VAX C Run-Time Library Reference Manual

Order Number: AA–JP84D–TE

February 1991

This manual describes the functions and macros in the VAX C Run-Time Library.

Revision/Update Information:	This revised manual supersedes the *VAX C Run-Time Library Reference Manual,* (Order Number AA–JP84C–TE).
Operating System and Version:	VMS Version 5.2 or higher
Software Version:	VAX C Version 3.2

digital equipment corporation
maynard, massachusetts

First Printing, May 1982
Revised, April 1985
Revised, March 1987
Revised, January 1989
Revised, December 1989
Revised, February 1991

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1982, 1985, 1987, 1989, 1991.

All Rights Reserved.
Printed in U.S.A.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC
DEC/CMS
DEC/MMS
DECnet
DECsystem–10
DECSYSTEM–20
DECUS
DECwriter
DIBOL
EduSystem
IAS
MASSBUS
PDP
PDT
RSTS
RSX
UNIBUS
VAX
VAXcluster
VMS
VT
digital™

UNIX is a registered trademark of UNIX System Laboratories, Inc.

ZK5629

Contents

Preface xvii

Chapter 1 VAX C Run-Time Library Information

1.1 Using the VAX C Run-Time Library 1–2
1.1.1 Using the VAX C RTL Object Libraries 1–2
1.1.2 Using the VAX C RTL as a Shareable Image 1–3
1.1.3 Macros 1–4

1.2 VAX C RTL Function and Macro Syntax 1–6
1.2.1 UNIX-Style File Specifications 1–8

1.3 Input and Output on VMS Systems 1–10
1.3.1 RMS Record and File Formats 1–13
1.3.2 Stream Access to RMS Record Files 1–15

1.4 Specific Portability Concerns 1–18

Chapter 2 Understanding Input and Output

2.1 UNIX I/O 2–5

2.2 Standard I/O 2–6

2.3 Conversion Specifications 2–7
2.3.1 Converting Input Information 2–8
2.3.2 Converting Output Information 2–11

2.4 Terminal I/O 2–13

2.5 Program Examples 2–14

Chapter 3 Character, String, and Argument List Functions and Macros

3.1 Character Classification Macros 3–4

3.2 Character Conversion Functions and Macros 3–9

3.3 String and Argument List-Handling Functions and Macros 3–11

3.4 Program Examples 3–12

Chapter 4 Error- and Signal-Handling

4.1 Error Handling 4–3

4.2 Signal Handling 4–5

4.3 Program Example 4–7

Chapter 5 Subprocess Functions

5.1 Implementing Child Processes in VAX C 5–2

5.2 The exec Functions 5–4
5.2.1 Exec Processing 5–4
5.2.2 Exec Error Conditions 5–5

5.3 Synchronizing Processes 5–6

5.4 Interprocess Communication 5–6

5.5 Program Examples 5–6

Chapter 6 Curses Screen Management Functions and Macros

6.1	**Curses Terminology**	6–5
	6.1.1 User-Defined Windows	6–6
6.2	**Getting Started with Curses**	6–9
6.3	**Predefined Variables and Constants**	6–12
6.4	**Cursor Movement**	6–14
6.5	**Program Examples**	6–15

Chapter 7 Math Functions

Chapter 8 Memory Allocation Functions

8.1	**Program Example**	8–3

Chapter 9 System Functions

Reference Section

ABORT	REF–3
ABS	REF–4
ACCESS	REF–5
ACOS	REF–7
[W]ADDCH	REF–8
[W]ADDSTR	REF–10
ALARM	REF–12
ASCTIME	REF–14
ASIN	REF–16
ASSERT	REF–17
ATAN	REF–19
ATAN2	REF–20
ATEXIT	REF–21
ATOF	REF–23
ATOI, ATOL	REF–24

BOX REF–25
BRK REF–27
BSEARCH REF–29
CABS REF–32
CALLOC REF–33
CEIL REF–34
CFREE REF–35
CHDIR REF–36
CHMOD REF–37
CHOWN REF–39
[W]CLEAR REF–40
CLEARERR REF–41
CLEAROK REF–42
CLOCK REF–43
CLOSE REF–44
[W]CLRATTR REF–46
[W]CLRTOBOT REF–48
[W]CLRTOEOL REF–49
COS REF–50
COSH REF–51
CREAT REF–52
[NO]CRMODE REF–57
CTERMID REF–59
CTIME REF–60
CUSERID REF–61
[W]DELCH REF–62
DELETE REF–63
[W]DELETELN REF–64
DELWIN REF–65
DIFFTIME REF–66
DIV REF–67
DUP, DUP2 . . . . . REF–68
[NO]ECHO REF–70
ECVT REF–71
ENDWIN REF–73
[W]ERASE REF–74
EXECL REF–75
EXECLE REF–77
EXECLP REF–80
EXECV REF–82
EXECVE REF–84
EXECVP REF–86
EXIT, _EXIT REF–88
EXP REF–89

FABS REF–90
FCLOSE REF–91
FCVT REF–92
FDOPEN REF–94
FEOF REF–96
FERROR REF–97
FFLUSH REF–98
FGETC REF–99
FGETNAME REF–100
FGETPOS REF–102
FGETS REF–104
FILENO REF–106
FLOOR REF–107
FMOD REF–108
FOPEN REF–109
FPRINTF REF–111
FPUTC REF–113
FPUTS REF–114
FREAD REF–115
FREE REF–117
FREOPEN REF–118
FREXP REF–120
FSCANF REF–122
FSEEK REF–124
FSETPOS REF–126
FSTAT REF–127
FTELL REF–130
FTIME REF–131
FWRITE REF–132
GCVT REF–134
GETC REF–136
[W]GETCH REF–137
GETCHAR REF–138
GETCWD REF–139
GETEGID REF–141
GETENV REF–142
GETEUID REF–144
GETGID REF–145
GETNAME REF–146
GETPID REF–148
GETPPID REF–149
GETS REF–150
[W]GETSTR REF–151
GETUID REF–152

GETW REF–153
GETYX REF–154
GMTIME REF–155
GSIGNAL REF–156
HYPOT REF–159
[W]INCH REF–160
INITSCR REF–161
[W]INSCH REF–162
[W]INSERTLN REF–163
[W]INSSTR REF–164
ISALNUM REF–165
ISALPHA REF–166
ISAPIPE REF–167
ISASCII REF–168
ISATTY REF–169
ISCNTRL REF–170
ISDIGIT REF–171
ISGRAPH REF–172
ISLOWER REF–173
ISPRINT REF–174
ISPUNCT REF–175
ISSPACE REF–176
ISUPPER REF–177
ISXDIGIT REF–178
KILL REF–179
LABS REF–181
LDEXP REF–182
LDIV REF–183
LEAVEOK REF–184
LOCALTIME REF–185
LOG, LOG10 . . . . REF–187
LONGJMP REF–188
LONGNAME REF–190
LSEEK REF–191
MALLOC REF–193
MEMCHR REF–194
MEMCMP REF–195
MEMCPY REF–197
MEMMOVE REF–198
MEMSET REF–200
MKDIR REF–201
MKTEMP REF–203
MODF REF–204
[W]MOVE REF–205

MVCUR REF–207
MVWIN REF–209
MV[W]ADDCH REF–211
MV[W]ADDSTR REF–212
MV[W]DELCH REF–213
MV[W]GETCH REF–214
MV[W]GETSTR REF–215
MV[W]INCH REF–216
MV[W]INSCH REF–217
MV[W]INSSTR REF–218
NEWWIN REF–219
NICE REF–221
[NO]NL REF–222
OPEN REF–223
OVERLAY REF–228
OVERWRITE REF–229
PAUSE REF–230
PERROR REF–231
PIPE REF–233
POW REF–238
PRINTF REF–240
[W]PRINTW REF–241
PUTC REF–243
PUTCHAR REF–244
PUTS REF–245
PUTW REF–246
QSORT REF–247
RAISE REF–249
RAND REF–252
[NO]RAW REF–253
READ REF–255
REALLOC REF–257
[W]REFRESH REF–259
REMOVE REF–260
RENAME REF–261
REWIND REF–262
SBRK REF–263
SCANF REF–264
[W]SCANW REF–265
SCROLL REF–267
SCROLLOK REF–268
[W]SETATTR REF–269
SETBUF REF–271
SETGID REF–272

SETJMP REF–273
SETUID REF–275
SETVBUF REF–276
SIGBLOCK REF–278
SIGNAL REF–279
SIGPAUSE REF–281
SIGSETMASK REF–282
SIGSTACK REF–283
SIGVEC REF–285
SIN REF–287
SINH REF–288
SLEEP REF–289
SPRINTF REF–290
SQRT REF–292
SRAND REF–293
SSCANF REF–294
SSIGNAL REF–296
[W]STANDEND REF–298
[W]STANDOUT REF–299
STAT REF–300
STRCAT REF–303
STRCHR REF–305
STRCMP REF–307
STRCPY REF–308
STRCSPN REF–309
STRERROR REF–310
STRLEN REF–312
STRNCAT REF–313
STRNCMP REF–314
STRNCPY REF–316
STRPBRK REF–318
STRRCHR REF–319
STRSPN REF–320
STRSTR REF–321
STRTOD REF–323
STRTOK REF–325
STRTOL REF–327
STRTOUL REF–329
SUBWIN REF–331
SYSTEM REF–333
TAN REF–335
TANH REF–336
TIME REF–337
TIMES REF–338

TMPFILE REF–339
TMPNAM REF–340
TOASCII REF–341
TOLOWER, _TOLOWER REF–342
TOUCHWIN REF–343
TOUPPER, _TOUPPER REF–344
TTYNAME REF–345
UMASK REF–346
UNGETC REF–347
VAXC$CALLOC_OPT REF–349
VAXC$CFREE_OPT REF–350
VAXC$CRTL_INIT REF–351
VAXC$ESTABLISH REF–352
VAXC$FREE_OPT REF–354
VAXC$MALLOC_OPT REF–355
VAXC$REALLOC_OPT REF–357
VA_ARG REF–359
VA_COUNT REF–361
VA_END REF–362
VA_START, VA_START_1 . . . . . . REF–363
VFORK REF–365
VFPRINTF REF–367
VPRINTF REF–369
VSPRINTF REF–371
WAIT REF–372
WRAPOK REF–373
WRITE REF–374

Appendix A VAX C RTL and RTLs of Other C Implementations

Appendix B VAX C Run-Time Modules and Entry Points

Appendix C VAX C Definition Modules

Appendix D VAX C Socket Routine Reference

D.1 Introduction D–1

D.2 Porting Considerations D–1
- D.2.1 Calling an IPC Routine from an AST State D–1
- D.2.2 Calling from KERNEL or EXEC Modes D–2
- D.2.3 Standard I/O D–2
- D.2.4 Event Flags D–2
- D.2.5 Suppressing VAX C Compilation Warnings D–2
- D.2.6 Header Files D–3

D.3 Linking an Internet Application Program D–3

D.4 VAX C Structures D–3
- D.4.1 hostent Structure D–3
- D.4.2 in_addr Structure D–4
- D.4.3 iovec Structure D–5
- D.4.4 linger Structure D–5
- D.4.5 msghdr Structure D–5
- D.4.6 netent Structure D–6
- D.4.7 sockaddr Structure D–7
- D.4.8 sockaddr_in Structure D–7
- D.4.9 timeval Structure D–8

D.5 Internet Protocols D–8
- D.5.1 Transmission Control Protocol D–8
- D.5.2 User Datagram Protocol D–9

D.6 errno Values D–9

D.7 Basic Communication Routines D–14
- ACCEPT D–15
- BIND D–17
- CLOSE D–19
- CONNECT D–20
- LISTEN D–22
- READ D–24
- RECV D–26
- RECVFROM D–28
- RECVMSG D–31
- SELECT D–34
- SEND D–37
- SENDMSG D–39

SENDTO . . . D–42
SHUTDOWN . . . D–45
SOCKET . . . D–47
WRITE . . . D–51

D.8 **Auxiliary Communication Routines** . . . D–53
GETPEERNAME . . . D–54
GETSOCKNAME . . . D–56
GETSOCKOPT . . . D–58
SETSOCKOPT . . . D–60

D.9 **Communication Support Routines** . . . D–63
GETHOSTBYADDR . . . D–65
GETHOSTBYNAME . . . D–67
GETHOSTENT . . . D–68
GETHOSTNAME . . . D–69
GETNETBYADDR . . . D–71
GETNETBYNAME . . . D–73
GETNETENT . . . D–75
HTONL . . . D–76
HTONS . . . D–78
INET_ADDR . . . D–80
INET_LNAOF . . . D–82
INET_MAKEADDR . . . D–83
INET_NETOF . . . D–84
INET_NETWORK . . . D–85
INET_NTOA . . . D–87
NTOHL . . . D–89
NTOHS . . . D–91
VAXC$GET_SDC . . . D–93

D.10 **Programming Examples** . . . D–94

Index

Examples

2–1 Output of the Conversion Specifications . . . 2–15
2–2 Using the Standard I/O Functions . . . 2–17
2–3 I/O Using File Descriptors and Pointers . . . 2–18
3–1 Character Classification Macros . . . 3–9
3–2 Converting Double Values to an ASCII String . . . 3–10
3–3 Changing Characters to and from Uppercase Letters . . . 3–11
3–4 Concatenating Two Strings . . . 3–13
3–5 Four Arguments to the **strcspn** Function . . . 3–14
3–6 The *varargs* Functions, Macros, and Definitions . . . 3–15
4–1 Suspending and Resuming Programs . . . 4–7
5–1 Creating the Child Process . . . 5–7
5–2 Passing Arguments to the Child Process . . . 5–9
5–3 Checking the Status of Child Processes . . . 5–11
5–4 Communicating Through a Pipe . . . 5–13
6–1 A Curses Program . . . 6–10
6–2 Manipulating Windows . . . 6–11
6–3 Refreshing the Terminal Screen . . . 6–12
6–4 Curses Predefined Variables . . . 6–13
6–5 The Cursor Movement Functions . . . 6–14
6–6 stdscr and Occluding Windows . . . 6–15
6–7 Subwindows . . . 6–18
7–1 Calculating and Verifying a Tangent Value . . . 7–4
8–1 Allocating and Deallocating Memory for Structures . . . 8–3
9–1 Accessing the User Name . . . 9–4
9–2 A Second Way to Access the User Name . . . 9–4
9–3 Accessing Terminal Information . . . 9–5
9–4 Manipulating the Default Directory . . . 9–5
9–5 Printing the Date and Time . . . 9–6
D–1 TCP/IP Server . . . D–94
D–2 TCP/IP Client . . . D–100
D–3 UDP Server . . . D–105
D–4 UDP Client . . . D–111

Figures

1–1 I/O Interface from C Programs . . . 1–11
1–2 Mapping Standard I/O and UNIX I/O to RMS . . . 1–13
5–1 Communications Links Between Parent and Child Processes . . . 5–3
6–1 An Example of the stdscr Window . . . 6–6
6–2 Displaying Windows and Subwindows . . . 6–8
6–3 Updating the Terminal Screen . . . 6–9
6–4 An Example of the **getch** Macro . . . 6–17
6–5 An Example of Overwriting Windows . . . 6–19
REF–1 Reading and Writing to a Pipe . . . REF–237

Tables

1–1 UNIX and VMS File Specification Delimiters . . . 1–8
1–2 Valid and Invalid Specifications . . . 1–9
2–1 I/O Functions and Macros . . . 2–1
2–2 Conversion Characters for Formatted Input . . . 2–9
2–3 Conversion Characters for Formatted Output . . . 2–11
2–4 Allowable Characters Between the Percent Sign and Conversion Character . . . 2–13
3–1 Character, String, and Argument List Functions and Macros . . . 3–1
3–2 Character Classification Macros and their Return Values . . . 3–5
3–3 Character Classification Macro Return Values (ASCII Table) . . . 3–5
4–1 Error- and Signal-Handling Functions and Macros . . . 4–1
4–2 The Errno Symbolic Values . . . 4–3
4–3 VAX C Signals . . . 4–6
5–1 Subprocess Functions . . . 5–2
6–1 Curses Functions and Macros . . . 6–2
6–2 Curses Predefined Variables and **#define** Constants . . . 6–12
7–1 Math Functions . . . 7–1
8–1 Memory Allocation Functions . . . 8–1
9–1 System Functions . . . 9–1
REF–1 Interpretation of the mode Argument . . . REF–5
REF–2 File Protection Values and their Meanings . . . REF–37
REF–3 RMS Valid Keywords and Values . . . REF–53
REF–4 SIGFPE Arithmetic Trap Signal Codes . . . REF–156

REF–5	Member Names	REF–185
REF–6	RMS Valid Keywords and Values	REF–225
REF–7	SIGFPE Signal Codes	REF–249
REF–8	The **vfork** and **fork** Functions	REF–365
A–1	Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros	A–1
B–1	VAX C Run-Time Modules	B–1
B–2	VAX C Run-Time Entry Points	B–6
B–3	Run-Time Library Procedures Called by VAX C	B–17
C–1	VAX C Definition Modules	C–1
D–1	errno Values	D–10
D–2	Basic Communication Routines	D–14
D–3	Auxiliary Communication Routines	D–53
D–4	Supported Communication Routines	D–63

Preface

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

It also describes the VAX C Socket routines used for writing Internet application programs for the VMS/ULTRIX Connection product.

