	Equivalent functionality.
fputc	2.4.4	Equivalent functionality.
fputs	2.4.2	Equivalent functionality.
fread	2.3.3	Equivalent functionality.
free	9.3	Equivalent functionality.
freopen	2.2.4	Not equivalent. VAX C adds optional file attributes to allow the creation of files with RMS formats other than stream.
frexp	7.13	Equivalent functionality.
fscanf	2.3.4	Not equivalent. VAX C provides the following conversion characters: hd, ho, hx, ld, lo, lx, le, lf, i, n, and p.
fseek	2.5.2	Not equivalent. When using record files, input from **ftell** is required for VAX C.
fstat	4.4.2	Equivalent functionality.
ftell	2.5.3	Not equivalent. When using record files, VAX C returns the position of the current record.
ftime	11.4.5	Equivalent functionality.
fwrite	2.4.3	Equivalent functionality.
gamma		Not provided.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
gcvt	5.2.1	Equivalent functionality.
getc	2.3.1	Equivalent functionality.
[w]getch	12.5.16	Equivalent functionality.
getchar	3.1	Equivalent functionality.
getcwd	11.2.3	Equivalent functionality.
getegid	11.2.4	See **getuid**.
getenv	11.2.5	Equivalent functionality.
geteuid	11.2.4	See **getuid**.
getfsent		Not provided.
getfsfile		Not provided.
getfsspec		Not provided.
getgid	11.2.4	See **getuid**.
getgrent		Not provided.
getgrgid		Not provided.
getgrnam		Not provided.
getlogin		Not provided.
getname	4.4.3	Not equivalent. VAX C returns either the VMS file specification or the DEC/Shell file specification.
getpass		Not provided.
getpgrp		Not provided.
getpid	11.2.6	Equivalent functionality.
getppid	11.2.7	Equivalent functionality.
getpw		Not provided.
getpwent		Not provided.
getpwnam		Not provided.
getpwuid		Not provided.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
getrgid		Not provided.
gets	3.2	Equivalent functionality.
[w]getstr	12.5.17	Equivalent functionality.
getuid	11.2.4	Not equivalent.
		VAX C returns the group and member codes from the UIC; VMS does not distinguish between real and effective user IDs.
getw	2.3.1	Equivalent functionality.
getyx	12.5.18	Equivalent functionality.
gmtime	11.4.6	Provided with no functionality.
gsignal	8.7.2	VAX C specific.
hypot	7.6	Equivalent functionality.
[w]inch	12.5.19	Equivalent functionality.
index		Not provided.
initscr	12.5.20	Equivalent functionality.
[w]insch	12.5.21	Equivalent functionality.
[w]insertln	12.5.22	Equivalent functionality.
[w]insstr	12.5.23	VAX C specific.
ioctl		Not provided.
isalnum	5.1.1	Equivalent functionality.
isalpha	5.1.2	Equivalent functionality.
isapipe	4.4.4	Equivalent functionality.
isascii	5.1.3	Equivalent functionality.
isatty	4.4.5	Equivalent functionality.
iscntrl	5.1.4	Equivalent functionality.
isdigit	5.1.5	Equivalent functionality.
isgraph	5.1.6	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
islower	5.1.7	Equivalent functionality.
isprint	5.1.8	Equivalent functionality.
ispunct	5.1.9	Equivalent functionality.
isspace	5.1.10	Equivalent functionality.
isupper	5.1.11	Equivalent functionality.
isxdigit	5.1.12	Equivalent functionality.
j0,j1,jn		Not provided.
kill	8.7.3	Not equivalent. VMS requires system privileges if the sending and receiving processes have different UICs. The receiving process ALWAYS terminates.
killpg		Not provided.
l3tol		Not provided.
label		Not provided.
ldexp	7.14	Equivalent functionality.
ldiv	7.15	Equivalent functionality.
leaveok	12.5.25	Equivalent functionality.
link		Not provided.
line		Not provided.
linemod		Not provided.
localtime	11.4.7	Not equivalent. On VAX C, daylight savings time always equals zero.
log,log10	7.17	Equivalent functionality.
longjmp	8.7.4	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
longname	12.5.24	Not equivalent. VAX C returns the terminal name, but to maintain portability, you must write a set of dummy routines to perform the same functionality as the database *termcap*.
lseek	4.3.1	Not equivalent. The VAX C function positions on record boundaries for RMS record files.
ltol3		Not provided.
malloc	9.2	Not equivalent. VAX C aligns the area returned on an octaword boundary.
memchr	6.11.1	Equivalent functionality.
memcmp	6.11.2	Equivalent functionality.
memcpy	6.11.3	Equivalent functionality.
memmove	6.11.3	Equivalent functionality.
memset	6.11.4	Equivalent functionality.
mkdir	11.3.4	Not equivalent. VAX C includes VMS specific optional arguments to specify UIC, maximum file version number, and the relative volume number.
mknod		Not provided.
mktemp	2.6.6	Equivalent functionality.
modf	7.18	Equivalent functionality.
monitor		Not provided.
mount,umount		Not provided.
[w]move	12.5.26	Equivalent functionality.
mpx		Not provided.
mv[w]addch	12.5.27	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
mv[w]addstr	12.5.28	Equivalent functionality.
mvcur	12.5.29	Equivalent to the function **move**.
mv[w]delch	12.5.30	Equivalent functionality.
mv[w]getch	12.5.31	Equivalent functionality.
mv[w]getstr	12.5.32	Equivalent functionality.
mv[w]inch	12.5.33	Equivalent functionality.
mv[w]insch	12.5.33	Equivalent functionality.
mv[w]insstr	12.5.35	VAX C specific.
mvwin	12.5.36	Equivalent functionality.
newwin	12.5.37	Equivalent functionality.
nice	11.3.5	Not equivalent. On VMS, the resulting priority cannot be greater than the process base priority.
[no]nl	12.5.38	Provided without functionality.
nlist		Not provided. This information can be obtained from the linker load map.
open	4.1.4	Not equivalent. VAX C requires mode = 2 when randomly writing to files.
openpl		Not provided.
overlay	12.5.39	Equivalent functionality.
overwrite	12.5.40	Equivalent functionality.
pause	8.7.5	Not equivalent. On VMS, processes can also be awakened with the SYS$WAKE system service.
pclose		Not provided.
perror	8.5	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
pipe	10.4.1	Not equivalent. VAX C specifies optional arguments for buffer size and asynchronous read operations.
point		Not provided.
popen		Not provided.
pow	7.19	Equivalent functionality.
printf	3.3	Equivalent functionality.
[w]printw	12.5.41	Equivalent functionality.
profil		Not provided.
ptrace		Not provided.
putc	2.4.4	Equivalent functionality.
putchar	3.4	Equivalent functionality.
puts	3.5	Equivalent functionality.
putw	2.4.4	Equivalent functionality.
qsort	11.1.2	Equivalent functionality.
raise	8.7.2	Defined in the ANSI C standard (equivalent to the **gsignal** function).
rand	7.20	Equivalent functionality.
[no]raw	12.5.42	Provided without functionality.
read	4.2.1	Equivalent functionality.
realloc	9.4	Not equivalent. On VAX C you can reallocate only the last freed area. For example, if you were to make two calls to **free**, only the second area could be reallocated.
reboot		Not provided.
[w]refresh	3	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
remove	2.6.7	Defined in the ANSI C standard (equivalent to the **delete** function).
rename	2.6.8	Equivalent functionality.
rewind	2.5.4	Equivalent functionality.
re_comp		Not provided.
re_exec		Not provided.
rindex		Not provided.
sbrk	9.1	Not equivalent. The VAX C version rounds the break address to the next higher multiple of 512 bytes.
scanf	3.6	Not equivalent. VAX C provides the following conversion characters: hd, ho, hx, ld, lo, lx, le, lf, i, n, and p.
[w]scanw	12.5.44	Equivalent functionality.
scroll	12.5.45	Equivalent functionality.
scrollok	12.5.46	Equivalent functionality.
[w]setattr	12.5.47	VAX C specific.
setbuf	2.6.9	Defined by ANSI C standard.
setgid	11.3.6	Provided without functionality.
setgrent		Not provided.
setjmp	8.7.4	Equivalent functionality.
setpgrp		Not provided.
setpwent		Not provided.
setsfent		Not provided.
setuid	11.3.6	Provided without functionality.
setvbuf	2.6.9	Not equivalent.
sigblock	8.7.6	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
sighold		Not provided. See VAX C **ssignal**, **gsignal** functions in Chapter 8, Error-Handling Functions.
sigignore		Not provided. See VAX C **ssignal**, **gsignal** functions in Chapter 8, Error-Handling Functions.
signal	8.7.7	Equivalent functionality.
sigpause	8.7.8	Equivalent functionality.
sigsetmask	8.7.9	Equivalent functionality.
sigstack	8.7.10	Equivalent functionality.
sigvec	8.7.11	Equivalent functionality.
sigrelse		Not provided. See VAX C **ssignal**, **gsignal** functions in Chapter 8, Error-Handling Functions.
sigset		Not provided. See VAX C **ssignal**, **gsignal** functions in Chapter 8, Error-Handling Functions.
sigsys		Not provided. See VAX C **ssignal**, **gsignal** functions in Chapter 8, Error-Handling Functions.
sin	7.21	Equivalent functionality.
sinh	7.22	Equivalent functionality.
sleep	8.7.12	Equivalent functionality.
space		Not provided.
sprintf	2.4.1	Equivalent functionality. VAX C also provides the conversion characters n and p. See the **fprintf** and **printf** functions for more information.
sqrt	7.23	Equivalent functionality.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
srand	7.20	Equivalent functionality.
sscanf	2.3.4	Not equivalent. VAX C provides the following conversion characters: h, ho, hx, ld, lo, lx, le, and lf.
ssignal	8.7.13	VAX C specific.
[w]standend	12.5.49	Equivalent functionality.
[w]standout	12.5.50	Equivalent functionality.
stat	4.4.2	Equivalent functionality.
stime		Not provided.
strcat	6.1	Equivalent functionality.
strchr	6.2	Equivalent functionality.
strcmp	6.3	Equivalent functionality.
strcpy	6.4	Equivalent functionality.
strcspn	6.5	Equivalent functionality.
strerror	8.6	Equivalent functionality.
strlen	6.6	Equivalent functionality.
strncat	6.1	Equivalent functionality.
strncmp	6.3	Equivalent functionality.
strncpy	6.4	Equivalent functionality.
strpbrk	6.5	Equivalent functionality.
strrchr	6.2	Equivalent functionality.
strspn	6.5	Equivalent functionality.
strtod	6.7	Equivalent functionality.
strtok	6.8	Equivalent functionality.
strtol	6.9	Equivalent functionality.
strtoul	6.10	Equivalent functionality.
subwin	12.5.48	Equivalent functionality.

Table A-1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
swab		Not provided.
sync		Not provided.
syscall		Not provided.
system	10.1.1	Equivalent functionality.
tan	7.24	Equivalent functionality.
tanh	7.25	Equivalent functionality.
tgetent		Not provided.
tgetflag		Not provided.
tgetnum		Not provided.
tgetstr		Not provided.
tgoto		Not provided.
time	11.4.8	Not equivalent. VAX C does not return timezone or daylight fields.
times	11.4.9	Not equivalent. VMS does not distinguish between system and user times. VAX C returns the time in 10-millisecond units.
timezone		Not provided.
tmpfile	2.6.10	Equivalent functionality.
tmpnam	2.6.11	Equivalent functionality.
toascii	5.2.2	Equivalent functionality.
tolower	5.2.3	Equivalent functionality.
touchwin	12.5.51	Equivalent functionality.
toupper	5.2.4	Equivalent functionality.
tputs		Not provided.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
ttyname	4.4.6	Not equivalent. VAX C returns a pointer to the null-terminated path name of the terminal device associated with file descriptor zero (standard input, stdin).
umask	11.3.7	Not equivalent. The default values of the **umask** function are set from RMS default file protection.
umount		Not provided.
ungetc	2.3.5	Equivalent functionality.
unlink		Not provided. This functionality is not provided in VMS. Temporary files can be created using the RMS extensions to **creat**. (See the **delete** and **remove** functions in 2.6.7.)
vadvise		Not provided.
valloc		Not provided.
va_arg	6.12.1	Equivalent functionality.
va_count	6.12.2	VAX C specific.
va_end	6.12.3	Equivalent functionality.
va_start	6.12.4	Equivalent functionality.
va_start_1	6.12.4	VAX C specific.
vfprintf	6.12.5	Equivalent functionality.
vfork	10.1.2	VAX C specific. This function is equivalent to the **fork** function in other implementations of the C language.
vhangup		Not provided.
vlimit		Not provided.
vprintf	6.12.5	Equivalent functionality.
vread		Not provided.

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

VAX C Function	Section Reference	Compared to Others
vsprintf	6.12.5	Equivalent functionality.
vswapon		Not provided.
vwrite		Not provided.
wait	10.3.1	Equivalent functionality.
wait3		Not provided.
wrapok	12.5.52	Provided without functionality.
write	4.2.2	Equivalent functionality.

Appendix B

VAX C Run-Time Modules and Entry Points

This appendix summarizes the modules and entry points in the VAX C run-time system. Table B–1 lists the modules in the library and describes their function. For an additional method of reference, Table B–2 lists the entry points defined in each module and describes their function. Table B–3 lists the modules from the VMS Run-Time Procedure Library that are called by VAX C run-time modules.

Table B–1: VAX C Run-Time Modules

Module	Description
C$$DOPRINT	Character-string print and scan routines.
C$$MAIN	Main start-off routine for C programs.
C$$MATH_HAND	Math routine condition handler.
C$$TRANSLATE	Translate VMS codes to UNIX codes.
C$ABORT	Abort the current process.
C$ABS	Integer absolute value math function.
C$ACOS	Arc cosine math function.
C$ADDSTR	Curses add string function.
C$ALARM	Set alarm function.
C$ASIN	Arc sine math function.
C$ASSERT	Run-time assertion function.
C$ATAN	Arc tangent math function.

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
C$ATAN2	Arc tangent math function.
C$ATEXIT	Declare exit handlers.
C$ATOF	ASCII to floating-point binary conversion.
C$ATOL	ASCII to integer binary conversion.
C$BOX	Curses create box function.
C$BREAK	Memory allocation routines.
C$BSEARCH	Binary chop search routine.
C$CEIL	Ceiling math function.
C$COS	Cosine math function.
C$COSH	Hyperbolic cosine math function.
C$CTERMID	Controlling terminal identification.
C$CTYPE	Character type data definitions.
C$CUSERID	User-name identification.
C$DATA	Data definitions of standard file structures.
C$DELWIN	Curses delete window function.
C$DIVIDE	**div** and **ldiv** math functions.
C$ECVT	Double float to ASCII string conversion.
C$ENDWIN	Terminate Curses session.
C$ERRNO	Run-time library error message definitions.
C$EXP	Base e exponentiation math function.
C$FABS	Floating-point double absolute math function.
C$FLOOR	Floor math library function.
C$FMOD	Floating-point remainder math function.
C$FREXP	Extract fraction and exponent math function.
C$FSTAT	Curses file status function.
C$GCVT	Double value to ASCII string conversion.
C$GETCWD	Get current working directory.
C$GETENV	Get environment value.
C$GETGID	Get group identification.

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
C$GETPID	Get the process identification.
C$GETPPID	Get the parent process identification.
C$GETSTR	Curses get string function.
C$GETUID	Get user identification.
C$HYPOT	Euclidean distance math library function.
C$INISIG	Initialize C RTL signal handler.
C$INITSCR	Begin Curses session.
C$INSSTR	Curses insert string function.
C$KILL	Terminate process.
C$LDEXP	Power of 2 math library function.
C$LOG	Logarithm base e math library function.
C$LOG10	Logarithm base 10 math library function.
C$LONGNAME	Retrieve terminal name.
C$MAIN	C main routines.
C$MALLOC	Memory allocation/deallocation.
C$MEMFUNC	**memchr**, **memcmp**, **memcpy**, **memmove**, and **memset** functions.
C$MODF	Extract fraction and integer math function.
C$MVWIN	Curses move window function.
C$NEWWIN	Curses create window function.
C$NICE	Set process priority.
C$OVERLAY	Curses window overlay function.
C$OVERWRITE	Curses window overwrite function.
C$PAUSE	Suspend the process until a signal is received.
C$PERROR	Print an error message.
C$POW	Power math library function.
C$PRINTW	Curses **printf** for window.
C$QSORT	Rapid sort function.
C$RAND	Random number generator.

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
C$RMS_PROTOTYPES	Definition of RMS data structures.
C$SCANW	Curses **scanf** for window.
C$SCROLL	Curses scroll window function.
C$SETGID	Set group identification.
C$SETJMP	Non-local goto functions (**setjmp/longjmp**).
C$SETUID	Set user identification.
C$SIGNAL	Manipulate signal database.
C$SIGVEC	Signal functionality.
C$SIN	Sine math function.
C$SINH	Hyperbolic sine math function.
C$SLEEP	Suspend the process for a number of seconds.
C$SQRT	Square root math function.
C$STAT	Get file status function.
C$STRCHR	Search for a character in a string.
C$STRCMP	Compare two strings.
C$STRERROR	Get RTL error message string.
C$STRFUNC	String manipulation functions.
C$STRINGS	Perform string manipulation.
C$STRNCMP	Compare two strings.
C$STRTOD	Convert string to a double.
C$STRTOK	Search for tokens in a string.
C$STRTOL	Convert string to a long or unsigned integer.
C$STRRCHR	Search for a character in a string.
C$SUBWIN	Curses create subwindow function.
C$TAN	Tangent math library function.
C$TANH	Hyperbolic tangent math function.
C$TIME	Get real-time values.
C$TIMEF	Manipulate/convert real-time values.
C$TMPFILE	Create a temporary file.

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
C$TMPNAM	Generate a name for a temporary file.
C$TOLOWER	Uppercase to lowercase conversion.
C$TOUCHWIN	Curses refresh window function.
C$TOUPPER	Lowercase to uppercase conversion.
C$TTYNAME	Get terminal name function.
C$UNIX	UNIX emulation routines.
C$VAXCIO	All I/O related functions.
C$WADDCH	Curses add character function.
C$WADDSTR	Curses add string function.
C$WCLEAR	Curses erase window function.
C$WCLRATTR	Curses stop attribute function.
C$WCLRTOBOT	Curses erase window to bottom function.
C$WCLRTOEOL	Curses erase window to the end of line function.
C$WDELCH	Curses delete character function.
C$WDELETELN	Curses delete line function.
C$WERASE	Curses erase window function.
C$WGETCH	Curses get character function.
C$WGETSTR	Curses get **stime** function.
C$WINCH	Curses insert character function.
C$WINSCH	Curses insert character function.
C$WINSERTLN	Curses insert line function.
C$WINSSTR	Curses insert string function.
C$WMOVE	Curses move cursor function.
C$WPRINTW	Curses **printf** for window.

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
C$WREFRESH	Curses refresh window function.
C$WSCANW	Curses **scanf** for window.
C$WSETATTR	Curses set attribute function.
C$WSTANDEND	Curses end bold function.
C$WSTANDOUT	Curses start bold function.
SHELL$CLINT	Interface shell argument lists.
SHELL$CLI_NAME	Determine user's CLI.
SHELL$FIX_TIME	UNIX system time formatting.
SHELL$FROM_VMS	DEC/Shell file translation.
SHELL$TO_VMS	DEC/Shell file translation.
SHELL$MATCH_WILD	Expand file name wild cards.
VAXC$ESTABLISH	Establish condition handler function.
VAXC$STACK_SWITCH	Switch to alternate signal stack.
VAXC$VARARGS	Variable argument list support.