Intended Audience

This manual is intended for experienced and novice programmers who need reference information on the functions and macros found in the VAX C RTL.

Document Structure

This manual describes the VAX C RTL. It provides information about portability concerns between operating systems and categorical descriptions of the functions and macros. This manual has nine chapters, a reference section, and five appendixes as follows:

- Chapter 1 provides an overview of the VAX C RTL.
- Chapter 2 discusses the Standard I/O, Terminal I/O, and UNIX I/O functions and macros. [1]
- Chapter 3 describes the character-, string-, and argument list-handling functions and macros.

[1] UNIX is a registered trademark of UNIX System Laboratories, Inc.

- Chapter 4 describes the error-, and signal-handling functions and macros.
- Chapter 5 explains the functions used to create subprocesses.
- Chapter 6 describes the Curses Screen Management functions and macros.
- Chapter 7 discusses the math functions and macros.
- Chapter 8 explains the memory allocation functions and macros.
- Chapter 9 describes the functions and macros used to interact with the operating system.
- The Reference Section describes all the functions and macros found in the VAX C RTL.
- Appendix A provides a comparison of VAX C RTL functions and macros, and corresponding functions of other C implementations.
- Appendix B provides a description of the VAX C modules and the VAX run-time modules used in this implementation.
- Appendix C lists the VAX C definition modules.
- Appendix D describes the VAX C Socket routines used for writing Internet application programs for the VMS/ULTRIX Connection product.

Associated Documents

The following documents may be 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 that cover the individual topics concerning access to the VMS operating system.

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.
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.
[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.

Convention	Meaning
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.

Chapter 1

VAX C Run-Time Library Information

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

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

These topics may seem unrelated, but a knowledge of all these issues is necessary to effectively use the VAX C RTL. This chapter shows the connections among these topics and the VAX C RTL. Read this chapter 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 7 describe the various tasks supported by the VAX C RTL. The Reference Section alphabetically lists and describes all the functions and macros available to perform these tasks.

1.1 Using the VAX C Run-Time Library

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

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

Second, the VAX C RTL functions are executed at run time, but references to these functions are resolved at link time. When you link your program, the VMS Linker (linker) resolves all references to VAX C RTL functions by searching any object code libraries or shareable code libraries specified on the LINK command line. 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.

1.1.1 Using the VAX C RTL Object Libraries

If you decide to link using the VAX C RTL object libraries, define the logical names LNK$LIBRARY, LNK$LIBRARY_1, and LNK$LIBRARY_2 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:

- If you do not need to use the Curses Screen Management (Curses) 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]
```

- If you 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]
```

- If you 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]
```

- Finally, if you 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 assuming that the program is in error. If the linker locates the function code, it places a copy of the code in the program's local program section (psect). 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 object libraries.

For more information about Curses, see Chapter 6. For more information about command procedures or the G_floating representation of **double** variables, see the *Guide to VAX C*.

1.1.2 Using the VAX C RTL as a Shareable Image

You can use the VAX C RTL as a shareable image instead of using the object code of the VAX C RTL functions. 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, using 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 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 due to decreased size.

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

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

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

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

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

1.1.3 Macros

You may need to use macros as well as functions from the VAX C RTL. Macros are resolved at compile time instead of at link time. The compiler replaces the macro reference with text found in a definition file. This process is called *macro expansion*. Macros are not the only segments of source code found in the definition files; these files can contain code fragments and definitions that are needed for some of the RTL functions to work properly.

Consequently, you need to learn about VAX C definition files to use the VAX C RTL wisely.

To understand definition files, you should know how the Standard I/O definitions are created. Definitions are composed of **#define** preprocessor directives. Traditionally in the C language, these **#define** directives are located in files with a .H file extension. If the VAX C software files are extracted during installation, 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, see your system manager about extracting the .H definition files from the text library.

Since it is often more efficient to access these files from the text library provided with VAX C, this manual refers to the .H definition files as definition modules. For more information about text libraries and modules, see the *Guide to VAX C*.

The following (nonstandard) 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 using the **#include** preprocessor directive in your source file. At compile time, the compiler replaces 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 *ctype* definition module, 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:

```
    .
    .
    .
a = _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 an uppercase letter or if it is not a letter, 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. The *Guide to VAX C* discusses the passing of arguments to macros.

The linker searches object libraries for the VAX C RTL function code, but the compiler searches text libraries or directories for the VAX C RTL macros. If you include definition modules in your source code, the compiler first searches the 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 is 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 include file searches, see the *Guide to VAX C*.

The .H definition files are distributed with VAX C, but the run-time libraries are distributed with the VMS operating system.

1.2 VAX C RTL Function and Macro Syntax

After learning how to link object modules and include definition modules, you must learn how to reference VAX C functions and macros in your program. The remaining chapters in this manual provide detailed descriptions of the VAX C RTL functions and macros.

In all chapters, the syntax describing each function and macro follows the standard convention for defining a function. This syntax is called a *function prototype* (or prototype). It is a compact representation of the order of a function's or macro's arguments (if any), the types of the arguments, and the type of the value returned by a 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 prototype descriptions provide insight into the functionality of the function or macro. These descriptions may not describe how to call the function or macro in your source code.

For example, consider the prototype for the **feof** function:

#include stdio
int feof(FILE *file_ptr;)

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

- 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 explicitly declare VAX C RTL macros yourself. This prototype merely indicates the arguments and the return value of **feof**.
- There is one argument, *file_ptr*, that is of type pointer to FILE. FILE is defined in the *stdio* module.

To use **feof** in a program, 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  */
          .
          .
          .
       }
}
```

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

- Optional parameters are enclosed in square brackets ([ ]).
- Use an ellipsis (. . .) 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(char **format_specification* [,*output_source*, . . .])

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 UNIX-Style 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.

NOTE

- The VAX C RTL 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.

Please note the differences between the UNIX system and VMS file specifications, as well as the method used by the RTL to access files. For example, the RTL accepts a valid VMS specification and most valid UNIX file specifications, but the RTL cannot accept a combination of both. Table 1–1 shows the differences between UNIX system and VMS system file specification delimiters.

Table 1–1: UNIX and VMS File Specification Delimiters

Description	VMS System	UNIX System
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 shown in Table 1–2.

Table 1–2: Valid and Invalid Specifications

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 how VAX C translates UNIX system file specifications and how 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 can 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 system equivalent of bin/today can be either [BIN]TODAY or [BIN.TODAY]. VAX C can make the correct interpretation only from the files present. If a file specification conforms to UNIX system file name syntax for a single file or directory, it is converted to the equivalent VMS file name if one of the following conditions is true:

- 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.
- 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.
- 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 I/O 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

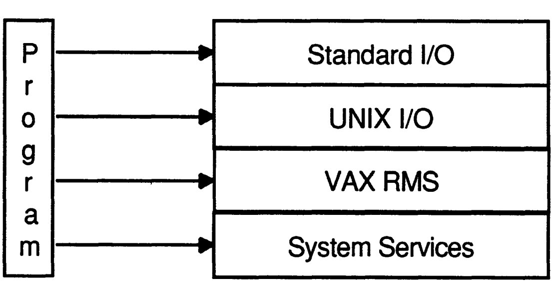
1.3 Input and Output on VMS Systems

After you learn how to link with the VAX C RTL, how to specify text libraries, and how to call VAX C functions and macros, you can use the VAX C RTL for its primary purpose: I/O.

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

Figure 1–1 shows the I/O methods available with VAX C. The VMS system services "talk" directly to the VMS operating system, so they are closest to the operating system. The VAX Record Management Services (RMS) functions use the system services, which manipulate the operating system. The VAX C Standard I/O and UNIX I/O functions and macros use the VAX RMS functions, which use the system services that 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



The C programming language was developed on the UNIX operating system, and the Standard I/O functions were designed to provide a convenient method of I/O that would be "powerful" enough to be efficient for most applications, and also 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. VAX C also implements some of the Standard I/O "functions" as preprocessor defined macros.

In a similar manner, the UNIX I/O functions originally were designed 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 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. VAX C also implements some of the UNIX I/O "functions" as preprocessor defined macros.

The VAX C RTL includes the Standard I/O functions that should 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 RMS to access files. To understand how the Standard and UNIX I/O functions manipulate VAX RMS formatted files, learn the fundamentals of VAX RMS. See Section 1.3.1 for more information about Standard and UNIX I/O in relationship to VAX RMS files. For an introduction to VAX RMS, see the *Guide to VMS File Applications*.

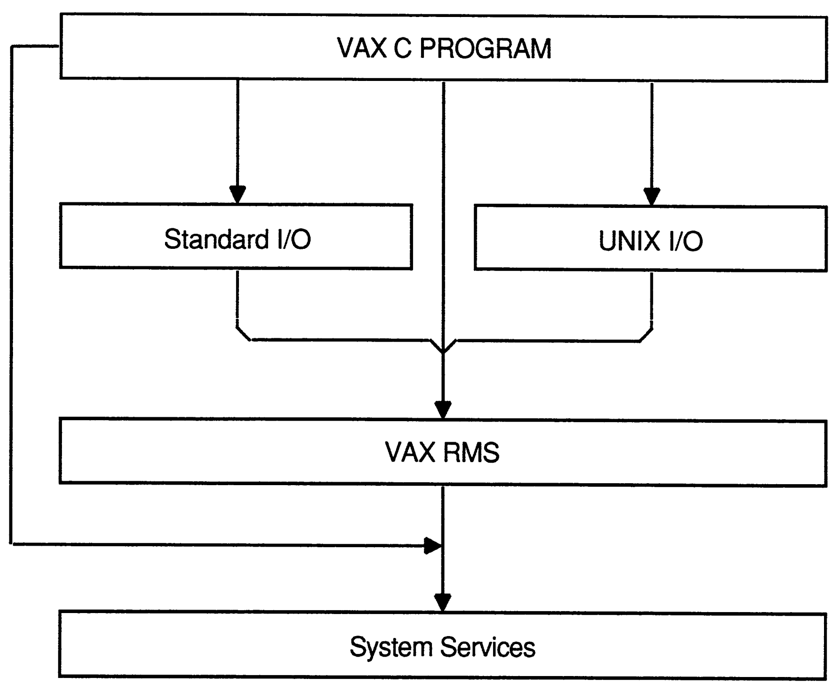
Before deciding which method is appropriate for you, first ask this question: Are you concerned with UNIX compatibility or with developing code that will run solely under the VMS operating system? If UNIX compatibility is important, you probably want to use the highest level of I/O—Standard I/O and UNIX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

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 the VMS operating system 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, use the VMS level of I/O; this level is recommended if you are an experienced VMS programmer. For examples of programs that call VMS system services, see the *Guide to VAX C*.

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

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



1.3.1 RMS Record and File Formats

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

VAX RMS supports the following file organizations:

- 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 about the RMS functions, see the *Guide to VAX C*.

VAX RMS is not concerned with the contents of records, but 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 the following 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 create 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 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 easily manipulated using the Standard I/O 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. Use the default VAX stream

format unless you need to create or access files to be used with other VAX languages or utilities.

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 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 using 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 how 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; and (\r) denotes the carriage-return character. Consider the following:

NULL	→ <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 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 your 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 RTL 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.

The VAX C RTL performs stream output to RMS record files 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, the VAX C RTL writes a record containing the data output so far and clears the buffer. If 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 to write 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 how 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; and (\r) denotes the carriage-return character. Consider the following:

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). The VAX C RTL then 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 you encounter when porting programs to and from VMS. Although portability is closely tied to the implementation of the VAX C RTL, 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 I/O and UNIX I/O functions and macros contained in the VAX C RTL are functionally equivalent to those of other implementations.

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 VAX C **delete** function replaces the **unlink** function found on implementations running on UNIX operating systems.

In other cases, VAX C includes functions or macros that provide no functionality under the VMS environment but are necessary so that you may port 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 a VMS system, 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. Appendixes A, B, and C provide information about porting C programs. Appendix A compares the functionality of VAX C RTL functions and macros with those of other implementations. Appendix B describes the run-time modules and entry points used by VAX C. Appendix C lists the .H definition files that are included in the compilation process to provide macro definitions and definitions used by some RTL functions. You may want to review the definitions contained within these files.

The following list documents issues of concern if you wish to port C programs to the VMS environment:

- VAX C does not implement the global symbols **end**, **edata**, and **etext**.

- Do not attempt to substitute your own code for functions that are supplied by VAX C. For example, the VAX C version of **strcpy** is expected to supply a legitimate return value. If you include a version of **strcpy** that does not return a value, the procedure will not perform correctly. For example:

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

  This definition of **strcpy** will not work because code inside the VAX C RTL expects, and makes use of, a return value.

- There are differences in how VMS and UNIX systems lay out virtual memory. In UNIX systems, the address space between 0 and the break address is accessible to your program. In VMS systems, the first page of memory is not accessible.

  For example, if a program tries to reference location 0 on a VMS system, a hardware error (ACCVIO) is returned and the program terminates abnormally. VMS systems reserve 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 you should modify them, as necessary.

- 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. See the *Guide to VAX C* for more information on built-in functions.

- 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 following initialization form:

```
int  foo  123;
```

  Programs using this form of initialization must 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, CC$g_float, and CC$parallel. These constants are useful for programs that must be compatible on other machines and operating systems. For more information, see the *Guide to VAX C*.
- The C language does not guarantee any memory order for the variables in a declaration. For example:

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

- The VMS Linker (linker) usually places VAX C **extern** variables in program sections (psects) of the same name as the variable. The linker then alphabetically links the psects by name. If you are porting a C program from another operating system to a VMS system, 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 ($) is a legal character in VAX C identifiers, and can be used as the first character.
- The C language does not define any order for evaluating expressions in function parameter lists or for many kinds of expressions. The way in which different C compilers evaluate an expression is only important when the expression has side effects. Consider the following examples:

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

x = func_y() + func_z();

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 **int** is 32 bits in VAX C. You will have to modify programs that are written for other machines and that assume a different size for a variable of type **int**. In addition, a variable of type **long** is the same size (32 bits) as a variable of type **int**.
- 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, unless **#pragma** member_alignment is specified. Other implementations may align structure members differently.
- References to structure members in VAX C cannot be vague. For more information, see the *Guide to VAX C*.

- Registers are allocated based upon how often a variable is used, but the keyword **register** gives the compiler a strong hint that you want to place a particular variable into a register. Whenever possible, the variable is placed into a register. You may allocate any scalar variable with the storage class **auto** or **register** to a register as long as the variable's address is not taken with the ampersand operator (&) and it is not a member of a structure or union.
- When moving programs from one operating system to another, you must consider the operations of the different linkers. 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 that you can avoid 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()
{
    .
    .
    .
}
```

To link the object code for the program and the module, either name the LABDATA object file in the LINK command, or explicitly extract 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 (for example, in a module that contains a function definition, a **globaldef** definition, or a **globalvalue** definition).