Table B-2: VAX C Run-Time Entry Points

Entry Point	Module	Description
abort	C$ABORT	Abort the current process.
abs	C$ABS	Integer absolute value math library function.
access	C$VAXCIO	Check the accessibility of a file.
acos	C$ACOS	Arc cosine math library function.
addstr	C$ADDSTR	Add a string to stdcr.
alarm	C$ALARM	Set alarm library function.
asctime	C$TIMEF	Convert broken-down time into a character string.
asin	C$ASIN	Arc sine math library function.
assert	C$ASSERT	Provide diagnostic information.
atan	C$ATAN	Arc tangent math library function.
atan2	C$ATAN2	Arc tangent math library function.
atexit	C$ATEXIT	Register function(s) to be called without arguments at program termination.
atof	C$ATOF	Convert ASCII to floating-point binary.
atoi	C$ATOL	Convert ASCII to integer binary.
atol	C$ATOL	Convert long ASCII to binary.
box	C$BOX	Create a box surrounding a window.
brk	C$BREAK	Determine the low virtual address for program data area.
bsearch	C$BSEARCH	Binary chop search routine.

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
c$$cond_hand	C$$MAIN	Image condition handler.
c$$ctrlc_hand	C$$MAIN	Control/C ast handler.
c$$doprint	C$$DOPRINT	Internal output formatting routine.
c$$doscan	C$$DOSCAN	Internal input formatting routine.
c$$environ	C$UNIX	Establish **vfork** environment.
c$$exhandler	C$UNIX	Emulator exit handler.
c$$main	C$$MAIN	Main start-up routine.
c$$math_hand	C$$MATH_HAND	Math condition handler.
c$$translate	C$$TRANSLATE	Translate VMS error codes to UNIX error codes.
c$main	C$MAIN	Start up main program with no arguments.
c$main_args	C$MAIN	Start up main program with arguments.
cabs	C$HYPOT	Euclidean distance math library function.
calloc	C$MALLOC	Allocate and clear storage.
cc$rms_fab	C$RMS_PROTOTYPES	File access block prototype.
cc$rms_nam	C$RMS_PROTOTYPES	Name block prototype.
cc$rms_rab	C$RMS_PROTOTYPES	Record access block prototype.
cc$rms_xaball	C$RMS_PROTOTYPES	Allocation control extended attribute block prototype.
cc$rms_xabdat	C$RMS_PROTOTYPES	Date and time extended attribute block prototype.
cc$rms_xabfhc	C$RMS_PROTOTYPES	File header characteristics extended attribute block prototype.
cc$rms_xabkey	C$RMS_PROTOTYPES	Indexed file key extended attribute block prototype.

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
cc$rms_xabpro	C$RMS_PROTOTYPES	File protection extended attribute block.
cc$rms_xabrdt	C$RMS_PROTOTYPES	Revision date and time extended attribute block prototype.
cc$rms_xabsum	C$RMS_PROTOTYPES	Summary extended attribute block prototype.
cc$rms_xabtrm	C$RMS_PROTOTYPES	Terminal characteristics extended attribute block.
ceil	C$CEIL	Ceiling math library function.
cfree	C$MALLOC	Deallocate storage.
chdir	C$VAXCIO	Change the default directory.
chmod	C$VAXCIO	Change a file's access mode.
chown	C$VAXCIO	Change a file's owner.
clock	C$UNIX	Determine CPU time.
close	C$VAXCIO	Close a file.
cos	C$COS	Cosine math library function.
cosh	C$COSH	Hyperbolic cosine math library function.
creat	C$VAXCIO	Create a file.
ctermid	C$TERMID	Identify the controlling terminal.
ctime	C$TIMEF	Convert time to an ASCII string.
cuserid	C$CUSERID	Identify the user name.
delete	C$VAXCIO	Delete a file by file name.
delwin	C$DELWIN	Delete a window.
difftime	C$TIMEF	Compute the difference between two times.
div	C$DIVIDE	Compute quotient and remainder.

Table B-2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
dup	C$VAXCIO	Create a duplicate file descriptor.
dup2	C$VAXCIO	Create a duplicate file descriptor.
ecvt	C$ECVT	Convert a **double** value to ASCII.
endwin	C$ENDWIN	End Curses session.
execl	C$UNIX	Execute a program image.
execle	C$UNIX	Execute a program image.
execlp	C$UNIX	Execute a program image.
execv	C$UNIX	Execute a program image.
execve	C$UNIX	Execute a program image.
execvp	C$UNIX	Execute a program image.
exit	C$UNIX	Close files and exit.
_exit	C$UNIX	Exit image.
exp	C$EXP	Base e exponentiation math function.
fabs	C$FABS	**double** absolute math function.
fclose	C$VAXCIO	Close a file.
fcvt	C$ECVT	Convert a **double** value to ASCII.
fdopen	C$VAXCIO	Open a file by file descriptor.
fflush	C$VAXCIO	Flush a file buffer.
fgetc	C$VAXCIO	Get a character from a file.
fgetname	C$VAXCIO	Get a file-name string.
fgets	C$VAXCIO	Get a string from a file.
floor	C$FLOOR	Floor math library function.
fmod	C$FMOD	Compute the floating-point remainder of X/Y.
fopen	C$VAXCIO	Open a file by file pointer.

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
fprintf	C$VAXCIO	Format a string to a file.
fputc	C$VAXCIO	Write a character to a file.
fputs	C$VAXCIO	Write a string to a file.
fread	C$VAXCIO	Read from a file.
free	C$MALLOC	Deallocate storage.
freopen	C$VAXCIO	Close and reopen a file.
frexp	C$FREXP	Extract fraction exponent math function.
fscanf	C$VAXCIO	Scan input from a file.
fseek	C$VAXCIO	Position to an offset in a file.
fstat	C$FSTAT	Get file status function.
ftell	C$VAXCIO	Return current offset in a file.
ftime	C$TIME	Get the time.
fwrite	C$VAXCIO	Write to a file.
gcvt	C$GCVT	Convert a **double** value to ASCII.
getchar	C$VAXCIO	Get a character from standard input.
getcwd	C$GETCWD	Get the specification for the current working directory.
getegid	C$GETGID	Get the effective group identification.
getenv	C$GETENV	Get an environment value.
geteuid	C$GETUID	Get the effective user identification.
getgid	C$GETGID	Get the group identification.
getname	C$VAXCIO	Get a file-name string.
getpid	C$GETPID	Get the process identification.
getppid	C$GETPPID	Get the parent process id of the calling process.

Table B-2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
gets	C$VAXCIO	Get a string from standard input.
getstr	C$GETSTR	Get a string from stdscr.
getuid	C$GETUID	Get the user identification.
getw	C$VAXCIO	Get a longword from an input file.
gmtime	C$TIMEF	Convert calendar time into broken-down time.
gsignal	C$SIGNAL	Generate a signal.
hypot	C$HYPOT	Euclidean distance math library function.
initscr	C$INITSCR	Begin Curses session.
isatty	C$VAXCIO	Check for a terminal file.
isapipe	C$VAXCIO	Check for a mailbox.
insstr	C$INSSTR	Insert a string on stdscr.
kill	C$KILL	Send a signal to a process.
ldexp	C$LDEXP	Power of 2 math library function.
ldiv	C$DIVIDE	Compute long int quotient and remainder.
localtime	C$TIMEF	Place time in a time structure.
log	C$LOG	Logarithm base e math library function.
log10	C$LOG10	Logarithm base 10 math library function.
longjmp	C$SETJMP	Return to **setjmp**'s entry point.
longname	C$LONGNAME	Retrieve terminal name.
lseek	C$VAXCIO	Seek to a position in a file.
malloc	C$MALLOC	Allocate memory.
memchr	C$MEMFUNC	Locate first occurrence of a character.

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
memcmp	C$MEMFUNC	Compare lexical values of two arrays.
memcpy	C$MEMFUNC	Copy characters from one array to another.
memmove	C$MEMFUNC	Copy characters from one array to another.
memset	C$MEMFUNC	Put a given character in *n* bytes of an array.
mkdir	C$VAXCIO	Create a new directory.
mktemp	C$TMPNAM	Make a temporary file-name string.
modf	C$MODF	Extract fraction and integer math function.
mvwin	C$MVWIN	Move a window.
newwin	C$NEWWIN	Define a new window.
nice	C$NICE	Set process priority.
open	C$VAXCIO	Open a file by file descriptor.
overlay	C$OVERLAY	Place one window over another.
overwrite	C$OVERWRITE	Write one window onto another.
pause	C$PAUSE	Suspend the process.
perror	C$PERROR	Print an error message.
pipe	C$UNIX	Allow two processes to exchange data.
pow	C$POW	Power math library function.
printf	C$VAXCIO	Format a string to standard output.
printw	C$PRINTW	A **printf** to stdscr.
putchar	C$VAXCIO	Write a character to standard output.

Table B-2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
puts	C$VAXCIO	Write a string to standard output.
putw	C$VAXCIO	Write a longword to a file.
qsort	C$QSORT	Sort an array of data objects.
raise	C$SIGNAL	Generate a signal.
rand	C$RAND	Compute a random number.
read	C$VAXCIO	Read a file.
realloc	C$MALLOC	Change the size of an area of storage.
remove	C$VAXCIO	Delete a file.
rename	C$VAXCIO	Rename a file.
rewind	C$VAXCIO	Return to the beginning of the file.
sbrk	C$BREAK	Add bytes to the program's low virtual address.
scanf	C$VAXCIO	Format input from the standard input.
scanw	C$SCANW	A **scanf** to stdscr.
scroll	C$SCROLL	Scroll a window.
setbuf	C$VAXCIO	Associate a buffer with a file.
setgid	C$SETGID	Set group identification.
setjmp	C$SETJMP	Set up a return site for **longjmp**.
setuid	C$SETUID	Set user identification.
setvbuf	C$VAXCIO	Establish I/O buffering for a file.
shell$cli_name	SHELL$CLI_NAME	Determine user's command language interpretor.
shell$fix_time	SHELL$FIX_TIME	Translate time to a UNIX format.

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
shell$from_vms	SHELL$FROM_VMS	Translate VMS file specifications to DEC/Shell specs.
shell$get_argv	SHELL$CLINT	Interface to argument lists under the shell.
shell$is_shell	SHELL$CLI_NAME	Determine CLI name.
shell$match_wild	SHELL$MATCH_WILD	Wildcard expansion to infinite names.
shell$to_vms	SHELL$TO_VMS	Translate DEC/Shell file specifications to VMS specs.
shell$translate_vms	SHELL$TO_VMS	Translate DEC/Shell file specifications to DEC/Shell specs.
sigblock	C$SIGVEC	Block signals from delivery.
sigpause	C$SIGVEC	Pause and wait for a signal.
sigsetmask	C$SIGVEC	Block signals from delivery.
sigstack	C$SIGVEC	Define alternate signal stack.
siguec	C$SIGVEC	Assign a handler function for a specific signal.
signal	C$SIGNAL	Set a signal.
sin	C$SIN	Sine math library function.
sinh	C$SINH	Hyperbolic sine math library function.
sleep	C$SLEEP	Suspend the process.
sprintf	C$VAXCIO	Format a string to a memory buffer.
sqrt	C$SQRT	Square root math library function.
srand	C$RAND	Reinitialize the random number generator.
sscanf	C$VAXCIO	Format input from memory.
ssignal	C$SIGNAL	Set a signal.

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
stat	C$STAT	Get file status function.
strcat	C$STRINGS	Concatenate two strings.
strchr	C$STRCHR	Search for a character in a string.
strcmp	C$STRCMP	Compare two strings.
strcpy	C$STRINGS	Copy a string to another string.
strcspn	C$STRINGS	Search string for a character.
strerror	C$PERROR	Translate an error message code.
strlen	C$STRINGS	Determine the length of a string.
strncat	C$STRINGS	Concatenate two strings.
strncmp	C$STRNCMP	Compare two strings.
strncpy	C$STRINGS	Copy from one string to another.
strpbrk	C$STRINGS	Search a string for a character.
strrchr	C$STRRCHR	Search a string for a character.
strspn	C$STRSPN	Search a string for a character.
strtod	C$ATOF	Convert a string to a double precision number.
strtok	C$STRTOK	Locate text tokens in a given string.
strtol	C$STRTOL	Convert a character string into a long integer value.
strtoul	C$STRTOL	Convert a character string into an unsigned value.
subwin	C$SUBWIN	Create a subwindow.

Table B-2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
system	C$UNIX	Pass a string to a command processor for execution.
tan	C$TAN	Tangent math library function.
tanh	C$TANH	Hyperbolic tangent math library function.
time	C$TIME	Get the epoch time.
times	C$UNIX	Get the process and CPU times.
tmpfile	C$TMPFILE	Create a temporary file.
tmpnam	C$TMPNAM	Generate a temporary file name.
tolower	C$TOLOWER	Convert uppercase to lowercase.
touchwin	C$TOUCHWIN	View occluded window.
toupper	C$TOUPPER	Convert lowercase to uppercase.
ttyname	C$TTYNAME	Set a pointer to a device associated with a file.
umask	C$VAXCIO	Set a file's protection mask.
ungetc	C$VAXCIO	Push a character back into the stream.
utime	C$VAXCIO	Set the access and modification times for a file.
va_arg	VAXC$VARARGS	Returns the next argument.
va_count	VAXC$VARARGS	Count the number of arguments.
va_end	VAXC$VARARGS	Terminates the processing of variable argument lists.
va_start	VAXC$VARARGS	Initialize to the beginning of an argument list.
va_start_1	VAXC$VARARGS	Initialize to the beginning of an argument list.

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
vaxc$crtl_init	C$$MAIN	Initialize C RTL signal handlers for non-C programs.
vaxc$establish	VAXC$ESTABLISH	Establish a condition handler function.
vaxc$stack_switch	VAXC$STACK_SWITCH	Switch the stack for **sigstack** function.
vfork	C$UNIX	Spawn a process.
vfprintf	C$VAXCIO	Print formatted output.
vprintf	C$VAXCIO	Print formatted output.
vsprintf	C$VAXCIO	Print formatted output.
waddch	C$WADDCH	Add a character to a window.
waddstr	C$WADDSTR	Add a string to a window.
wait	C$VAXCIO	Suspend a process.
wclear	C$WCLEAR	Erase window.
wclrattr	C$WCLRATTR	Turn off a screen attribute.
wclrtobot	C$CLRTOBOT	Erase window to the bottom.
wclrtoeol	C$CWCLRTOEOL	Erase window to the end of current line.
wdelch	C$WDELCH	Delete a character from a window.
wdeleteln	C$DELETELN	Delete a line from a window.
werase	C$WERASE	Erase a window.
wgetch	C$WGETCH	Get a character from standard input; echo it on a window.
wgetstr	C$WGETSTR	Get a string from standard input; echo it on a window.
winch	C$WINCH	Return the character from a window at the cursor position.
winsch	C$WINSCH	Insert a character on a window.

Table B-2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
winsertln	C$WINSERTLN	Insert a blank line on a window.
winsstr	C$WINSSTR	Insert a string on a window.
wmove	C$WMOVE	Move the cursor position.
wprintw	C$WPRINTW	Perform a **printf** on a specified window.
wrefresh	C$WREFRESH	View edits made to a window.
write	C$VAXCIO	Write a file.
wscanw	C$WSCANW	Perform a **scanf** on a specified window.
wsetattr	C$WSETATTR	Turn on a screen attribute.
wstandend	C$WSTANDEND	Turn off boldface attribute.
wstandout	C$WSTANDOUT	Turn on boldface attribute.

Table B-3: Run-Time Library Procedures Called by VAX C

Procedure	Description
lib$get_foreign	Get DCL command line.
lib$free_vm	Virtual memory deallocation.
lib$get_vm	Virtual memory allocation.
lib$signal	Condition signaling.
lib$stop	Stop condition signal.
lib$spawn	Spawn a subprocess.
lib$establish	Establish an error handler.
lib$getsymbol	Translate DCL symbol.

The VAX C mathematical functions are performed by the VMS run-time procedures in the following list:

mth$dacos_r7	mth$dasin_r7	mth$datan_r7
mth$datan2	mth$dcos_r7	mth$dcosh
mth$dexp_r6	mth$dsqrt_r5	mth$dlog_r8
mth$dlog10_r8	mth$dsin_r7	mth$dsinh
mth$dsqrt_r5	mth$dtan_r7	mth$dtanh
mth$gacos_r7	mth$gasin_r7	mth$gatan_r7
mth$gatan2	mth$gcos_r7	mth$gcosh
mth$gexp_r6	mth$gsqrt_r5	mth$glog_r8
mth$glog10_r8	mth$gsin_r7	mth$gsinh
mth$gsqrt_r5	mth$gtan_r7	mth$gtanh

VAX C also calls run-time library modules that perform data conversion. The following list presents these modules:

ots$cvt_t_g
ots$cvt_t_d
ots$cvt_ti_l
ots$cvt_to_l
ots$cvt_tz_l
ots$$cvt_d_t_r8
ots$$cvt_g_t_r8
ots$powdd
ots$powgg

The following formatting routines are called by VAX C:

for$cvt_d_tg
for$cvt_d_te
for$cvt_d_tf
for$cvt_g_tg
for$cvt_g_te
for$cvt_g_tf

Appendix C

VAX C Definition Modules

This appendix lists the library definition modules contained in the text library named SYS$LIBRARY:VAXCDEF.TLB.

The contents of these modules can be examined in the appropriate definition file. All definition files have the file extension .H and they are contained in the directory SYS$LIBRARY. You can print or type individual files, or you can issue the following command to print all the files with their file names appearing at the top of each page:

```
$ PRINT  SYS$LIBRARY:*.H/HEADER
```

Table C–1 describes each of the definition modules:

Table C–1: VAX C Definition Modules

Module	Description
accdef	Accounting file record definitions
atrdef	File attribute definitions
chfdef	Structure definitions for condition handlers
climsgdef	Command language interpreter error code definitions
ctype	Character type and macro definitions for character classification functions
curses	Curses Screen Management related definitions
dcdef	Device class and type code definitions
descrip	Descriptor structure and constant definitions
devdef	Device characteristics definitions

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
dvidef	$GETDVI system service request code definitions
errno	Error number definitions
errnodef	VAX C error message constants
fab	File access block definitions
fchdef	File characteristics definitions
fibdef	File information block definitions
file	Symbol definitions for **open** function
float[1]	Macro definitions which provide implementation-specific floating-point restrictions
iodef	I/O function code definitions
jpidef	$GETJPI system service request code definitions
lckdef	Lock manager definitions
lkidef[1]	Lock information data identifier information.
libdef[1]	Definitions of LIB$ return codes
limits[1]	Macro definitions which provide implementation-specific constraints
lnmdef[1]	Logical name flag definitions
math	Math function definitions
msgdef	System mailbox message type definitions
nam	Name block definitions
nfbdef	DECNET file access definitions
opcdef	OPCOM request code definitions
perror	PERROR function related definitions
pqldef	Process quota code definitions
prcdef[1]	Create process (SYS$CREPRC) system service status flags
prdef	Processor register definitions
prvdef	Privilege mask bit definitions
psldef	Processor status longword definitions

[1]New definition modules.