Chapter 2

Understanding Input and Output

There are three types of input and output (I/O) in the VAX C RTL: UNIX, Standard, and Terminal. Table 2–1 lists all the I/O functions and macros found in the VAX C RTL. For more detailed information on each function and macro, see the Reference Section.

Table 2–1: I/O Functions and Macros

Function or Macro	Purpose
UNIX I/O—Opening and Closing Files	
close	Closes the file associated with a file descriptor.
creat	Creates a new file.
dup,dup2	Allocates a new descriptor that refers to a file specified by a file descriptor returned by **open**, **creat**, or **pipe**.
open	Positions the file at its beginning.
UNIX I/O—Reading from Files	
read	Reads bytes from a file and places them in a buffer.
UNIX I/O—Writing to Files	
write	Writes a specified number of bytes from a buffer to a file.

(continued on next page)

Table 2–1 (Cont.): I/O Functions and Macros

Function or Macro	Purpose
UNIX I/O—Maneuvering in Files	
lseek	Positions a file to an arbitrary byte position and returns the new position as an **int**.
UNIX I/O—Additional Standard I/O Functions and Macros	
fileno	Returns an integer file descriptor that identifies the specified file.
fgetpos	Stores the current value of the file position indicator for the stream.
fsetpos	Sets the file position indicator for the stream according to the value of the object pointed to.
fstat, stat	Accesses information about the file descriptor or the file specification.
getname	Returns the file specification associated with a file descriptor.
isapipe	Returns 1 if the file descriptor is associated with a mailbox and 0 if it is not.
isatty	Returns 1 if the specified file descriptor is associated with a terminal and 0 if it is not.
ttyname	Returns a pointer to the NUL-terminated name of the terminal device associated with file descriptor 0, the default input device.
Standard I/O—Opening and Closing Files	
fclose	Closes a function by flushing any buffers associated with the file control block, and freeing the file control block and buffers previously associated with the file pointer.
fdopen	Associates a file pointer with a file descriptor returned by an **open**, **creat**, **dup**, **dup2**, or **pipe** function.
fopen	Opens a file by returning the address of a FILE structure.
freopen	Substitutes the file, named by a file specification, for the open file addressed by a file pointer.

(continued on next page)

Table 2–1 (Cont.): I/O Functions and Macros

Function or Macro	Purpose
Standard I/O—Reading from Files	
fgetc	Returns characters from a specified file.
getc	Returns characters from a specified file.
getw	Returns characters from a specified file.
fgets	Reads a line from a specified file and stores the string in an argument.
fread	Reads a specified number of items from a file.
fscanf	Performs formatted input from a specified file.
sscanf	Performs formatted input from a character string in memory.
ungetc	Pushes back a character into the input stream and leaves the stream positioned before the character.
Standard I/O—Writing to Files	
fprintf	Performs formatted output to a specified file.
fputs	Writes a character string to a file without copying the string's NUL terminator.
fwrite	Writes a specified number of items to a file.
fputc	Writes characters to a specified file.
putc	Writes characters to a specified file.
putw	Writes characters to a specified file.
sprintf	Performs formatted output to a string in memory.
Standard I/O—Maneuvering in Files	
fflush	Writes out any buffered information for the specified file.
fseek	Positions the file to the specified byte offset in the file.
ftell	Returns the current byte offset to the specified stream file.
rewind	Sets the file to its beginning.

(continued on next page)

Table 2–1 (Cont.): I/O Functions and Macros

Function or Macro	Purpose
Standard I/O—Additional Standard I/O Functions and Macros	
access	Checks a file to see whether a specified access mode is allowed.
clearerr	Resets the error and end-of-file indications for a file.
feof	Tests a file to see if the end-of-file has been reached.
ferror	Returns a nonzero integer if an error has occurred while reading or writing a file.
fgetname	Returns the file specification associated with a file pointer.
mktemp	Creates a unique file name from a template.
remove, delete	Causes a file to be deleted.
rename	Gives a new name to an existing file.
setbuf	Associates a buffer with an input or output file.
setvbuf	Associates a buffer with an input or output file.
tmpfile	Creates a temporary file that is opened for update.
tmpnam	Creates a character string that can be used in place of the file-name argument in other function calls.
Terminal I/O—Reading from Files	
getchar	Reads a single character from the standard input (stdin).
gets	Reads a line from the standard input (stdin).
scanf	Performs formatted input from the standard input.
Terminal I/O—Writing to Files	
printf	Performs formatted output from the standard output (stdout).
putchar	Writes a single character to the standard output and returns the character.
puts	Writes a character string to the standard output followed by a newline.

2.1 UNIX I/O

The UNIX I/O functions and macros access files with 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

See the description of **creat** and **open** in the Reference Section for information on the values to supply.

For more information about RMS, see 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 file access mode 0, carriage-return control, variable-length records, and it associates

the variable 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, see Chapter 1.

2.2 Standard I/O

In VAX C, and most other implementations of C, stream files and their associated functions form the Standard I/O facilities. *Stream files* are files that are 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 about RMS file organization, see the *Guide to VAX C*. For more information about the VAX C RTL and RMS file organization, see Chapter 1.)

Stream files in VAX C correspond to RMS stream files with the line-feed terminator attribute. To perform 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 you open the file for update, you can read (**fread**) and write (**fwrite**) at the current byte position in the file. File sharing is not supported for stream files.

The Standard I/O **fopen** 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**.

You can position a stream file 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 0; therefore, specifying 0 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 by using 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 C language implementations. Accessing files using the functions and macros provided as part of the VAX C RTL allows you to port file pointers.

2.3 Conversion Specifications

Several of the Standard I/O functions (including the Terminal I/O functions) use conversion characters to specify data formats for I/O. 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.

2.3.1 Converting Input Information

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

- White-space characters (spaces, tabs, and newlines), which match optional white-space characters in the input field.
- Ordinary characters (not %), which must match the next nonwhite-space character in the input.
- 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 2–2 describes the conversion characters for formatted input.

Table 2–2: Conversion Characters for Formatted Input

Character	Meaning
d	Expects a decimal integer in the input. The corresponding argument must point to an **int**.
o	Expects an octal integer in the input (with or without a leading 0). The corresponding argument must point to an **int**.
x	Expects a hexadecimal integer in the input (without a leading 0x). The corresponding argument must point to an **int**.
c	Expects 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	Expects 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	Expects 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]. The n's and d's are decimal digits (as many as indicated by the field width minus the signs and the letter E).
i	Expects an integer whose type is determined by the leading input characters. For example, a leading 0 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. (These are 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.

(continued on next page)

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

Character	Meaning
hd, ho, hx	Same as d, o, and x, except that a **short** integer of the specified radix is expected.
[. . .]	Expects a string that is not delimited by white-space characters. The brackets enclose a set of characters (not a string). This set (or "character class") is usually made up of the characters that comprise the string field. Any character not in the set terminates 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

- You can change the delimiters of the input field 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 and 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 the following form:

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

  You cannot use the following form:

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

- White space in a format specification matches optional white space in the input field. Consider the following format specification:

  ```
  field = %x
  ```

This format specification matches the following forms:

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

These forms do not match the following example:

```
fiel d=5218
```

2.3.2 Converting Output Information

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

- Ordinary characters, which are 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 2–3 describes the conversion characters for formatted output.

Table 2–3: Conversion Characters for Formatted Output

Character	Meaning
d	Converts to decimal format.
o	Converts to octal format.
X, x	Converts to unsigned hexadecimal format (without a leading 0x). An uppercase X causes the hexadecimal digits A to F to be printed in uppercase. A lowercase x causes those digits to be printed in lowercase.
u	Converts to unsigned decimal format (giving a number in the range 0 to 4,294,967,295).
c	Outputs a single character (NUL characters are ignored).
s	Writes characters until a NUL is encountered or until the number of characters indicated by the precision specification is exhausted. If the precision specification is 0 or omitted, all characters up to a NUL are output.

(continued on next page)

Table 2–3 (Cont.): Conversion Characters for Formatted Output

Character	Meaning
E, e	Converts **float** or **double** to the format [–]m.nnnnnnE[+ I –]xx. The number of n's is specified by the precision (the default is 6). If the precision is explicitly 0, 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	Converts **float** or **double** to the format [-]m..m.nnnnnn. The number of n's is specified by the precision (the default is 6). The precision does not determine the number of significant digits printed. If the precision is explicitly 0, no decimal point and n's appear.
G, g	Converts **float** or **double** to d, e, or f format, whichever is shorter (suppress insignificant zeros). If E format is used, an E is printed if the conversion character is an uppercase G, and an e is printed if the conversion character is a lowercase g.
%	Writes out the percent symbol. No conversion is performed.
p	Is implementation defined. Requires an argument to be a pointer to **void**.
n	Requires an argument to be a pointer to **void**. Causes the number of characters output to be written to the designated integers.
i	Requires an integer argument. Converts the argument to a signed decimal.

You can use the following characters between the percent sign (%) and the conversion character. They are optional, but if specified, they must occur in the order listed in Table 2–4.

Table 2–4: Allowable Characters Between the Percent Sign and Conversion Character

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. Pad with spaces or with zeros if the field width is specified with a leading 0; this does not mean that the width is an octal number. Padding is normally on the left and on the right if a minus sign is used.
. (period)	Separates the field width from precision.
precision	Use this integer constant to designate the maximum number of characters to print with an s format, or the number of fractional digits with an e or f format.
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 or the precision specification. The corresponding width or precision is given in the output source.
+	Requests that an explicit sign be present on a signed conversion.
#	Requests an alternate form conversion. Depending on the conversion specified, different actions will occur. For e, E, f, g, and G, the result contains a decimal point even at the end of an integer value. For g and G trailing zeros are not trimmed. For other conversions, the effect of # is undefined.

2.4 Terminal I/O

VAX C defines three file pointers that allow you to perform I/O to and from the logical devices usually associated with your terminal (for interactive jobs) or a batch stream (for batch jobs). In the VMS environment, the three permanent process files SYS$INPUT, SYS$OUTPUT, and SYS$ERROR perform the same functions for both interactive and batch jobs. 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 stdin file pointer is associated with the terminal to perform input. This file is equivalent to SYS$INPUT. The stdout file pointer is associated with the terminal to perform output. This file is equivalent to SYS$OUTPUT. The stderr file pointer is associated with the terminal to report run-time errors. This file is equivalent to SYS$ERROR.

There are three file descriptors 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.

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 since there are no arguments.

The VAX C RTL opens channels to SYS$INPUT, SYS$OUTPUT, and SYS$ERROR on the first execution of any VAX C I/O. If either of the process permanent files SYS$OUTPUT or SYS$ERROR is redirected to a file prior to this, a new, null version of the file is created when the I/O is executed. To avoid this problem, force the mapping to a process permanent file yourself. For example:

```
$  OPEN/WRITE SYSERR ERROR.FILE
$  ASSIGN SYSERR SYS$ERROR
$  RUN MYAPPLICATION
$  DEASSIGN SYS$ERROR
$  CLOSE SYSERR
```

This eliminates duplicate file generation.

2.5 Program Examples

Example 2–1 shows the **printf** function.

Example 2–1: Output of the Conversion Specifications

```
/*  This program uses the printf function to print the          *
 *  various conversion specifications and their effect on the   *
 *  output.                                                     */
                               /*  Include the 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);

}
```

The sample output from Example 2–1 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:        <      >

$
```

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

Example 2–2: 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             */
}
```

The sample output, to the terminal, from Example 2–2 is as follows:

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

The sample output, to DATA.DAT, from Example 2–2 is as follows:

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

Example 2–3 shows the use of both a file pointer and a file descriptor to access a single file.

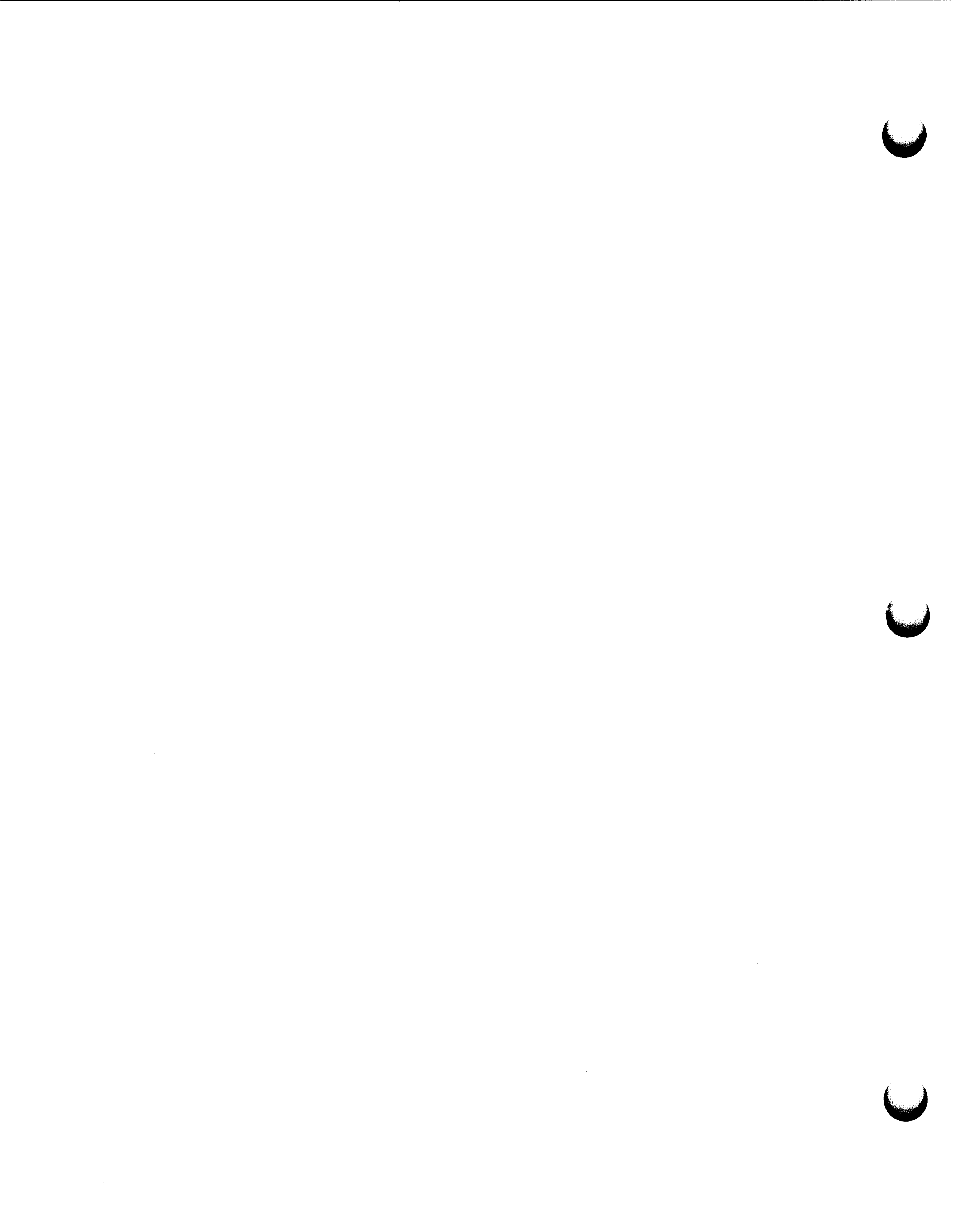
Example 2–3: I/O Using File Descriptors and Pointers

```
/*  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);
         }
```

(continued on next page)

Example 2–3 (Cont.): I/O Using File Descriptors and Pointers

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



Chapter 3

Character, String, and Argument List Functions and Macros

This chapter discusses the character, string, and argument list functions and macros. Table 3–1 lists and describes all the character, string, and argument list functions and macros found in the VAX C RTL. For more detailed information on each function and macro, see the Reference Section.

Table 3–1: Character, String, and Argument List Functions and Macros

Function or Macro	Purpose
Character Classification	
isalnum	Returns a nonzero integer if its argument is one of the alphanumeric ASCII characters.
isalpha	Returns a nonzero integer if its argument is one of the alphabetic ASCII characters.
isascii	Returns a nonzero integer if its argument is any ASCII character.
iscntrl	Returns a nonzero integer if its argument is an ASCII DEL character (177 octal) or any nonprinting ASCII character (code less than 40 octal).
isdigit	Returns a nonzero integer if its argument is a decimal digit character (0 to 9).
isgraph	Returns a nonzero integer if its argument is a graphic ASCII character.

(continued on next page)

Table 3–1 (Cont.): Character, String, and Argument List Functions and Macros

Function or Macro	Purpose
Character Classification	
islower	Returns a nonzero integer if its argument is a lowercase alphabetic ASCII character.
isprint	Returns a nonzero integer if its argument is an ASCII printing character (ASCII codes from 40 octal to 176 octal).
ispunct	Returns a nonzero integer if its argument is an ASCII punctuation character.
isspace	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.
isupper	Returns a nonzero integer if its argument is an uppercase alphabetic ASCII character.
isxdigit	Returns a nonzero integer if its argument is a hexadecimal digit (0 to 9, A to F, or a to f).
Character Conversion	
ecvt	Converts its argument to a NUL-terminated string of ASCII digits and returns the address of the string.
fcvt	Converts its argument to a NUL-terminated string of ASCII digits and returns the address of the string.
gcvt	Converts its argument to a NUL-terminated string of ASCII digits and returns the address of the string.
toascii	Converts its argument, an 8-bit ASCII character, to a 7-bit ASCII character.
tolower, _tolower	Convert their argument, an ASCII character, to lowercase.
toupper, _toupper	Convert their argument, an ASCII character, to uppercase.

(continued on next page)

Table 3–1 (Cont.): Character, String, and Argument List Functions and Macros

Function or Macro	Purpose
String Manipulation	
atof	Converts a given string to a double-precision number.
atoi	Converts a given string of ASCII characters to the appropriate numeric values.
atol	Converts a given string of ASCII characters to the appropriate numeric values.
strcat, strncat	Concatenate the arguments of one string to the end of another string.
strchr, strrchr	Return, respectively, the address of the first or last occurrence of a given character in a NUL-terminated string.
strcmp, strncmp	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.
strcpy, strncpy	Copy all or part of one string into another.
strcspn	Searches a string for a character in a specified set of characters.
strlen	Returns the length of a string of ASCII characters. The returned length does not include the terminating NUL character (\0).
strpbrk	Searches a string for the occurrence of one of a specified set of characters.
strspn	Searches a string for the occurrence of a character that is not in a specified set of characters.
strtol	Converts a string of ASCII characters to the appropriate numeric values.
strtod	Converts a given string to a double-precision number.
strtok	Locates text tokens in a given string.
strtoul	Converts the initial portion of the string pointed to by the pointer to the character string to an unsigned long integer.

(continued on next page)

Table 3–1 (Cont.): Character, String, and Argument List Functions and Macros

Function or Macro	Purpose
String Handling—Accessing Binary Data	
memchr	Locates the first occurrence of the specified byte within the initial length of the object to be searched.
memcmp	Compares two objects byte by byte.
memcpy	Copies a specified number of bytes from one object to another.
memmove	Copies a specified number of bytes from one object to another.
memset	Sets a specified number of bytes in a given object to a given value.
Argument-List Handling—Accessing a Variable-Length Argument List	
va_arg	Returns the next item in the argument list.
va_count	Returns the number of longwords in the argument list.
va_end	Finishes the *varargs* session.
va_start, va_start_1	Initialize a variable to the beginning of the argument list.
vfprintf	Prints formatted output based on an argument list.
vprintf	Prints formatted output based on an argument list.
vsprintf	Prints formatted output based on an argument list.

3.1 Character Classification Macros

VAX C implements all character classification “functions” as preprocessor defined macros. Do not pass arguments to those macros that may cause side effects, such as arguments with the increment and decrement operators. For more information about macros, see the *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; 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 **isascii** macro, 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 0 indicates false.

Table 3–2 assigns a number to each of the character classification macros.

Table 3–2: Character Classification Macros and their Return Values

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 3–3 lists the numbers of the macros (as assigned in the previous table) 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 3–3: 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

(continued on next page)

Table 3–3 (Cont.): Character Classification Macro Return Values (ASCII Table)

ASCII Values	Macro Numbers	ASCII Values	Macro Numbers
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
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	^ 136	3,6,8,9
US 37	3,4	– 137	3,6,8,9
SP 40	3,8,10	?` 140	3,6,8,9

(continued on next page)

Table 3–3 (Cont.): Character Classification Macro Return Values (ASCII Table)

ASCII Values	Macro Numbers	ASCII Values	Macro Numbers
! 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
% 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

(continued on next page)

Table 3–3 (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

Example 3–1 shows how the character classification macros are used.

Example 3-1: Character Classification 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);
}
```

The sample input and output from Example 3-1 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
$
```

3.2 Character Conversion Functions and Macros

The character conversion functions and macros convert one type of character to another type. These functions include **ecvt**, **fcvt**, **gcvt**, **toascii**, **tolower**, **_tolower**, **toupper**, and **_toupper**. For more information on each of these functions, see the Reference Section.

Example 3–2 shows how to use the **ecvt** function.

Example 3–2: 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 Example 3–2 is as follows:

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

Example 3–3 shows how to use the **toupper** and **tolower** functions.

Example 3–3: 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 Example 3–3 are as follows:

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

3.3 String and Argument List-Handling Functions and Macros

VAX C contains a group of 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.

VAX C also contains a set of functions that allow you to copy buffers containing binary data.

The set of functions and macros defined and declared in the *varargs* and the *stdarg* definition modules provide 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* or *stdargs* 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 that 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;
        .
        .
        .
```

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

```
#include varargs
```

3.4 Program Examples

Example 3–4 shows how to use the **strcat** and **strncat** functions.

Example 3–4: Concatenating 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));
}
```

The sample output from Example 3–4 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 3–5 shows how to use the **strcspn** function.

Example 3–5: Four Arguments to the strcspn 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 charset: %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"));
}
```

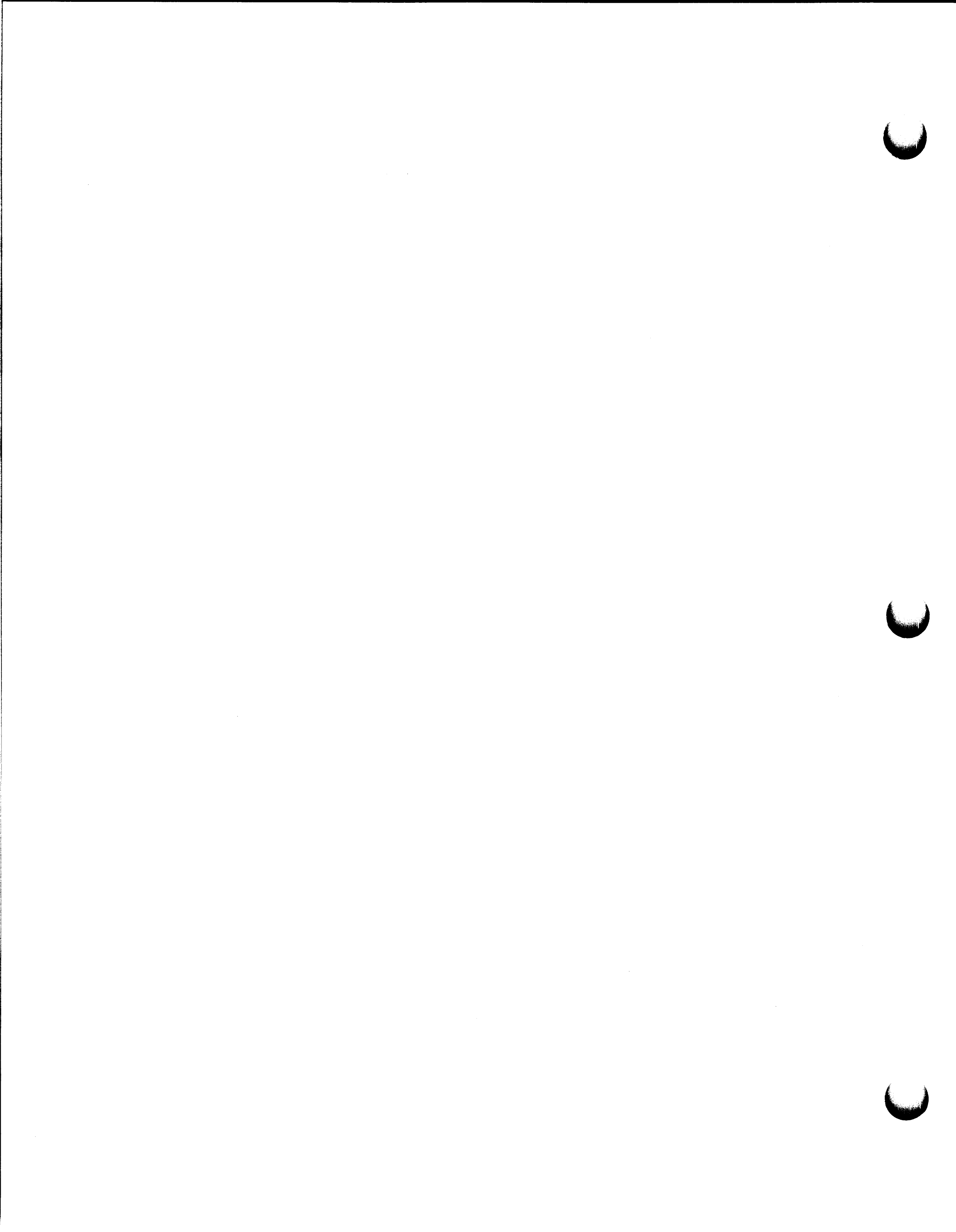
The sample output, to the file strcspn.out, in Example 3–5 is as follows:

```
$ RUN  EXAMPLE2 [RETURN]

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

Example 3–6 shows how to use the *varargs* definition module.

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

```
/*  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 0 */

      ;
   va_end(incrmtr);                  /* End varargs session      */
   return execv(file, args);         /* Return proper values     */
}
```



Chapter 4

Error- and Signal-Handling

Table 4–1 lists and describes all the error- and signal-handling functions and macros found in the VAX C RTL. For more detailed information on each function and macro, see the Reference Section.

Table 4–1: Error- and Signal-Handling Functions and Macros

Error-Handling Functions and Macros	
abort	Executes an illegal instruction that terminates the process.
assert	Puts diagnostics into programs.
atexit	Registers a function that will be called without arguments at program termination.
exit, _exit	Terminate the current process.
perror	Writes a short error message to **stderr** describing the last error encountered during a call to the VAX C RTL from a C program.
strerror	Maps the error number in *errnum* to an error message string.

(continued on next page)

Table 4–1 (Cont.): Error- and Signal-Handling Functions and Macros

Signal-Handling Functions and Macros	
alarm	Sends the signal SIGALARM to the invoking process after the number of seconds indicated by its argument has elapsed.
gsignal	Generates a specified software signal.
kill	Sends a signal to the process specified by a process ID.
longjmp	Provides a way to transfer control from a nested series of function invocations back to a predefined point without returning normally; this is not done by a series of **return** statements.
pause	Causes its calling process to stop until the process receives a signal.
raise	Generates a specified software signal.
setjmp	Provides a way to transfer control from a nested series of function invocations back to a predefined point without returning normally; this is not done by a series of **return** statements.
sigblock	Causes the signals in its argument to be added to the current set of signals being blocked from delivery.
signal	Allows you to either catch or ignore a signal.
sigpause	Assigns its argument to the current set of masked signals and then waits for a signal.
sigsetmask	Establishes the signals that are blocked from delivery.
sigstack	Defines an alternate stack on which to process signals.
sigvec	Assigns a handler for a specific signal.
sleep	Suspends the execution of the current process for at least the number of seconds indicated by its argument.
ssignal	Allows you to specify the action to be taken when a particular signal is raised.
VAXC$ESTABLISH	Establishes a special VAX C RTL exception handler that catches all RTL-related exceptions and passes on all others to your handler.

4.1 Error Handling

When an error occurs during a call to any of the VAX C RTL functions, the function returns an unsuccessful status and sets the external variable, errno, to a value that indicates the reason for the failure. This makes the errno variable 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. At program startup, the value of *errno* is 0. The value of *errno* can be set to a nonzero value by many VAX C RTL functions; it is not reset to zero by any VAX C RTL function. Table 4–2 lists the symbolic values that can be assigned to errno.