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
rab	Record access block definitions
rms	All RMS structures and return status value definitions
rmsdef	RMS return status value definitions
secdef	Image section flag bit and match constant definitions
setjmp	State buffer definition for the **setjmp** and **longjmp** functions
sfdef	Stack call frame definitions
signal	Signal value definitions
smgdef	Curses Screen Management interface definitions
ssdef	System service return status value definitions
stat	STAT and FSTAT function related definitions
stdio	Standard I/O definitions
stsdef	System service status code format definitions
syidef[1]	Definitions for Get System-wide Information (SYS$GETSYI) system service
time	Definitions for the function **localtime**
timeb	Definitions for the function **ftime**
ttdef	Terminal definitions
tt2def	Terminal definitions
types	Type definitions
varargs	Variable argument list access definitions
xab	Extended attribute block definitions
xwdef[1]	System definitions for DECnet DDCMP

[1]New definition modules.

Table C–2 lists each of the modified definition modules and gives a description of the modification:

Table C–2: Modified Definition Modules

Modules	Description of Modification
atrdef	Constant identifier is in uppercase; structure tag is changed from ATTRIB to atrdef.
dcdef	Update incomplete symbol definitions.
dvidef	Update incomplete symbol definitions; constant identifier is in uppercase; structure tag is in lowercase.
fab	Update incomplete symbol definitions.
fchdef	Constant identifiers are in uppercase.
fibdef	Constant identifiers are in uppercase; update obsolete/incomplete symbol defintions.
iodef	Update obsolete/incomplete symbol definitions.
jpidef	Update incomplete symbol definitions.
lckdef	Update obsolete symbol definitions.
msgdef	Update incomplete symbol definitions.
nam	Update obsolete symbol definitions.
opcdef	Update incomplete symbol definitions.
prvdef	Update incomplete symbol definitions.
rmsdef	Update obsolete/incomplete symbol defintions.
smgdef	Update incomplete symbol definitions.
ttdef	Update incomplete symbol definitions.

Appendix D

Syntax Summary

This appendix describes the syntax of the VAX C Run-Time Library functions and macros.

abort Function

```
void abort(void)
```

Return Values:

None.

abs Function

```
#include math
```

```
int abs(int x);
double fabs(double x);
```

Return Values:

Absolute value of x.

access Function

```
#include stdio
```

```
int  access(char file_specification, int mode);
```

Return Values:

0 if the access (privilege) is allowed; –1 if not.

acos Function

```
#include math

double acos(double x);
```

Return Values:

Arc cosine of *x* in the range 0 to *pi*.

addch Macro and waddch Function

```
#include curses

addch(char ch);
waddch(WINDOW *win, char ch);
```

Return Values:

OK on success; ERR if illegal scrolling occurs.

addstr Macro and waddstr Function

```
#include curses

addstr(char *str);
waddstr(WINDOW *win, char *str);
```

Return Values:

OK on success; ERR if illegal scrolling occurs.

alarm Function

```
int  alarm(unsigned seconds);
```

Return Values:

The number of seconds remaining from a previous alarm request.

asctime Function

```
#include time

char *asctime (const time_t *timeptr);
```

Return Values:

Converts the contents of tm into a 26-character string.

asin Function

```
#include math
double asin(double x);
```

Return Values:

Arc sine of x in the range $-pi/2$ to $pi/2$. When $|x| > 1$, this function returns 0 and sets the variable, errno, to EDOM.

assert Macro

```
#include assert
void assert (int expression);
```

Return Values:

If *expression* is false, the **assert** macro displays information about the call that failed. If *expression* is true, the **assert** macro has no value.

atan Function

```
#include math
double atan(double x);
```

Return Values:

Arc tangent of x in the range $-pi/2$ to $pi/2$.

atan2 Function

```
#include math
double  atan2(double x, double y);
```

Return Values:

Arc tangent of x/y in the range $-pi$ to pi.

atexit Function

```
#include stdlib
int atexit (void (*func) (void));
```

Return Values:

A value that is not equal to zero if the registration succeeds.

atof, atoi, and atol Functions

```
#include math

double atof(char *nptr);
int atoi(char *nptr);
long atol(char *nptr);
```

Return Values:

Numeric value of the string as a **double** (**atof**), as an **int** (**atoi**), and as a **long int** (**atol**).

box Function

```
#include curses

box(WINDOW *win, char vert, char hor);
```

Return Values:

OK on success; ERR on failure.

brk and sbrk Functions

```
char  *brk(unsigned addr);
char  *sbrk(unsigned incr);
```

Return Values:

Lowest virtual address not used by the program (**brk**) and the old break address if the new break address is successfully set (**sbrk**). If the program requests too much memory, both return –1.

bsearch Function

```
#include stdlib
void *bsearch (const void *key,
               const void *base,
               size_t nmemb,
               size_t size,
               int (*compar (const void *, const void *));
```

Return Values:

A pointer to the matching member of the array or a NUL pointer if no match is found.

cabs and hypot Functions

```
#include math

double cabs(cabs_t z);
double hypot(double x, double y);
```

Return Values:

Square root of the sum of their two squared arguments.

calloc and malloc Functions

```
char *calloc(unsigned number, unsigned size);
char *malloc(unsigned size);
```

Return Values:

Pointer to the first byte of the allocated space; 0 if the memory cannot be allocated.

ceil Function

```
#include math

double ceil(double x);
```

Return Values:

Smallest integer that is greater than or equal to *x*.

cfree and free Functions

```
int cfree(void *pointer);
int free(void *pointer);
```

Return Values:

0 if the area previously allocated by **calloc**, **malloc**, or **realloc** is successfully freed; –1 if not.

chdir Function

```
int chdir(char *name);
```

Return Values:

0 if the directory is successfully changed; –1 if not.

chmod Function

```
int chmod(char *name, unsigned mode);
```

Return Values:

0 if the change is successful; –1 if not.

chown Function

```
int chown(char *name, unsigned owner, unsigned group);
```

Return Values:

0 if the owner UIC of the file is changed; –1 if not.

clear Macro and wclear Function

```
#include curses

clear()
wclear(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

clearerr Macro

```
#include stdio

clearerr(FILE *file_ptr);
```

Return Values:

None.

clearok Macro

```
#include curses

clearok(WINDOW *win, bool boolf);
```

Return Values:

OK on success; ERR on failure.

clock Function

```
#include time

clock_t clock (void);
```

Return Values:

The sum of the user and system times of the calling process and any terminated child processes for which the calling process has executed **wait** or **system**.

close Function

```
int  close(int file_desc);
```

Return Values:

0 if the file is successfully closed; –1 if the file descriptor is undefined or if an error occurs while the file is being closed.

clrattr Macro and wclrattr Function

```
#include curses

clrattr(int attr);
wclrattr(WINDOW *win, int attr);
```

Return Values:

OK on success; ERR on failure.

clrtobot Macro and wclrtobot Function

```
#include curses

clrtobot()
wclrtobot(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

clrtoeol Macro and wclrtoeol Function

```
#include curses

clrtoeol()
wclrtoeol(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

cos Function

```
#include math

double cos(double x);
```

Return Values:

Cosine of *x*.

cosh Function

```
#include math

double cosh(double x);
```

Return Values:

Hyperbolic cosine of *x*.

creat Function

```
int  creat(char *file_spec, unsigned mode [,char *file_atts, . . . ])
```

Return Values:

Integer file descriptor, if the file is successfully created; –1, if an error occurs.

[no]crmode Macros

```
#include curses

crmode()
nocrmode()
```

Return Values:

None.

Notes:

VAX C provides these macros only for portability; they have no functionality.

ctermid Function

```
#include stdio

char *ctermid([char *string]);
```

Return Values:

Name of the controlling terminal is returned to *string*. If no argument is given, the function returns the address of an internal storage area containing the string.

ctime Function

```
#include time

char  *ctime(int bintim);
```

Return Values:

Pointer to the time in the format: *wkd mmm dd hh:mm:ss* 19*yy*\n\0.

cuserid Function

```
#include stdio

char *cuserid(char *string);
```

Return Values:

The name of the user who initiated the current process is returned to *string*. If no argument is given, the function returns the address of the name.

delch Macro and wdelch Function

```
#include curses

delch()
wdelch(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

delete Function

```
#include stdio

int delete(char *file_spec);
```

Return Values:

0 if the file is deleted; –1 if not.

deleteln Macro and wdeleteln Function

```
#include curses

deleteln()
wdeleteln(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

delwin Function

```
#include curses

delwin(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

div and ldiv Functions

```
ldiv_t ldiv (long int numer, long int denom);
div_t div (int numer, int denom);
```

Return Values:

The quotient and remainder after the division of their arguments.

difftime Function

```
#include time

double difftime (time_t time1, time_t time2);
```

Return Values:

The difference in seconds expressed as a double.

dup and dup2 Functions

```
int  dup(int file_desc_1);
int  dup2(int file_desc_1, int file_desc_2);
```

Return Values:

A new file descriptor (**dup**) and a new descriptor pointing to the same file as *file_desc_1* (**dup2**); −1, if *file_desc_1* does not point to an open file or if the new file descriptor cannot be allocated.