Table 4–2: The 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
EAGAIN	No more processes
ENOMEM	Not enough memory

(continued on next page)

Table 4–2 (Cont.): The Errno Symbolic Values

Symbolic Constant	Description
EACCES	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

You can translate the errno values to a message, similar to that found in UNIX systems, by using 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:nontranslatable vms error code: xxxxxx vms message:
```

In the message, %s is the string you supply to **perror**; xxxxxx is the VMS message number.

The VMS error code is available in the vaxc$errno variable and can be examined in your programs. The vaxc$errno variable is declared in the *errno* definition module.

4.2 Signal Handling

Signals are raised by a variety of events, including any of the following events:

- Typing CTRL/C at a terminal (which raises 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 signals 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;
```

sigint
Is the designated signal number.

code
Designates the type of signal if more than one exists.

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 for a signal, it remains in effect until the program calls **sigvec** again to handle that signal.

The handler specified by the scp argument is established as the handler to be called when the signal specified by sigint is raised.

Table 4–3 shows the signals defined in the signal definition module, ways to generate the signals on the VMS operating system, and the attributes of the signals, such as whether or not the signal can be ignored. Unless noted, each signal can be reset and it can be caught or ignored.

Table 4–3: 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/underflow
SIGKILL[2]	Kill	External signal only
SIGBUS	Bus error	Access violation or change mode user
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

[1]Cannot be reset when caught.
[2]Cannot be caught or ignored.

4.3 Program Example

Example 4–1 shows how the **signal**, **alarm**, and **pause** functions operate.

Example 4–1: 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      *
                                     *   the alarm counter      */
   printf("\t<%d\007>", number_of_alarms);
                                    /*  Pass signal and the     *
                                     *   function to SIGNAL     */
   signal(SIGALRM, alarm_action);

   alarm(SECONDS);                  /*  Set the alarm clock     */

   if (--number_of_alarms)          /* Decrement alarm counter */
      pause();
}
```

The sample output from Example 4–1 is as follows:

```
$ RUN  EXAMPLE [RETURN]
          <5> <4>   <3>    <2>   <1>
```

Chapter 5

Subprocess Functions

The VAX C RTL provides functions that allow you to create subprocesses from a VAX C program. The creating process is called the *parent* and the created subprocess is called the *child.*

To create a child process within the parent process, use 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 how to implement and use 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 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).

This chapter discusses the subprocess functions. Table 5–1 lists and describes all the subprocess functions found in the VAX C RTL. For more detailed information on each function and macro, see the Reference Section.

Table 5–1: Subprocess Functions

Function or Macro	Purpose
Implementation of Child Processes	
system	Passes a given string to the host environment to be executed by a command processor.
vfork	Creates an independent child process.
The exec Functions	
excel, execle, execlp execv, execve, execvp	Pass the name of the image to be activated in a child process.
Synchronizing Processes	
wait	Suspends the parent process until a value is returned from the child.
Interprocess Communication	
pipe	Implemented as an array of file descriptors associated with a mail box.

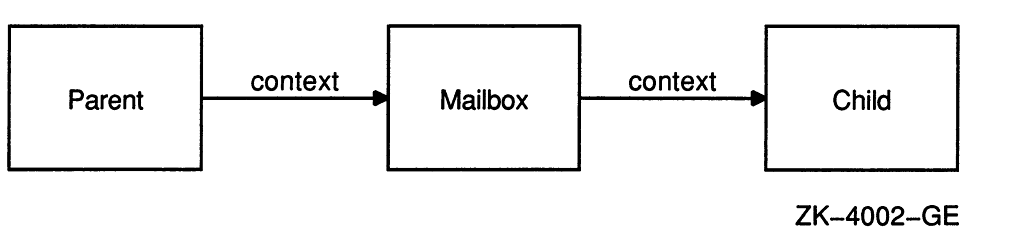
5.1 Implementing Child Processes in VAX C

Child processes are created by VAX C functions with the VMS LIB$SPAWN RTL routine. (See the *VMS Run-Time Library Routines Volume* for information on LIB$SPAWN.) Using 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; 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). You may not run the parent as a detached process or under control of a user-supplied CLI.

Parent and child processes communicate through a mailbox as shown in Figure 5–1. This mailbox transfers the context in which the child will run. The context mailbox, as it is called, passes information to the child that it inherits from the parent, such as the names and file descriptors of all the files opened by the parent and the current location within those files. The mailbox is deleted by the parent when the child image exits.

Figure 5–1: Communications Links Between Parent and Child Processes



ZK–4002–GE

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 use by the VAX C 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 *VMS I/O User's Reference Volume*.

The VMS operating system does not permit two processes to use the same physical terminal for input, and the VAX C RTL does not support file sharing or the default C stream file type. If stdir is connected to a terminal or if stdout or stderr is connected to stream files, these standard streams are redirected to the NUL device _NLA0:.

5.2 The exec Functions

There are six exec functions that you can call to execute a VAX C image in the child process. These functions expect that **vfork** has been called to set up a return address. The exec functions will call **vfork** if the parent process did not.

When **vfork** is called by the parent, exec returns to the parent process. When **vfork** was called by an exec function, the exec returns to itself, waits for the child to exit, and then exits the parent process. The **exec** function 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 RTL 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.

5.2.1 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 about the **umask** function, see the Reference Section.
- 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. 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 about file descriptors, see Chapter 2. For more information on the **lseek** function, see the Reference Section.

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. 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 about the **fdopen** function, see the Reference Section.

Process permanent input files are not inherited by the child process. They are replaced with the null device NLA0. See Section 5.1 for restrictions on the use of the parent's process permanent files by the child process.

- The signal database. 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 is 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.

5.2.2 Exec Error Conditions

The exec functions will fail if LIB$SPAWN cannot create the subprocess. Conditions that can cause a failure include exceeding the subprocess quota or finding the communications by the context mailbox between the parent and child to be broken. Exceeding some quotas will not cause LIB$SPAWN to fail, but will put LIB$SPAWN 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, which returns the child's process ID. In this case, the child process ID returned by exec is less than zero. For more information about the **exit** function, see the Reference Section.

5.3 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 using the VAX C RTL function **wait**.

5.4 Interprocess Communication

You must use a mailbox to read and write data between the parent and child. A channel through which the processes communicate is called a *pipe*. Use the **pipe** function to create a temporary mailbox.

5.5 Program Examples

Example 5–1 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 following the example.

The child process in Example 5–1 prints a message 10 times.

Example 5–1: 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 5–1 (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");
}
```

Key to Example 5–1:

1. The **vfork** function is called to set up the return address for the exec call.

   The **vfork** function is normally 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 0 and the other is nonzero.

2. A 0 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**.

3. A negative child process ID is returned when an exec function fails. The return value is checked for these conditions.

4. The **wait** function is used to synchronize the parent and child processes.

5. Since the exec functions can indicate success up to this point even if the image to be activated in the child does not exist, the parent checks the child's return status for the predefined status, CLI$_IMAGEFNF (file not found).

In Example 5–2, the parent passes arguments to the child process.

Example 5–2: Passing Arguments to the Child Process

```
/*  In this example, the arguments are placed in an array,     *
 *  gargv, but they can be passed to the child                 *
 *  explicitly as a zero-terminated series of character        *
 *  strings. The child program in this example 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 5–2 (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 5–3 shows how to use the **wait** function to check the final status of multiple children being run simultaneously.

Example 5–3: 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, 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 5–3 (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 5–4 shows how to use the **pipe** and **dup2** functions 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 cannot 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. The default case checks for the file-not-found condition, because an exec call cannot detect this condition.

Example 5–4: 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 will not be synchronized.             *
 *                                                             *
 *     PARENT:                                                 *
 *                                                             */

#include perror
#include climsgdef
#include stdio
#define inpipe 11
#define outpipe 12
#include processes
#include unixio

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();
```

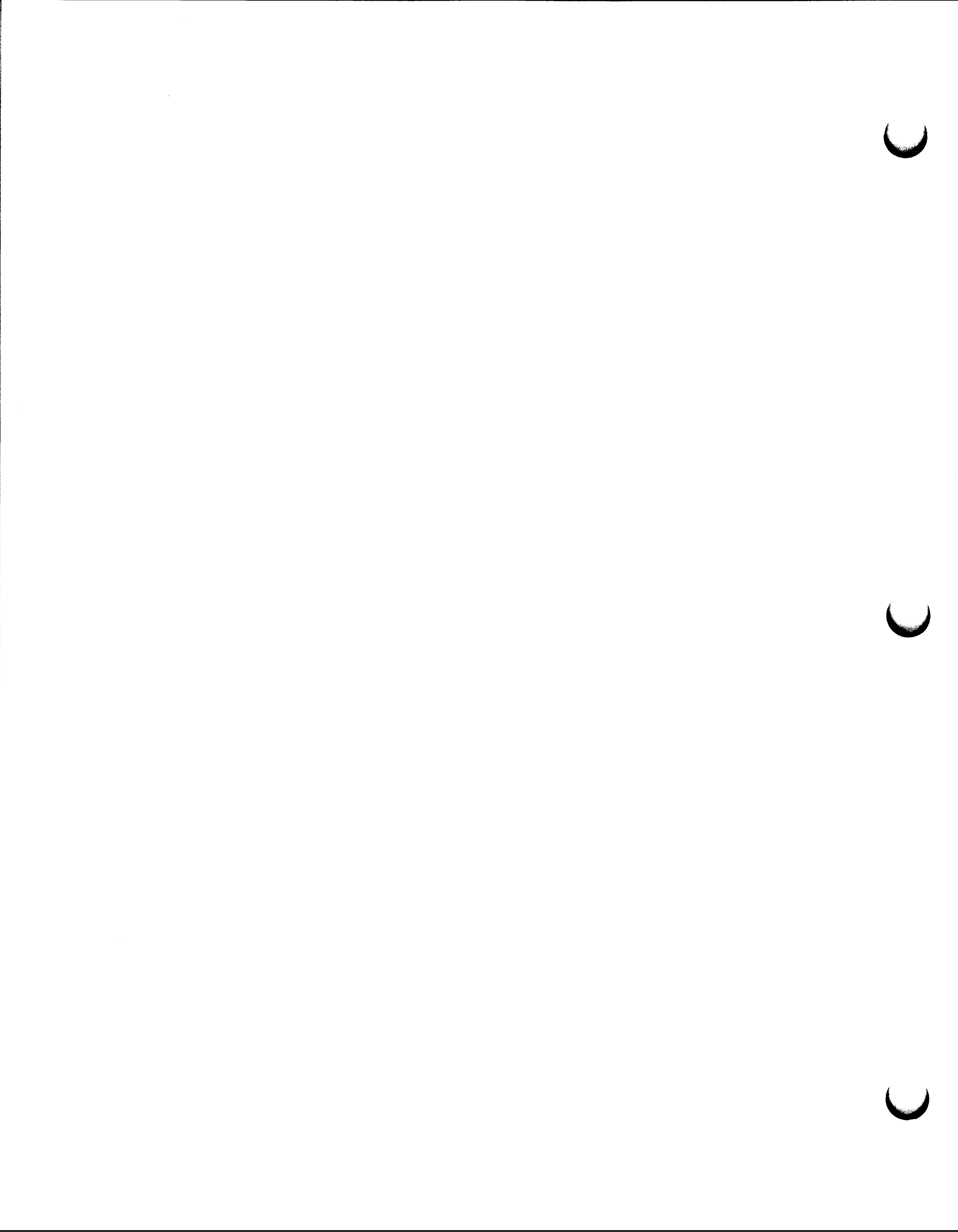
(continued on next page)

```
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 5–4 (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 6

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.

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, see the *Guide to VAX C*.

To implement Curses, the terminal-independent Screen Management Software, which is part of the VMS RTL, is used. For portability purposes, most functions and macros are designed to perform in a manner similar to other C implementations. However, VAX C Curses depends upon the VMS system 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.

Table 6–1 lists all of the Curses functions and macros found in the VAX C RTL. For more detailed information on each function and macro, see the Reference Section.

Table 6–1: Curses Functions and Macros

Function or Macro	Purpose
[w]addch	Add the character ch to the window at the current position of the cursor.
[w]addstr	Add the string pointed to by an argument to the window at the current position of the cursor.
box	Draws a box around the window.
[w]clear	Erase the contents of the specified window and reset the cursor to coordinates (0,0).
clearok	Sets the clear flag for the window.
[w]clrattr	Deactivate the video display attribute within the window.
[w]clrtobot	Erase the contents of the window from the current position of the cursor to the bottom of the window.
[w]clrtoeol	Erase the contents of the window from the current cursor position to the end of the line on the specified window.
[no]crmode	Set and unset the terminal from cbreak mode.
[w]delch	Delete the character on the specified window at the current position of the cursor.
[w]deleteln	Delete the line at the current position of the cursor.
delwin	Deletes the specified window from memory.
[no]echo	Set the terminal so that characters may or may not be echoed on the terminal screen.
endwin	Clears the terminal screen and frees any virtual memory allocated to Curses data structures.
[w]erase	Erase the window by painting it with blanks.
[w]getch	Get a character from the terminal screen and echo it on the specified window.
[w]getstr	Get a string from the terminal screen, store it in a character variable, and echo it on the specified window.
getyx	Puts the (y,x) coordinates of the current cursor position on the window in the variables y and x.

(continued on next page)

Table 6–1 (Cont.): Curses Functions and Macros

Function or Macro	Purpose
[w]inch	Return the character at the current cursor position on the specified window without making changes to the window.
initscr	Initializes the terminal-type data and all screen functions.
[w]insch	Insert a character variable at the current cursor position in the specified window.
[w]insertln	Insert a line above the line containing the current cursor position.
[w]insstr	Insert a string at the current cursor position on the specified window.
leaveok	Signals Curses to leave the cursor at the current coordinates after an update to the window.
longname	Assigns the full terminal name to a character name that must be large enough to hold the character string.
[w]move	Change the current cursor position on the specified window to the coordinates (y,x).
mv[w]addch	Move the cursor to (x,y) and add the character variable to the specified window.
mv[w]addstr	Move the cursor to (x,y) and add the specified string to the specified window.
mvcur	Moves the terminal's cursor.
mv[w]delch	Move the cursor to (x,y) and delete the character on the specified window.
mv[w]getch	Move the cursor to (y,x), get a character from the terminal screen, and echo it on the specified window.
mv[w]getstr	Move the cursor (y,x), get a string from the terminal screen, store it in a variable that must be large enough to contain the string, and echo it on the specified window.
mv[w]inch	Move the cursor (y,x) and return the character on the specified window without making changes to the window.
mv[w]insch	Move the cursor (y,x) and insert a character variable in the specified window.

(continued on next page)

Table 6–1 (Cont.): Curses Functions and Macros

Function or Macro	Purpose
mv[w]insstr	Move the cursor (y,x) and insert a string in the specified window.
mvwin	Moves the starting position of the window to the specified (y,x) coordinates.
newwin	Creates a new window with lines and columns starting at the coordinates on the terminal screen.
[no]nl	Provided only for UNIX software compatibility and have no functionality in the VMS environment.
overlay	Writes the contents of one window that will fit over the contents of another window, beginning at the starting coordinates of both windows.
overwrite	Writes the contents of one window, insofar as it will fit, over the contents of another window beginning at the starting coordinates of both windows.
[w]printw	Perform a **printf** on the window starting at the current position of the cursor.
[no]raw	Provided only for UNIX software compatibility and have no functionality in the VMS environment.
[w]refresh	Repaint the specified window on the terminal screen.
[w]scanw	Perform a **scanf** on the window.
scroll	Moves all the lines on the window up one line.
scrollok	Sets the scroll flag for the specified window.
[w]setattr	Activate the video display attribute within the window.
subwin	Creates a new subwindow with lines and columns starting at the coordinates on the terminal screen.
[w]standend	Deactivate the boldface attribute for the specified window.
[w]standout	Activate the boldface attribute of the specified window.
touchwin	Places the most recently edited version of the specified window on the terminal screen.
wrapok	Allows the wrapping of a word from the right border of the window to the beginning of the next line.

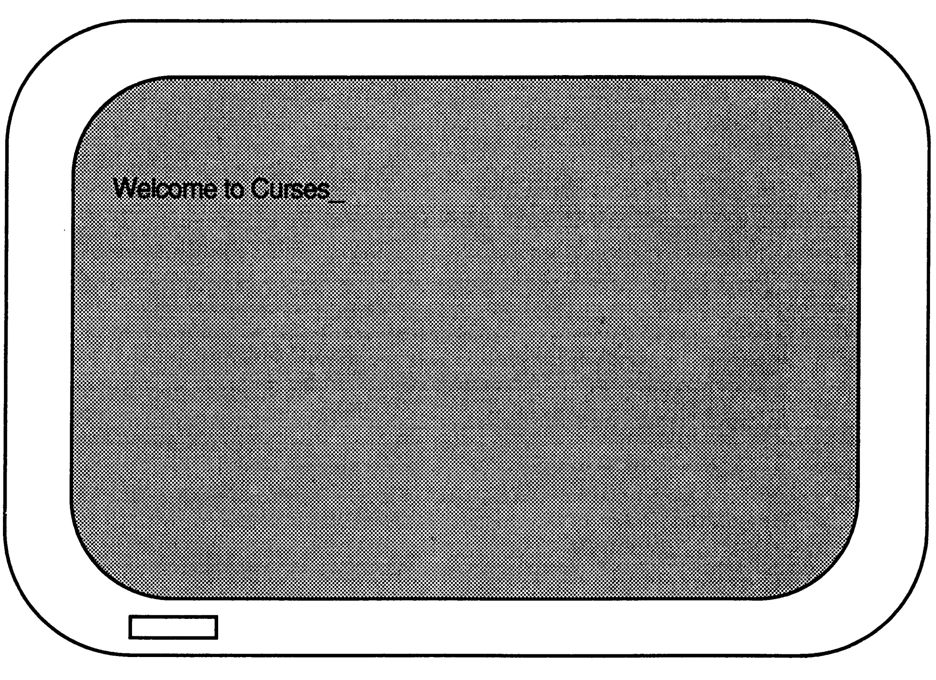
6.1 Curses Terminology

This section explains some of the Curses terminology and shows you how Curses looks on the terminal screen.

Consider 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 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 will look like that in Figure 6–1.

Figure 6–1: An Example of the stdscr Window



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 your Curses applications.

6.1.1 User-Defined Windows

You can 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.

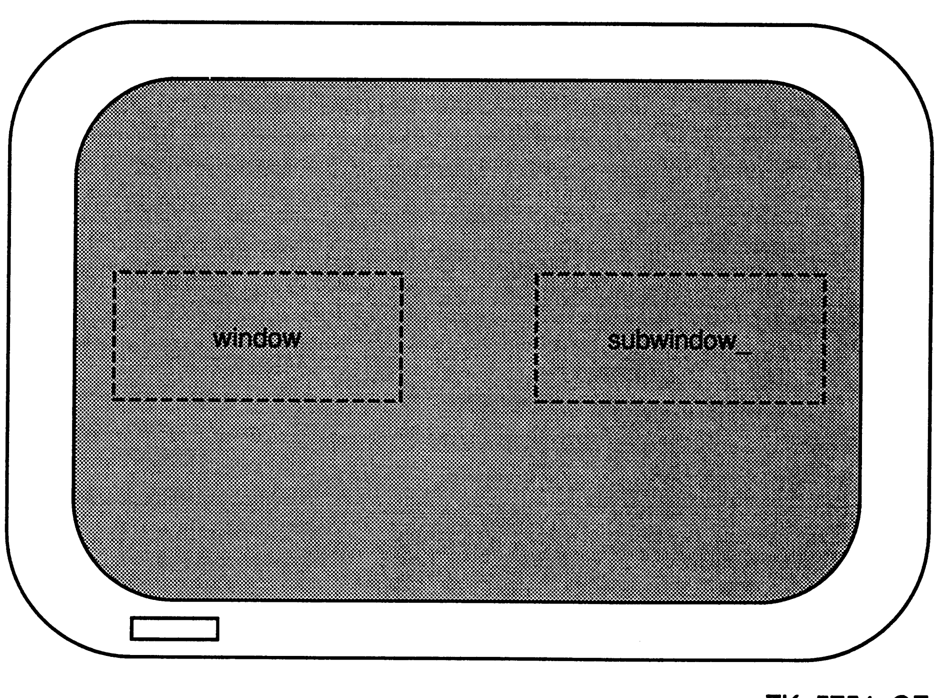
When writing to or deleting from windows, changes do not appear on the terminal screen until the window is *refreshed*. When refreshing a window, you place the updated window onto the terminal screen, which leaves the rest of the screen unaltered.

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 you create a window that occludes stdscr and a subwindow of stdscr, your terminal screen will look like Figure 6–2.

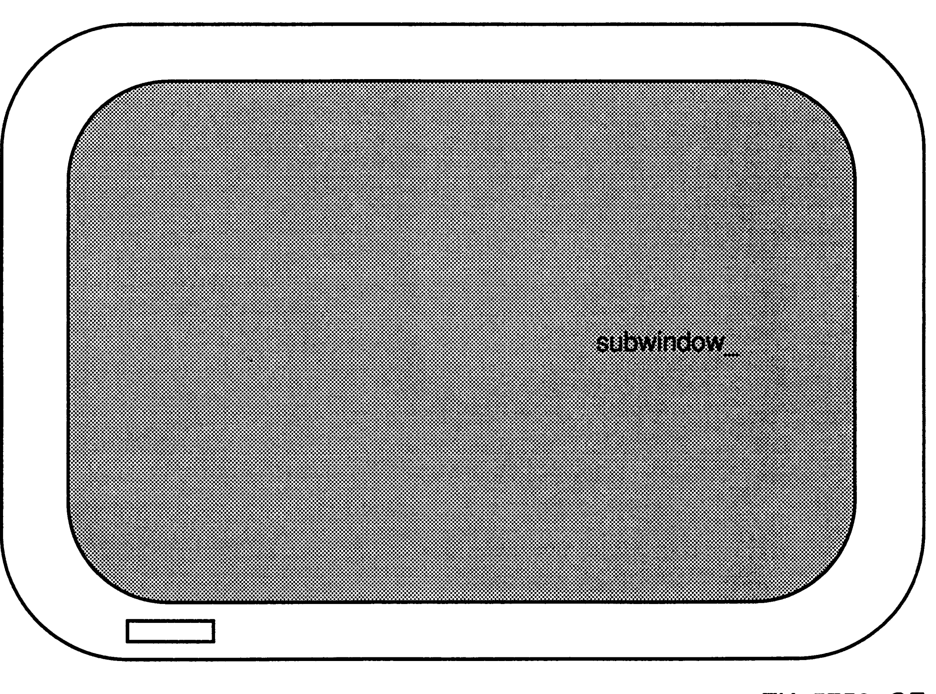
Figure 6–2: Displaying Windows and Subwindows



ZK–5754–GE

If you delete both the user-defined window and the subwindow, and then update the terminal screen with the new image, your terminal screen will look like Figure 6–3.

Figure 6–3: Updating the Terminal Screen



The string written on the window is deleted, but the string written on the subwindow remains on stdscr.

6.2 Getting Started with Curses

There are commands that 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 6–1 shows how to set up a program using Curses.

Example 6–1: A Curses Program

```
❶ #include curses

❷ WINDOW  *win1, *win2, *win3;

main()
{
❸    initscr();
        .
        .
        .
     endwin();
}
```

Key to Example 6–1:

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 Example 6–1, 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 6–2 shows you how to define and write to a single window.

Example 6–2: Manipulating Windows

```
#include curses

WINDOW  *winl, *win2, *win3;

main()
{
   initscr();

❶  winl = newwin(24, 80, 0, 0);
❷  mvwaddstr(winl, 2, 2, "HELLO");
       .
       .
       .
   endwin();
}
```

Key to Example 6–2:

❶ The **newwin** function defines a window 24 rows high and 80 columns wide with a starting position at coordinates (0,0), the upper left corner of the terminal screen. The program assigns these attributes to winl. The coordinates are specified as follows: (lines,columns) or (y,x).

❷ The **mvwaddstr** macro performs the same task as a call to the separate macros **move** and **addstr**. The **mvwaddstr** macro moves the cursor to the specified coordinates and writes a string onto stdscr.

NOTE

> 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. 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 want to add a character to the window at its starting position, specify the coordinates (0,0), not (10,10).

The string HELLO in Example 6–2 does not appear on the terminal screen until you refresh the screen. You accomplish this by using the **wrefresh** function. Example 6–3 shows how to display the contents of winl on the terminal screen.

Example 6–3: 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 want all of win1 to be displayed on the terminal screen, call the **touchwin** function.

6.3 Predefined Variables and Constants

There is a group of variables, defined in the *curses* definition module, that is useful when you implement Curses. There is also a group of constants defined in *curses*, using the **#define** preprocessor directive, that are useful. Table 6–2 describes the variables and constants defined in the *curses* definition module.

Table 6–2: 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 the terminal screen

(continued on next page)

Table 6–2 (Cont.): Curses Predefined Variables and #define Constants

Name	Type	Description
COLS	int	VAR: Number of columns on the 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 6–4 shows how to perform such a test.

Example 6–4: 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 Example 6–4, if the **mvwin** function fails, the program adds a string to stdscr that explains the outcome. The Curses **mvwin** function moves the starting position of a window.

6.4 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, which moves the cursor in a more efficient manner.

In VAX C, the two functions are functionally equivalent and move the cursor with the same efficiency.

Example 6–5 shows how to use the **move** and **mvcur** functions.

Example 6–5: 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();
}
```

Key to Example 6–5:

❶ The **clear** macro erases stdscr and 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 **mvcur** function forces absolute addressing. This function can 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 if you are unsure of the current position of the cursor, but **move** works just as well.

6.5 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 6–6 shows the definition and manipulation of one user-defined window and stdscr.

Example 6–6: 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      */
```

(continued on next page)

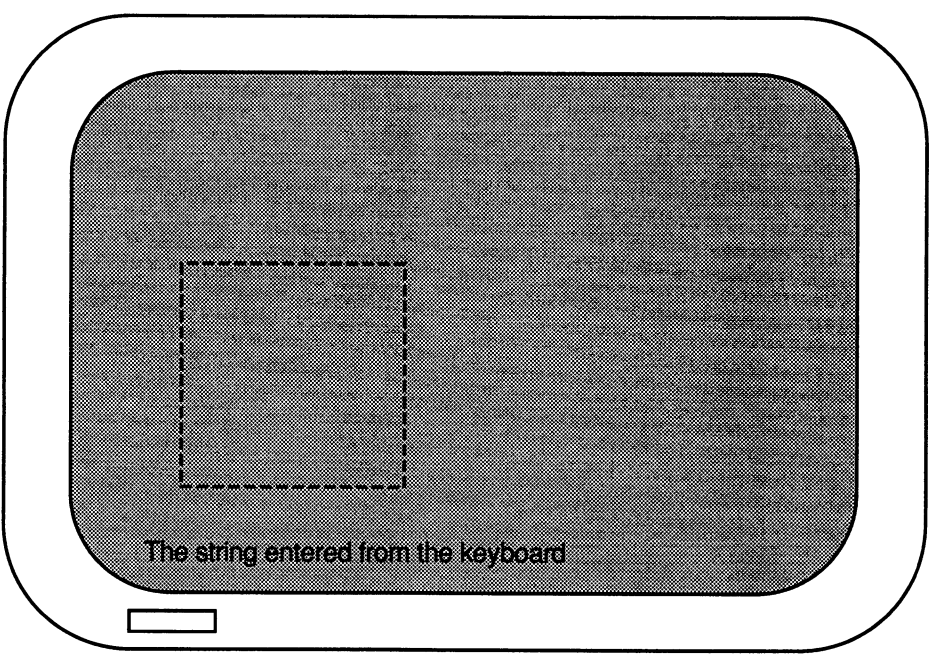
Example 6–6 (Cont.): stdscr and Occluding Windows

```
    getch();                    /* Pause.  Press RETURN      */
    endwin();                   /* Ends session.             */
}
```

Key to Example 6–6:

1. The program waits for input. The echo was disabled using the **noecho** macro, so 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.
2. 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 6–4.
3. The **touchwin** function refreshes the screen so that all of stdscr is visible and the deleted occluding window no longer appears on the screen.

Figure 6–4: An Example of the getch Macro



Example 6–7 shows the **overlay** function.

Example 6–7: Subwindows

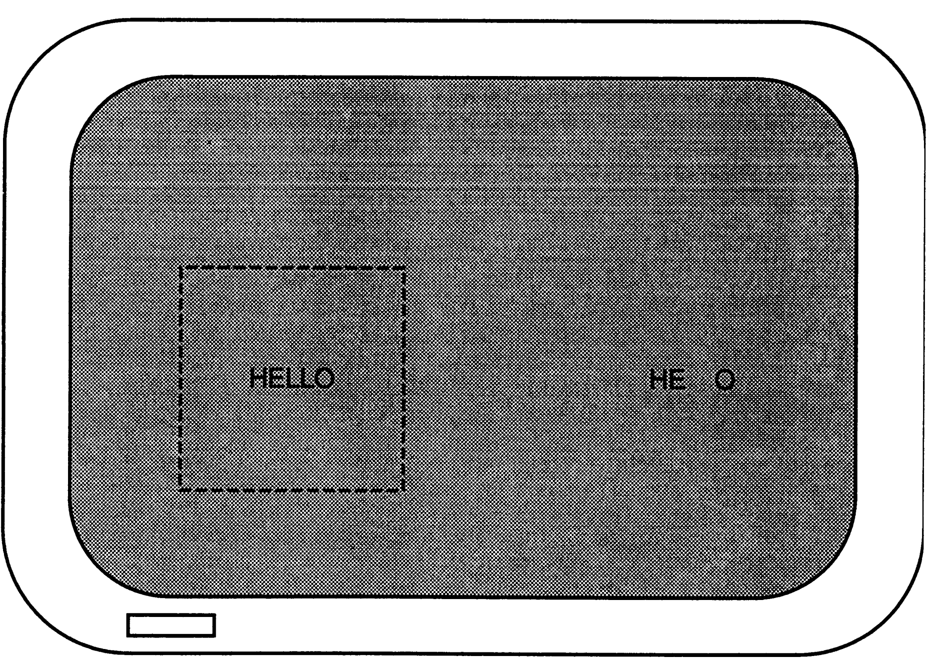
```
/*  The following program creates subwindows ---  WIN1     *
 *  and WIN2 --- and shows the effects of OVERLAY.         */