[no]echo Macros

```
#include curses

echo()
noecho()
```

Return Values:

None.

ecvt, fcvt, and gcvt Functions

```
char  *ecvt(double value, int ndigit, int decpt, int sign);
char  *fcvt(double value, int ndigit, int decpt, int sign);
char  *gcvt(double value, int ndigit, char *buffer);
```

Return Values:

ecvt and **fcvt**

The position of the decimal point relative to the first character in the string is returned by the *decpt* argument. A nonzero integer is returned to the *sign* argument if the input value is negative; otherwise, 0 is returned.

gcvt

The converted string is placed in the *buf* argument and the address of *buf* is returned.

endwin Function

```
#include curses

int endwin(void);
```

Return Values:

OK on success; ERR on failure.

erase Macro and werase Function

```
#include curses

erase()
werase(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

execl, execle, execlp, execv, execve, and execvp Functions

```
int execl(char *file-spec, char *argn,...);
int execle(char *file-spec, char *argn,...);
int execlp(char *file-name, char *argn,...);
int execv(char *file-spec, char argv[]);
int execve(char *file-spec, char *argv[], char *envp[]);
int execvp(char *file_name, char *argv[]);
```

Return Values:

–1 on failure.

exit and _exit Functions

```
exit[int status];
_exit[int status];
```

Return Values:

The process status is returned to the parent process, if any, or to the command language interpreter.

exp Function

```
#include math
double exp(double x);
```

Return Values:

Base e raised to the power of the argument. If an overflow occurs, the function returns the largest possible floating-point value and sets the variable errno to ERANGE.

fclose Function

```
#include stdio
int fclose(FILE *file_ptr);
```

Return Values:

0 if the file is successfully closed; –1 if not.

fdopen Function

```
#include stdio
#include file

FILE *fdopen(int file_desc, char *a_mode);
```

feof Macro

```
#include stdio

int feof(FILE *file_ptr);
```

Return Values:

Nonzero integer on end-of-file; 0 otherwise.

ferror Macro

```
#include stdio

int ferror(FILE file_ptr);
```

Return Values:

A nonzero integer if an error occurs. Subsequent calls to **ferror** continue to return this value until the file is closed or until **clearerr** is called.

fflush Function

```
#include stdio

int fflush(FILE *file_ptr);
```

Return Values:

0 if the file is successfully flushed; EOF if not.

fgetc and getw Functions, and getc Macro

```
#include stdio

int fgetc(FILE file_ptr);
int getw(FILE file_ptr);
int getc(FILE file_ptr);
```

Return Values:

The next character in the file (**fgetc** and **getc**) and the next four characters from the file (**getw**); EOF on error.

fgetname Function

```
#include stdio
char *fgetname(FILE *file_ptr, char *buffer [,int style]);
```

Return Values:

Address of the buffer containing the file specification on success; 0 on error.

fgets Function

```
#include stdio
char *fgets(char *str, int maxchar, FILE file_ptr);
```

Return Values:

NULL on end-of-file; otherwise, the address of the first character in *string*.

fileno Macro

```
#include stdio
int fileno(FILE file_ptr);
```

Return Values:

Integer file descriptor that identifies the file.

floor Function

```
#include math
double floor(double x);
```

Return Values:

Largest integer that is less than or equal to x.

fmod Function

```
#include math
double fmod (double x, double y);
```

Return Values:

x if y is zero. Otherwise, it returns the value f, which has the same sign as x.

fopen Function

```
#include stdio

FILE *fopen(const char *file_spec, const char *a_mode,...);
```

Return Values:

A file pointer for the named file, if the file was successfully opened; NULL on error.

fprintf and sprintf Functions

```
#include stdio

int fprintf(FILE *file_ptr, char *format_spec [,output_src, . . . ]);
int sprintf(char *str, char *format_spec[,output_src, . . . ]);
```

Return Values:

The number of characters written to the file (**fprintf**) or to the string (**sprintf**); –1 on error.

fputc and putw Functions, and putc Macro

```
#include stdio

int fputc(char character, FILE *file_ptr);
int putw(int integer, FILE *file_ptr);
int putc(char character, FILE *file_ptr);
```

Return Values:

The argument *character* in the file (**fputc** and **putc**), and the four characters in *integer* (**getw**); EOF on error.

fputs Function

```
#include stdio

int fputs(char *str, FILE *file_ptr);
```

Return Values:

Last character written; EOF on error.

fread Function

```
#include stdio

int fread(void *pointer, int size_of_item, int number_items, FILE *file_ptr);
```

Return Values:

The number of items read; 0 on end-of-file or error.

freopen Function

```
#include stdio

FILE *freopen(char *file_spec, char *a_mode,
              FILE *file_ptr [,file_atts, . . . ]);
```

Return Values:

File pointer that points to the newly opened file; NULL on error.

frexp Function

```
#include math

double frexp(double value, int *eptr);
```

Return Values:

The mantissa of *value* with a magnitude less than 1. The exponent is returned to **eptr*.

fscanf and sscanf Functions

```
#include stdio

fscanf(FILE *file_ptr, char *format_spec [,input_ptr, . . . ])
sscanf(char *str, char *format_spec [,input_ptr . . . ])
```

Return Values:

The number of successfully matched and assigned input items; EOF on error or end-of-file.

fseek Function

```
#include stdio

int fseek(FILE *file_ptr, int offset, int direction);
```

Return Values:

0 for successful seeks; EOF on error.

fstat and stat Functions

```
#include stat

fstat(int file_desc, struct stat *buffer);
stat(char *file_spec, struct stat *buffer);
```

Return Values:

0 on success; –1 on error.

ftell Function

```
#include stdio

int ftell(FILE *file_ptr);
```

Return Values:

Byte offset from the beginning of the file to the current location within the file; –1 on error.

ftime Function

```
#include timeb

ftime (struct timeb *time_pointer);
```

Return Values:

The number of seconds that have elapsed on the system since 00:00:00 January 1, 1970 is returned to the structure, timeb.

fwrite Function

```
#include stdio

int fwrite(void *ptr, int size_of_item, int number_items, FILE *file_ptr);
```

Return Values:

The number of items written; 0 on error.

getch Macro and wgetch Function

```
#include curses

getch()
wgetch(WINDOW *win);
```

Return Values:

The character from the window; ERR if the screen scrolls illegally.

getchar Function

```
int getchar(void)
```

Return Values:

The next character from the standard input device (stdin, the terminal); EOF on error.

getegid, geteuid, getgid, and getuid Functions

```
unsigned  getegid(void)
unsigned  geteuid(void)
unsigned  getgid(void)
unsigned  getuid(void)
```

Return Values:

The group number from the UIC (**getgid** and **getegid**), or the member number from the UIC (**getuid** and **geteuid**).

getcwd Function

```
char *getcwd(char *buffer, unsigned int size,...);
```

Return Values:

A pointer to the file specification for the current working directory.

getenv Function

```
char *getenv(char *name);
```

Return Values:

The function returns one of the following values depending on the value of the argument, name, specified in the function call:

Argument	Return Value
HOME	The user's login directory
TERM	The terminal type
PATH	The default device and directory
USER	The name of the user initiating the process

getname Function

```
char *getname(int file_desc, char *buffer [,int style])
```

Return Values:

The address of the buffer containing the file specification; –1 on error.

getpid Function

```
int getpid(void)
```

Return Values:

The current process ID.

getppid Function

```
int getppid (void);
```

Return Values:

The parent process ID of the calling process.

gets Function

```
#include stdio
```

```
char *gets(char *str);
```

Return Values:

A pointer to the character string containing the line; NULL if end-of-file is reached before the newline is encountered or on error.

getstr Macro and wgetstr Function

```
#include curses
```

```
getstr(char *str);
wgetstr(WINDOW *win, char *str);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

getyx Macro

```
#include curses

getyx(WINDOW *win, int y, int x);
```

Return Values:

OK on success; ERR on failure.

gsignal Function

```
#include signal

int  gsignal(int sig [,code]);
```

Return Values:

If **gsignal** specifies a sig argument that is outside the range defined in the signal module, then **gsignal** returns zero, and the variable, errno, is set to EINVAL.

If **ssignal** establishes SIG_DFL (default action) for the signal, then **gsignal** does not return. The image is exited with the VMS error code that corresponds to the signal.

If **ssignal** establishes SIG_IGN (ignore signal) as the action for the signal, then **gsignal** returns its sig.

Otherwise, **ssignal** must have established an action function for the signal. That function is called, and that function's return value is returned by **gsignal**.

gmtime Function

```
#include time

struct tm *gmtime (const time_t *timer);
```

Return Values:

A NULL pointer because GMT is not available under VMS.

inch Macro and winch Function

```
#include curses

inch()
winch(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

initscr Function

```
#include curses

void initscr(void)
```

Return Values:

OK on success; ERR on failure.

insch Macro and winsch Function

```
#include curses

insch(ch)
winsch(WINDOW *win, char ch);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

insertln Macro and winsertln Function

```
#include curses

insertln()
winsertln(WINDOW *win);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

insstr Macro and winsstr Function

```
#include curses

insstr(char *str);
winsstr(WINDOW *win, char *str);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

Notes:

This function and macro are VAX C specific.

isalnum Macro

```
#include ctype
int isalnum(char character);
```

Return Values:

A nonzero integer, if the character is alphanumeric; 0 if it is not.

isalpha Macro

```
#include ctype
int isalpha(char character);
```

Return Values:

A nonzero integer, if the character is alphabetic; 0 if it is not.

isapipe Function

```
int isapipe(int file_desc);
```

Return Values:

1 if the specified file descriptor is associated with a mailbox; 0 if it is not; −1 on error.

isascii Macro

```
#include ctype
int isascii(char character);
```

Return Values:

A nonzero integer, if the character is ASCII; 0 if it is not.

isatty Function

```
int isatty(int file_desc);
```

Return Values:

1 if the file is a terminal; 0 if the file is not.

iscntrl Macro

```
#include ctype
int iscntrl(char character);
```

Return Values:

A nonzero integer, if the character is a control character; 0 if not.

isdigit Macro

```
#include ctype
int isdigit(char character);
```

Return Values:

A nonzero integer, if the character is a digit; 0 if it is not.

isgraph Macro

```
#include ctype
int isgraph(char character);
```

Return Values:

A nonzero integer, if the character is an ASCII graphic character; 0 if it is not.