#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.            */
}
```

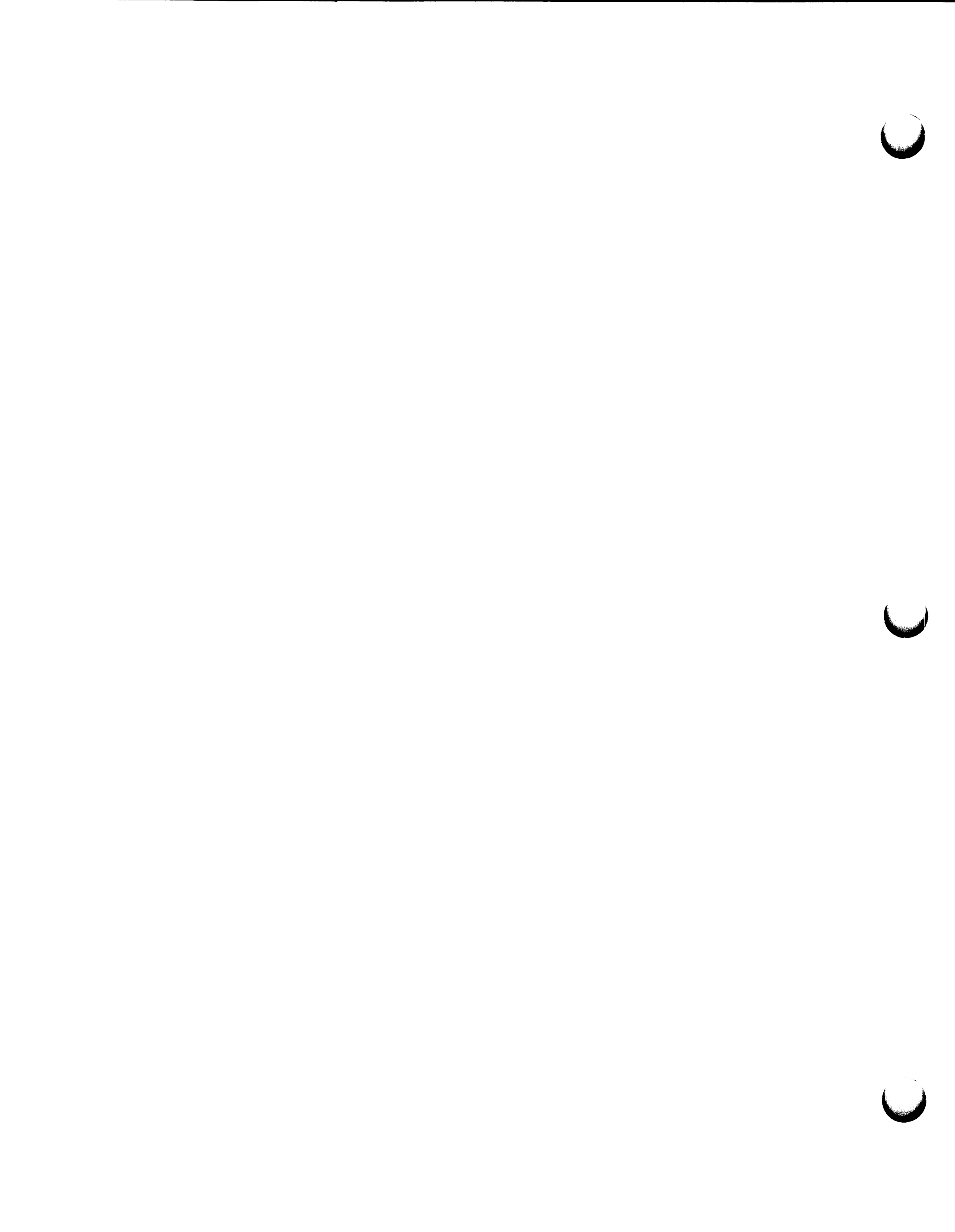
Key to Example 6–7:

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 will appear instead of HELLO, with the blanks overwriting the letters. The screen appears like Figure 6–5.

Figure 6–5: An Example of Overwriting Windows





Chapter 7

Math Functions

Table 7–1 lists and describes all the math functions and macros found in the VAX C RTL. For more detailed information on each function and macro, see the Reference Section.

Table 7–1: Math Functions

Function or Macro	Purpose
abs	Returns the absolute value of an integer.
acos	Returns a value in the range 0 to π, which is the arc cosine of its radian argument.
asin	Returns a value in the range $\pi/2$ to $\pi/2$, which is the arc sine of its radian argument.
atan	Returns a value in the range $-\pi/2$ to $\pi/2$, which is the arc tangent of its radian argument.
atan2	Returns a value in the range $-\pi$ to π, which is the arc tangent of y/x where y and x are the two arguments.
cabs	Return: sqrt (x * x + y * y).
ceil	Returns (as a **double**) the smallest integer that is greater than or equal to its argument.
cos	Returns the cosine of its radian argument.
cosh	Returns the hyperbolic cosine of its argument.
exp	Returns the base e raised to the power of the argument.
fabs	Returns the absolute value of a floating-point value.

(continued on next page)

Table 7–1 (Cont.): Math Functions

Function or Macro	Purpose
floor	Returns (as a **double**) the largest integer that is less than or equal to its argument.
fmod	Computes the floating-point remainder of the first argument to **fmod** divided by the second.
frexp	Returns the mantissa of a **double** value.
hypot	Returns the square root of the sum of the squares of two arguments.
labs	Returns the absolute value of an integer as a **long int**.
ldexp	Returns its first argument multiplied by 2 raised to the power of its second argument.
ldiv, div	Return the quotient and remainder after the division of their arguments.
log, log 10	Return the logarithm of their arguments.
modf	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.
pow	Returns the first argument raised to the power of the second argument.
rand, srand	Return pseudorandom numbers in the range 0 to $2^{31} - 1$.
sin	Returns the sine of its radian argument.
sinh	Returns the hyperbolic sine of its argument.
sqrt	Returns the square root of its argument.
tan	Returns a **double** value that is the tangent of its radian argument.
tanh	Returns a **double** value that is the hyperbolic tangent of its **double** argument.

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 occurs.

The following program example checks the variable errno for the value EDOM, which indicates 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 returns 0 when a negative number is entered. For more information about the *errno* definition module, see Chapter 4.

Example 7–1 shows how the **tan, sin**, and **cos** functions operate.

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));
}
```

The sample output from Example 7–1 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

Memory Allocation Functions

Table 8–1 lists and describes all the memory allocation functions and macros found in the VAX C RTL. For a more detailed description of each function and macro, see the Reference Section.

Table 8–1: Memory Allocation Functions

Function or Macro	Purpose
brk, sbrk	Determine the lowest virtual address that is not used with the program.
calloc, malloc	Allocate an area of memory.
cfree, free	Make available for reallocation the area allocated by a previous **calloc, malloc,** or **realloc** call.
realloc	Changes the size of the area pointed to by the first argument to the number of bytes given by the second argument.
VAXC$CALLOC_OPT	Allocates an area of memory.
VAXC$CFREE_OPT	Makes available for reallocation the area allocated by a previous call to **VAXC$CALLOC_OPT**, **VAXC$MALLOC_OPT**, or **VAXC$REALLOC_OPT**.
VAXC$FREE_OPT	Makes available for reallocation the area allocated by a previous call to **VAXC$CALLOC_OPT**, **VAXC$MALLOC_OPT**, or **VAXC$REALLOC_OPT**.
VAXC$MALLOC_OPT	Allocates an area of memory.

(continued on next page)

Table 8–1 (Cont.): Memory Allocation Functions

Function or Macro	Purpose
VAXC$REALLOC_ OPT	Changes the size of the area pointed to by the first argument to the number of bytes given by the second argument.

All the VAX C RTL functions requiring 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 some time to supply the virtual memory, so the VAX C RTL tries to reduce the number of calls to these functions in the following manner.

The VAX C RTL 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 the VMS system by LIB$FREE_VM. Instead, the VAX C RTL tries 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 the VMS system through LIB$FREE_VM. The new block is then saved to be used, if possible, for the next request.

Since the VAX C RTL 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$GET_VM is not the same as the address given to you by the VAX C RTL routines. Therefore, any memory allocated by a VAX C RTL routine must be deallocated by a VAX C RTL routine. Similarly, any memory allocated by LIB$GET_VM must be deallocated by LIB$FREE_VM.

The **brk** and **sbrk** functions assume that memory can be allocated contiguously from the top of your address space. However, the **malloc** function and RMS may allocate space from this same address space. You

should not use the **brk** and **sbrk** functions in conjunction with RMS and VAX C RTL routines that use **malloc**.

8.1 Program Example

Example 8–1 shows the use of the **malloc, calloc, free, and cfree** functions.

Example 8–1: 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 = NULL, *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));
```

(continued on next page)

```
          if (next_line == NULL)
             {
                perror("calloc");
                exit();
             }
                                      /* 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      */

    while (next_line != NULL);
       {
          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);
       }
}
```

The sample input and output for Example 8–1 are as follows:

```
$ RUN  EXAMPLE [RETURN]
line one
line two
[CTRL/Z]
EXIT
line one
line two
$
```

Chapter 9

System Functions

The C programming language is a good choice if you 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 and other miscellaneous functions.

Table 9–1 lists and describes all the system functions found in the VAX C RTL. For a more detailed description of each function and macro, see the Reference Section.

Table 9–1: System Functions

Function or Macro	Purpose
System Functions—Searching and Sorting Utilities	
bsearch	Performs a binary search on an array of sorted objects for a specified object.
qsort	Sorts an array of objects in place by implementing the quick-sort algorithm.

(continued on next page)

Table 9–1 (Cont.): System Functions

Function or Macro	Purpose
System Functions—Retrieving Process Information	
ctermid	Returns a character string giving the equivalence string of SYS$COMMAND, which is the name of the controlling terminal.
cuserid	Returns a pointer to a character string containing the name of the user who initiated the current process.
getcwd	Returns a pointer to the file specification for the current working directory.
getegid, geteuid, getgid, getuid	Return, in VMS terms, group and member numbers from the user identification code (UIC).
getenv	Searches the environment array for the current process and returns the value associated with a specified environment.
getpid	Returns the process ID of the current process.
getppid	Returns the parent process ID of the calling process.
System Functions—Changing Process Information	
chdir	Changes the default directory.
chmod	Changes the file protection of a file.
chown	Changes the owner user identification code (UIC) of a file.
mkdir	Creates a directory.
nice	Increases or decreases the process priority to the process base priority by the amount of the argument.
setgid, setuid	Implemented for program portability and have no functionality.
umask	Creates a file protection mask that is used whenever a new file is created. It returns the old mask value.

(continued on next page)

Table 9–1 (Cont.): System Functions

Function or Macro	Purpose
System Functions—Retrieving Time Information	
asctime	Converts a broken-down time into a 26-character string.
clock	Determines the CPU time (in microseconds) used since the beginning of the program execution.
ctime	Converts a time, in seconds, to an ASCII string to the form generated by the **asctime** function.
difftime	Computes the difference, in seconds, between the two times specified by its arguments.
ftime	Returns the elapsed time since 00:00:00, January 1, 1970, in the structure timeb.
gmtime	Converts a given calendar time into a broken-down time, expressed as Greenwich Mean Time (GMT).
localtime	Converts a time (expressed as the number of seconds elapsed since 00:00:00, January 1, 1970) into hours, minutes, seconds, and so on.
time	Returns the time elapsed since 00:00:00, January 1, 1970, in seconds.
times	Returns the accumulated times of the current process and of its terminated child processes.
System Functions—Miscellaneous	
VAXC$CRTL_INIT	Allows a call from other languages by initializing the run-time environment and establishing an exit and condition handler.

Example 9–1 and Example 9–2 show how the **cuserid** function is used.

Example 9–1: 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);
}
```

If a user named TOLLIVER is running the program, the output to stdout is as follows:

```
$ RUN  EXAMPLE1 RETURN
Initiating user: TOLLIVER
```

Example 9–2 produces the same output.

Example 9–2: 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 9–3 shows the **getenv** function.

Example 9–3: Accessing Terminal Information

```
cfunc()
{
    printf("Terminal type: %s\n", getenv("TERM"));
}
```

If the terminal in use is a DIGITAL VT100 in 132-column mode, the sample output from Example 9–3 is as follows:

```
$ RUN  EXAMPLE3 RETURN
Terminal type: vt100-132
```

Example 9–4 shows how to use **getenv** to find the user's default login directory and how to use **chdir** to change to that directory.