islower Macro

```
#include ctype
int islower(char character);
```

Return Values:

A nonzero integer, if the character is lowercase; 0 if it is not.

isprint Macro

```
#include ctype
int isprint(char character);
```

Return Values:

A nonzero integer, if the character is an ASCII printing character; 0 if it is not.

ispunct Macro

```
#include ctype
int ispunct(char character);
```

Return Values:

A nonzero integer, if the character is a punctuation character; 0 if it is not.

isspace Macro

```
#include ctype
int isspace(char character);
```

Return Values:

A nonzero integer, if the character is one of the whitespace characters; 0 if it is not.

isupper Macro

```
#include ctype
int isupper(char character);
```

Return Values:

A nonzero integer, if the character is uppercase; 0 if it is not.

isxdigit Macro

```
#include ctype
int isxdigit(char character);
```

Return Values:

A nonzero integer, if the character is a hexadecimal digit; 0 if it is not.

kill Function

```
int kill(int pid, int sig);
```

Return Values:

0 if the signal is successfully queued; –1 if an error occurs.

ldexp Function

```
#include math

double ldexp(double x, double e);
```

Return Values:

The first argument times 2 to the power of its second argument ($x(2^e)$). If underflow occurs, this function returns 0. If overflow occurs, it returns the largest possible value of the appropriate sign.

leaveok Macro

```
#include curses

leaveok(WINDOW *win, bool boolf);
```

Return Values:

OK on success; ERR on failure.

localtime Function

```
#include time

struct tm *localtime(int bintim);
```

Return Values:

A pointer to the structure with the tag tm.

log and log10 Function

```
#include math

double log(double x);
double log10(double x);
```

Return Values:

The natural base-e (**log**) and the base-10 (**log10**) logarithm of its argument. If the argument, x, is 0 or negative, the function returns 0 and sets the variable errno to EDOM.

longjmp and setjmp Functions

```
#include setjmp

setjmp(jmp_buf env);
longjmp(jmp_buf env, int val);
```

Return Values:

The function **longjmp** returns *val* to the **setjmp** function with which it is associated.

When the function **setjmp** is first called, it returns 0. After **longjmp** is called with the same *env* argument as the first **setjmp** call, **setjmp** returns the value of the **longjmp** call's *val* argument.

longname Function

```
longname(char *termbuf, char *name);
```

Return Values:

The full terminal name is placed in the argument name.

lseek Function

```
int lseek(int file_desc, int offset, int direction);
```

Return Values:

The new position in the file. If the file descriptor is undefined or if you try to seek before the beginning of the file, the function returns –1.

memchr Function

```
#include string

int memchr (const void *s1, int c, size_t size);
```

Return Values:

A pointer to the first occurrence of the character. If the character does not occur, the **memchr** function returns a NUL pointer.

memcmp Function

```
#include string

int memcmp (const void *s1, const void *s2, size_t size);
```

Return Values:

An integer less than, equal to, or greater than 0 depending on whether the lexical value of the first array is less than, equal to, or greater than that of the second array.

memcpy Function

```
#include string

void *memcpy (void *s1, const void *s2, size_t size);
```

Return Values:

Value of *s1*.

memset Function

```
#include string

void *memset (void *s, char character, size_t size);
```

Return Values:

Value of *s*.

mkdir Function

```
int mkdir(char *dir_spec, unsigned mode [,unsigned uic,
[unsigned max_versions[,unsigned r_v_num]]]);
```

Return Values:

0 on success; –1 on error.

mktemp Function

```
#include stdio

char *mktemp(char *template);
```

Return Values:

A pointer to a temporary file name created from a *template* of the form "[*nam*]XXXXXX". If a unique file name cannot be created, **mktemp** returns a pointer to an empty string.

modf Function

```
#include math

double modf(double value, double *iptr);
```

Return Values:

The positive fractional part of *value* and the address of the integral part is assigned to *iptr*.

move Macro and wmove Function

```
#include curses

move(int y, int x);
wmove(WINDOW *win, int y, int x);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

mv[w]addch Macros

```
#include curses

mvaddch(int y, int x, char ch);
mvwaddch(WINDOW *win, int y, int x, char ch);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

mv[w]addstr Macros

```
#include curses

mvaddstr(int y, int x, char *str);
mvwaddstr(WINDOW *win, int y, int x, char *str);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

mvcur Function

```
#include curses

mvcur(int lasty, int lastx, int newy, int newx);
```

Return Values:

OK on success; ERR on failure.

mv[w]delch Macros

```
#include curses

mvdelch(int y, int x);
mvwdelch(WINDOW *win, int y, int x);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

mv[w]getch Macros

```
#include curses

mvgetch(int y, int x);
mvwgetch(WINDOW *win, int y, int x);
```

Return Values:

OK on success; ERR on failure.

mv[w]getstr Macros

```
#include curses

mvgetstr(int y, int x, char *str);
mvwgetstr(WINDOW *win, int y, int x, char *str);
```

Return Values:

OK on success; ERR on failure.

mv[w]inch Macros

```
#include curses

mvinch(int y, int x);
mvwinch(WINDOW *win, int y, int x);
```

Return Values:

OK on success; ERR on failure.

mv[w]insch Macros

```
#include curses

mvinsch(int y, int x, char ch);
mvwinsch(WINDOW *win, int y, int x, char ch);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

mv[w]insstr Macros

```
#include curses

mvinsstr(int y, int x, char *str);
mvwinsstr(WINDOW *win, int y, int x, char *str);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

mvwin Function

```
#include curses

mvwin(WINDOW *win, int y, int x);
```

Return Values:

OK on success; ERR, if moving the window puts all or part of the window off of the terminal screen. On error, this function does not attempt to move the window and the screen remains unaltered.

newwin Function

```
#include curses

newwin(int numlines, int numcols, int begin_y, int begin_x);
```

Return Values:

A pointer to a newly created window; ERR on failure.

nice Function

```
int  nice(int increment);
```

Return Values:

0 if the process priority is successfully lowered; –1 if it is not.

[no]nl Macros

```
#include curses

nl()
nonl()
```

Return Values:

None.

open Function

```
#include file
int open(char *file_spec, int flags, int mode [,file_attribute, . . . ]);
```

Return Values:

An integer file descriptor if the file is successfully opened; −1 if it is not.

overlay Function

```
#include curses
overlay(WINDOW *win1, WINDOW *win2);
```

Return Values:

OK on success; ERR on failure.

overwrite Function

```
#include curses
overwrite(WINDOW *win1, WINDOW *win2);
```

Return Values:

OK on success; ERR on failure.

pause Function

```
void pause(void)
```

Return Values:

None.

perror Function

```
#include perror
int perror(char *string);
```

Return Values:

A message to stdout (the terminal) of the form: *string: message*\n

pipe Function

```
#include file

int pipe(int array_fdscpt [, int flags[, int bufsize]]);
```

Return Values:

0 if the pipe is successfully created; –1 if not.

pow Function

```
#include math

double pow(double x, double y);
```

Return Values:

The argument x to the power of y.

If the result overflows, the function returns the largest possible floating-point value and sets the variable errno to ERANGE.

If y is negative or nonintegral, or if both arguments are 0, the function returns 0.

printf Function

```
#include stdio

int printf(char *format_spec [,output_src . . . ]);
```

Return Values:

The number of characters written; –1 on error.

[w]printw Functions

```
#include curses

printw(format_spec [,output_src,  . . . ])
wprintw(WINDOW *win, format_spec [,output_src,  . . . ]);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

putchar Function

```
int putchar(char character);
```

Return Values:

The character written; EOF on failure.

puts Function

```
#include stdio

int puts(char *str);
```

Return Values:

0 if the *string* was written to stdout (the terminal); EOF on failure.

qsort Function

```
#include stdlib

void qsort (void *base,
            size_t nmemb,
            size_t size,
            int (*compar) (const void *, const void *));
```

Return Values:

An integer less than, equal to, or greater than zero.

rand and srand Functions

```
int rand(void)
int srand(int seed);
```

Return Values:

Pseudorandom numbers in the range 0 to $2^{31} - 1$.

[no]raw Macros

```
#include curses

raw()
noraw()
```

Return Values:

None.

read Function

```
int read(int file_desc, char *buffer, int nbytes);
```

Return Values:

The number of bytes read; 0 if end-of-file is reached; –1 on error.

realloc Function

```
char *realloc(char *pointer, unsigned size);
```

Return Values:

The address of the area; 0 on error.

refresh Macro and wrefresh Function

```
#include curses

refresh()
wrefresh(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

remove Function

```
#include stdio

int remove (const char *file_spec);
```

Return Values:

A nonzero value if the operation fails.

rename Function

```
#include stdio

int rename (const char *old_file_spec, const char *new_file_spec);
```

Return Values:

A nonzero value if the operation fails.

rewind Function

```
#include stdio

int rewind(FILE *file_ptr);
```

Return Values:

0 if the file is successfully rewound; –1 if an error occurs.

scanf Function

```
#include stdio

int scanf(char *format_spec [,input_ptr . . . ]);
```

Return Values:

The number of successfully scanned items; EOF on end-of-file.

Notes:

For information concerning *input_ptr,* see **fscanf**.

scanw Macro and wscanw Function

```
#include curses

scanw(fmt_spec [,input_ptr,  . . . ])
wscanww(WINDOW *win, fmt_spec [,input_ptr,  . . . ]);
```

Return Values:

OK on success; ERR if the screen scrolls illegally.

scroll Function

```
#include curses

scroll(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

scrollok Macro

```
#include curses

scrollok(WINDOW *win, bool boolf);
```

Return Values:

OK on success; ERR on failure.

setattr Macro and wsetattr Function

```
#include curses

setattr(int attr)
wsetattr(WINDOW *win, int attr);
```

Return Values:

OK on success; ERR on failure.

setbuf and setvbuf Functions

```
#include stdio

void setbuf(FILE *file_ptr, char *buffer);

int setvbuf(FILE *file_ptr, char *buffer, int type, size_t size);
```

Return Values:

None.

setgid and setuid Functions

```
int setgid(unsigned group_number);
int setuid(unsigned member_number);
```

Return Values:

None.

sigblock Function

```
int sigblock(int mask);
```

Return Values:

The previous set of masked signals; –1 on error.

signal Function

```
#include signal

int (*signal(int sig, void (*func) (int,...))) (int,...);
```

Return Values:

The address of the function previously (or initially) established to handle the signal. If the argument sig is out of range, this function returns –1 and the variable errno is set to EINVAL.

sigpause Function

```
int sigpause(int mask);
```

Return Values:

After restoring the previous set of masked signals, this function returns EINTR which causes an interrupt; –1 on error.

sigsetmask Function

```
int sigsetmask(int mask);
```

Return Values:

The previous set of masked signals; –1 on error.

sigstack Function

```
#include signal
```

```
int sigstack(struct sigstack *ss, struct sigstack *oss);
```

Return Values:

0 on success; –1 on failure.

sigvec Function

```
#include signal
```

```
int sigvec(int sigint, struct sigvec *sv, struct sigvec *osv);
```

Return Values:

0 if the call to the signal handler is successful; –1 on error.

sin Function

```
#include math
```

```
double sin(double x);
```

Return Values:

The sine of *x*.

sinh Function

```
#include math

double sinh(double x);
```

Return Values:

The hyperbolic sine of its argument. Both *x* and its sine must be of type **double**. On overflow error, this function returns a **double** value with the largest possible magnitude and appropriate sign.

sleep Function

```
int  sleep(unsigned seconds);
```

Return Values:

The number of seconds that the process slept; –1 on error.

sqrt Function

```
#include math

double sqrt(double x);
```

Return Values:

The square root of *x*; 0 if *x* is negative.

ssignal Function

```
#include signal

int (*ssignal(int sig, void (*func)(int,...)
```

Return Values:

The address of the function previously (or initially) established as the action for the signal; 0 if the previous action was SIG_DFL.

standend Macro and wstandend Function

```
#include curses

standend()
wstandend(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

standout Function

```
#include curses

standout()
wstandout(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

strcat and strncat Function

```
char *strcat(char *str_1, char *str_2);
char *strncat(char *str_1, char *str_2, int maxchar);
```

Return Values:

The address of *str_1*.

strchr and strrchr Functions

```
char  *strchr(char *str, char character);
char  *strrchr(char *str, char character);
```

Return Values:

The address of the first occurrence (**strchr**) or the last occurrence (**strrchr**) of *character* in the string; 0 if the character was not found.

strcmp and strncmp Functions

```
int  strcmp(char *str_1, char *str_2);
int  strncmp(char *str_1, char *str_2, int maxchar);
```

Return Values:

A negative, 0, or positive integer indicating whether *str_1* is composed of more, equal, or less characters than *str_2*.

strcpy and strncpy Functions

```
char  *strcpy(char *str_1, char *str_2);
char  *strncpy(char *str_1, char *str_2, int maxchar);
```

Return Values:

The address of *str_1*.

strcspn Function

```
int strcspn(char *str, char *charset);
```

Return Values:

The number of characters preceding the first character in *str* that is also in *charset*. If no match is found, the function returns the length of *str*. This function returns 0 if *str* is NULL.

strerror Function

```
#include string
```

```
char *strerror(int errnum);
```

Return Values:

A pointer to a buffer that contains the appropriate error message.

strlen Function

```
int strlen(char *str);
```

Return Values:

The length of *str*.

strpbrk Function

```
char *strpbrk(char *str, char *charset);
```

Return Values:

The address of the first character in *str* that is also in *charset*; NULL if no match is found.

strspn Function

```
int strspn(char *str, char *charset);
```

Return Values:

The number of characters that precede the first character in *str* that is not also in *charset*. If *charset* is a NULL string, the function returns 0. If all the characters in *str* are also in *charset*, the function returns the length of *str*.

strtod Function

```
#include stdlib
double strtod (const char *nptr, char *endptr);
```

Return Values:

A double-precision value.

strtok Function

```
#include string
char *strtok (char *s1, const char *s2);
```

Return Values:

A pointer to the initial character in the first token. Subsequent calls return a pointer to a subsequent token.

strtol Function

```
#stdlib
long int strtol (const char *nptr, char *endptr, int base);
```

Return Values:

The converted value. If no conversion is performed, 0 is the returned value.

strtoul Function

```
#include stdlib
unsigned long int strtoul(const char *nptr, char *endptr, int base);
```

Return Values:

The converted value. If no conversion is performed, 0 is the returned value.

subwin Function

```
#include curses
WINDOW *subwin(WINDOW *win, int numlines, int numcols, int begin_y,
int begin_x);
```

Return Values:

A pointer to a newly created subwindow; ERR on failure.

system Function

```
#include processes

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

Return Values:

If the argument is a NUL pointer, the **system** function returns a nonzero value to indicate that the **system** function is supported.

tan Function

```
#include math

double tan(double x);
```

Return Values:

The tangent of x. At its singular points (. . . , $-3pi/2$, $-pi/2$, $pi/2$, . . .), the return value is the largest possible **double** value, and the variable errno is set to ERANGE

tanh Function

```
#include math

double tanh(double x);
```

Return Values:

The hyperbolic tangent of x.

time Function

```
long time(long *time_location);
```

Return Values:

The elapsed system time since 00:00:00 January 1, 1970. If the argument, time_location, is specified, it points to the location of the returned time.

times Function

```
void times(struct tbuffer *buffer);
```

Return Values:

The accumulated times of the current process and of its terminated child process. The times are placed in the user-defined structure with the tag tbuffer. The structure should have the following members of type **int**: *proc_user_time*, *proc_system_time*, *child_user_time*, and *child_system_time*. All system times are returned as 0. Accumulated CPU times are returned in 10-millisecond units.

tmpfile Function

```
#include stdio
```

```
FILE *tmpfile(void)
```

Return Values:

A file pointer to the temporary file; a NUL pointer on error.

tmpnam Function

```
#include stdio
```

```
char *tmpnam(char *name);
```

Return Values:

If *name* is specified, the function returns the file name string to *name*, or, if no argument is given, it returns the address of an internal storage area containing the string.

toascii Function

```
#include ctype
```

```
int toascii(char character);
```

Return Values:

The *character* converted to 7-bit ASCII.

tolower Function and _tolower Macro

```
#include ctype

char tolower(char character);
char _tolower(char character);
```

Return Values:

The *character* converted to lowercase. Lowercase input characters are returned unchanged.

touchwin Function

```
#include curses

int touchwin(WINDOW *win);
```

Return Values:

OK on success; ERR on failure.

ttyname Function

```
#include curses

int *ttyname()
```

Return Values:

A pointer to the NUL-terminated pathname of the terminal device associated with the file descriptor 0, stdin (the terminal).

toupper Function and _toupper Macro

```
#include ctype

char toupper(char character);
char _toupper(char character);
```

Return Values:

The *character* converted to uppercase. Uppercase input characters are returned unchanged.

umask Function

```
int umask(unsigned mode_complement);
```

Return Values:

The **chmod** argument mode corresponds to the argument mask except that mask has the effect of denying the specified privileges.

ungetc Function

```
#include stdio

int ungetc(char character, FILE *file_ptr);
```

Return Values:

The next character to be read (by **getc**); EOF on error.

va_arg Macro

```
#include varargs
va_arg(va_list list_incrementor, item_type);
```

Return Values:

The next argument in the argument list.

va_count Macro

```
#include varargs
va_count(int count);
```

Return Values:

The number of longwords in the argument list, in the argument count.

va_end Macro

```
#include varargs
va_end(va_list list_incrementor);
```

Return Values:

None.

va_start and va_start_1 Macros

```
#include varargs
va_start(va_list list_incrementor);
va_start_1(va_list list_incrementor, int offset);

function_name(va_alist)
va_dcl
{
   va_list  list_incrementor;
      .
      .
      .
```

Return Values:

These macros initialize the argument, list_incrementor, to the first argument in the list.

VAXC$ESTABLISH Function

```
void  VAXC$ESTABLISH (exception_handler)
int (*exception_handler)();
```

Return Values:

Establishes the *exception_handler* as a legitimate one. Only condition handlers declared in this way should be used in VAX C programs. In this way **VAXC$ESTABLISH** catches all RTL-related exceptions and passes on all others to the declared handler.

vfork Function

```
int vfork(void)
```

Return Values:

0 to the child process and the child process ID to the parent process.

vprintf, vfprintf, and vsprintf Functions

```
#include stdio
#include stdarg

int vprintf (const char *format, va_list arg);
int vfprintf (FILE *file_ptr, const char *format, va_list arg);
int vsprintf (char *s, const char *format, va_list arg);
```

Return Values:

The number of characters transmitted or a negative value if an output error occurs.

wait Function

```
int wait[int *status];
```

Return Values:

The process ID of the terminated child process; –1 if there are no child processes.

wrapok Macro

```
#include curses
wrapok(WINDOW *win, bool boolf);
```

Return Values:

None.

write Function

```
int write(int file_desc, char *buffer, char nbytes);
```

Return Values:

The number of bytes written; –1 on error.

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Vax C Run-Time Library
Reference Manual
AI–JP84A–TE

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____

____

____

____

____

____

Did you find errors in this manual? If so, specify the error and the page number.

____

____

____

____

____

____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________________ State ____________ Zip Code ________
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line

Vax C Run-Time Library
Reference Manual
AI–JP84A–TE

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____

____

____

____

____

____

Did you find errors in this manual? If so, specify the error and the page number.

____

____

____

____

____

____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________________ State ____________ Zip Code or Country ________

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line