Example 9–4: 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);
}
```

The sample output from Example 9–4 is as follows:

```
$ RUN EXAMPLE4 RETURN
Current directory: dba0:[tolliver]
$
```

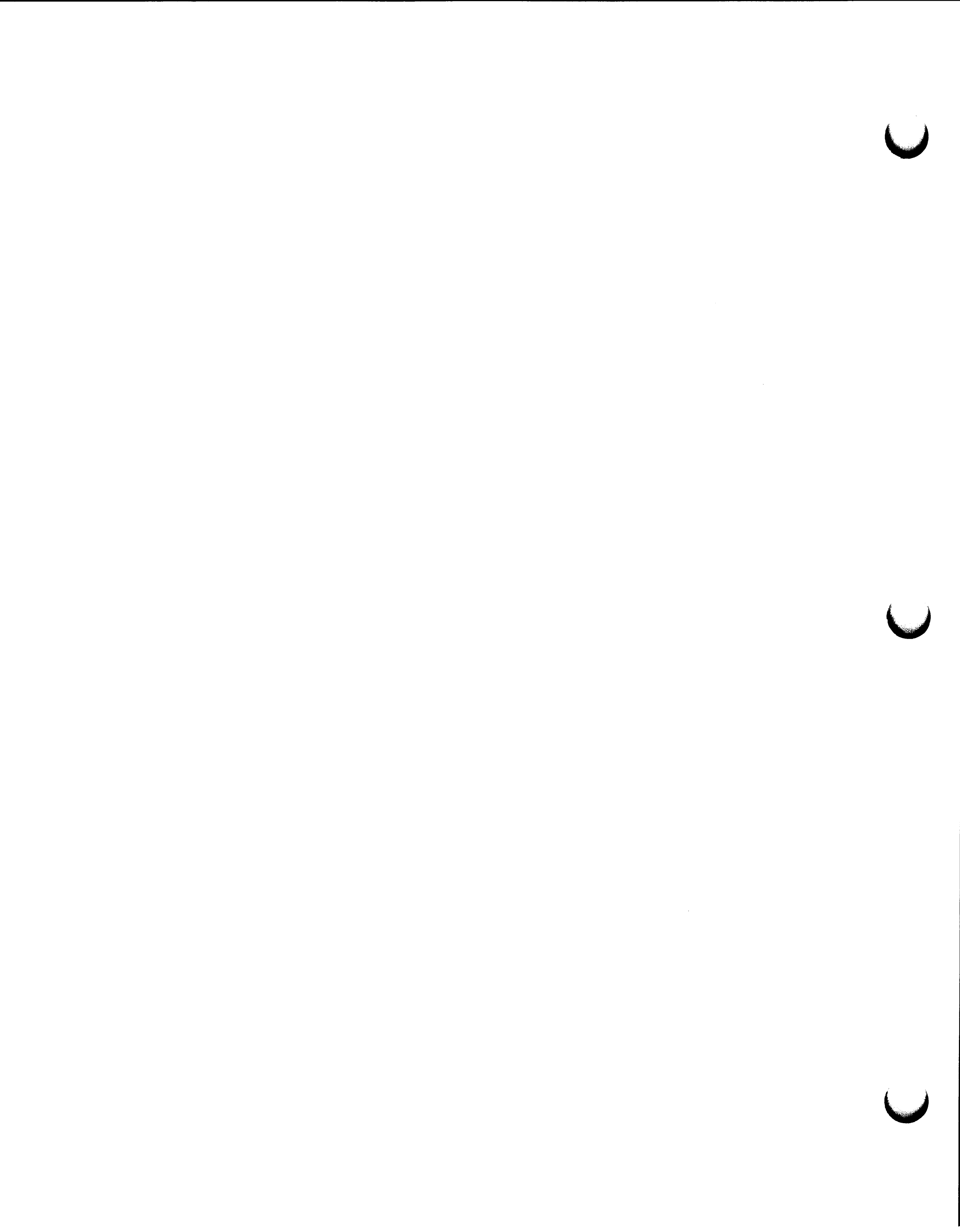
Example 9–5 shows how to use the **time** and **localtime** functions to print the correct date and time at the terminal.

Example 9–5: 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;
   time_t time_val;
   int 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]);
}
```

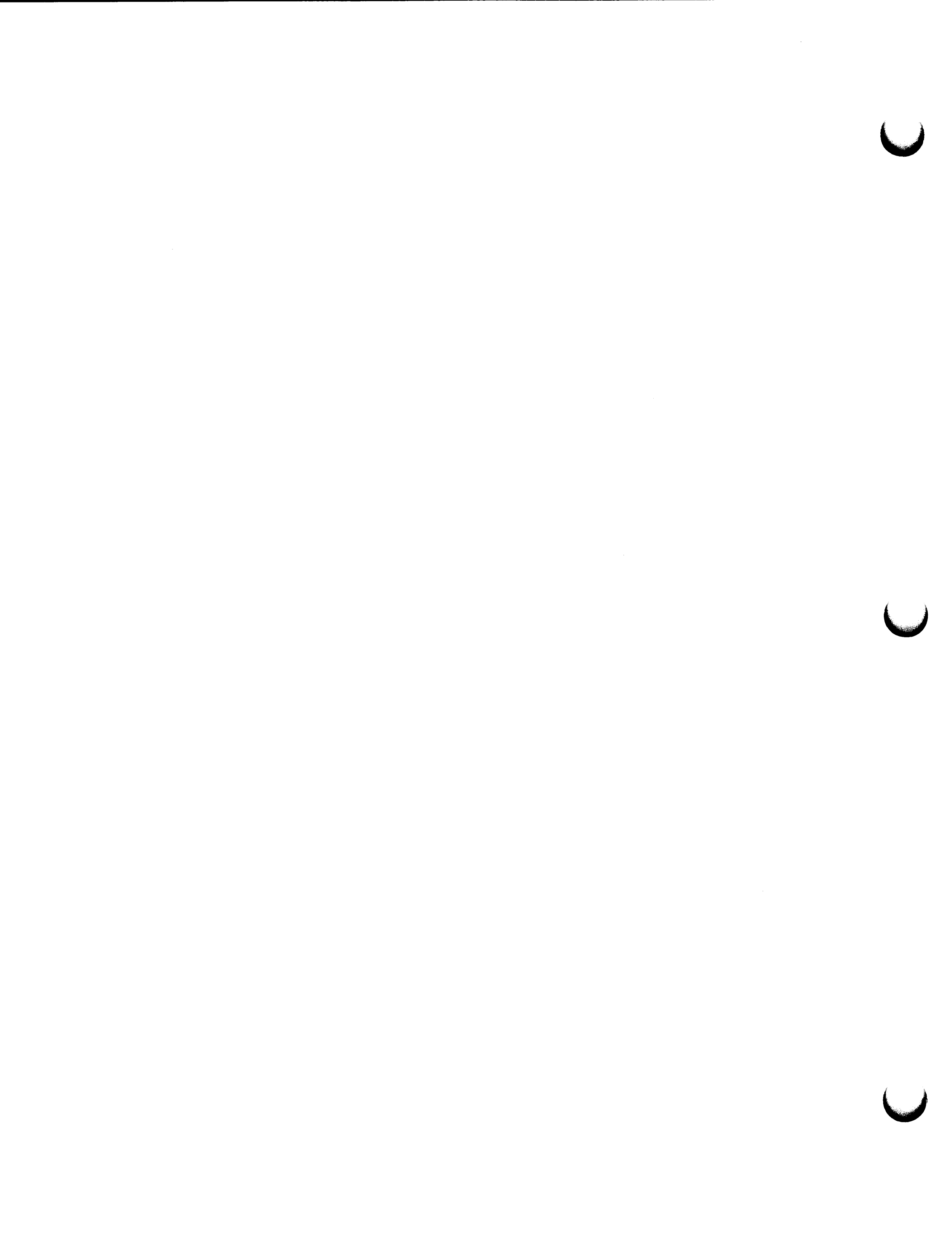
The sample output from Example 9–5 is as follows:

```
$ RUN  EXAMPLE5 RETURN
Today is Thursday, February 7, 1985
The time is 10:18 AM
$
```



Reference Section

This section alphabetically describes all the functions and macros contained in the VAX C Run-Time Library.



abort

The **abort** function executes an illegal instruction that terminates the process.

Format

#include *stdlib*

void abort *(***void***);*

abs

The **abs** function returns the absolute value of an integer.

Format

#include *stdlib*

int abs *(***int** *x);*

Arguments

x
Is an integer.

access

The **access** function checks a file to see whether a specified access mode is allowed. This function only checks UIC protection; ACLs are not checked.

NOTE

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

Format

#include *stdio*

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

Arguments

file_spec
Is a character string that gives a VMS or UNIX-style file specification. The usual defaults and logical name translations are applied to the file specification.

mode
Is interpreted as follows in Table REF–1.

Table REF–1: Interpretation of the mode Argument

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.

Return Values

0	Indicates that the access is allowed.
EOF	Indicates that the access is not allowed.

Example

```
#include stdio
main()
{
if (access("cdtm$:[c.don]dtm.com",0))
        perror("ACCESS - FAILED"),
         exit(2);

}
```

acos

The **acos** function returns a value in the range 0 to π, which is the arc cosine of its radian argument.

Format

#include *math*

double acos (**double** *x);*

Arguments

x
Is a radian expressed as a real value.

Description

When x is a real number greater than 1, the value of acos(x) is 0 and the **acos** function sets errno to EDOM.

[w]addch

The **addch** macro and the **waddch** function add the character ch to the window at the current position of the cursor.

Format

#include *curses*

#define bool int

addch *(ch);*

int waddch *(***WINDOW** **win,* **char** *ch);*

Arguments

win
Is a pointer to the window.

ch
Is an object of type **char**. If the character is a newline (\n), the **addch** macro and **waddch** 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.

Description

When the **waddch** function is used on a subwindow, it writes the character onto the underlying window as well. For more information, see the **scrollok** function in this section.

The **addch** macro performs the same function as the **waddch** function but on the stdscr window.

Return Values

ERR	Indicates that the function causes the screen to scroll illegally.
1	Indicates success.

[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.

Format

#include *curses*

#define bool int

addstr *(str);*

int waddstr *(***WINDOW** **win,* **char** **str);*

Arguments

win
Is a pointer to the window.

str
Is a pointer to a character string.

Description

When the **waddstr** function is used on a subwindow, the string is written onto the underlying window as well. For more information, see the **scrollok** function in this section.

The **addstr** macro performs the same function as the **waddstr** function but on the stdscr window.

Return Values

ERR	Indicates that the function causes the screen to scroll illegally, but it places as much of the string onto the window as possible.
1	Indicates success.

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.

Format

#include *signal*

int alarm **(unsigned int** *seconds);*

Arguments

seconds
Has a maximum limit of 4,294,967,295 seconds.

Description

Calling the **alarm** function with a 0 argument cancels any pending alarms.

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 held up due to scheduling delays.

When the SIGALRM signal is generated, a call to SYS$WAKE is generated whether or not the process is hibernating. The pending wake causes either the current **pause**( ) or a subsequent **pause**( ) to return immediately (after completing any function that catches the SIGALRM).

Return Values

n	Indicates the number of seconds remaining from a previous alarm request.

asctime

The **asctime** function converts a broken-down time (see the **localtime** function for more information) into a 26-character string in the following form:

```
Sun Sep 16 01:03:52 1984\n\0
```

All fields have a constant width.

Format

#include *time*

char *asctime **(const tm_t** ***timeptr);***

Arguments

timeptr
Is a pointer to a structure of type **tm**, which contains the broken-down time.

Description

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;
```

The **asctime** function converts the contents of tm into a 26-character string, as shown in the previous 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 the **localtime** function in this section for a list of the members in tm.

Return Values

x	Indicates a pointer to the string.

asin

The **asin** function returns a value in the range $-\pi/2$ to $\pi/2$, which is the arc sine of its radian argument.

Format

#include *math*

double asin *(***double** *x)* **;**

Arguments

x
Is a radian expressed as a real number.

Description

When x is a real number greater than 1, the value of asin(x) is 0 and the **asin** function sets errno to EDOM.

assert

The **assert** function puts diagnostics into programs.

Format

#include *assert*

void assert *(***int** *expression);*

Arguments

expression
Is an expression that has an **int** type.

Description

When the **assert** macro is executed, if *expression* is false (that is, it evaluates to 0), 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 **assert** macro writes a message in the following form:

```
Assertion failed:  expression, file aaa, line nnn
```

If expression is true (that is, it 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 you use **#undef** to remove the macro definition and obtain access to a real function, the behavior is undefined.

Example

```
#include stdio
#include assert
main(){

printf("Only this and the assert");
assert( 1==2 );     /* expression is FALSE */

/* abort should be called so the printf will not happen. */

printf("FAIL abort did not execute");
```

atan

The **atan** function returns a value in the range $-\pi/2$ to $\pi/2$, which is the arc tangent of its radian argument.

Format

#include *math*

double atan *(***double** *x);*

Arguments

x
Is a radian expressed as a real value.

atan2

The **atan2** function returns a value in the range $-\pi$ to π. The returned value is the arc tangent of y/x, where y and x are the two arguments.

Format

#include *math*

double atan2 *(***double** *y,* **double** *x);*

Arguments

y
Is a real value.

x
Is a real value.

atexit

The **atexit** function registers a function that is called without arguments at program termination.

Format

#include *stdlib*

int atexit *(***void** *(*func) (***void***));*

Arguments

func
Is a pointer to the function to be registered.

Description

Up to 32 functions can be registered. However, you should not register a function more than once.

Return Values

zero	Indicates that the registration has succeeded.
nonzero	Indicates failure.

Example

```
#include stdlib.h
#include stdio.h

static void hw(void);

main()
{
    atexit(hw);
}

    static void hw()
{
        puts("Hello, world\n");
}
```

atof

The **atof** function converts a given string to a double-precision number.

This function recognizes 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.

Format

#include *stdlib*

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

Arguments

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

Description

For **atof**, overflows resulting from the conversion are not accounted for, **strtod(str,(char **)0)**, arithmetic exceptions not withstanding.

Return Values

n	Indicates the converted value.

atoi, atol

The **atoi** and **atol** functions convert strings of ASCII characters to the appropriate numeric values.

Format

#include *stdlib*

int atoi *(***const char** **nptr);*

long int atol *(***const char** **nptr);*

Arguments

nptr
Is a pointer to the character string to be converted to a **long**.

Description

The **atoi** and **atol** functions recognize strings in various formats, depending on the value of the base. These functions are the same in VAX C. The **atoi** and **atol** functions do not account for overflows resulting from the conversion. 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 equal to **strtol (str, (char**)0, 10)**. Similarly, the function call **atoi (str)** is equivalent to **(int) strtol (str, (char**)0, 10)**.

Return Values

n	Indicates the converted value.

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.

Format

#include *curses*

#define bool int

int box *(***WINDOW** **win,* **char** *vert,* **char** *hor);*

Arguments

win
Specifies the address of the window.

vert
Specifies the character for the vertical edges of the window.

hor
Specifies the character for the horizontal edges of the window.

Description

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.

Return Values

0	Indicates an error.
1	Indicates success.

brk

The **brk** function determines the lowest virtual address that is not used with the program.

Format

#include *stdlib*

void *brk *(***unsigned long int** *addr);*

Arguments

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.

Description

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.

When a program is executed, the break address is set to the highest location defined by the program and data storage areas. Consequently, **brk** is needed only by programs that have growing data areas.

Return Values

n	Indicates the break address (the address of an object of type **char**).
–1	Indicates that the program requests too much memory.

bsearch

The **bsearch** function performs a binary search. It searches an array of sorted objects for a specified object.

Format

#include *stdlib*

void *bsearch *(***const void** **key,* **const void** **base,* **size_t** *nmemb,* **size_t** *size,* **int** *(*compar) (***const void** **,* **const void** **));*

Arguments

key
Is a pointer to the object to be sought in the array. This pointer should be of type pointer-to-object and cast to type pointer-to-character.

base
Is a pointer to the initial member of the array. This pointer should be of type pointer-to-object and cast to type pointer-to-character.

nmemb
Is the number of objects in the array.

size
Is the size of an object, in bytes.

compar
Is a pointer to the comparison function.

Description

The array must first be 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 0.

It is not necessary for the comparison function (**compar**) to compare every byte in the array. Therefore, the objects in the array can contain arbitrary data in addition to the data being compared.

Since it is declared as type pointer-to-void, the value returned must be cast or assigned into type pointer-to-object.

Return Values

x	Indicates a pointer to the matching member of the array or a null pointer if no match is found.
NULL	Indicates that the key cannot be found in the array.

Example

```
#include stdio
#include stdlib

#define SSIZE 30

extern int time();
int trand();
extern int compare();

int array[SSIZE] = 0;           /* The array to sort */
int stmp = 0;
int lcnt = LOOPCOUNT;           /* Number of times to go around */

void *bsearch (const void *key, const void *base, int nmemb, int elt_size,
              int (*bscmp)() );
```

```
main()
{
register int i;
int success_count =0;
volatile int j;
int *rkey;

 /* sort array */
 qsort(array, SSIZE, sizeof(array[0]), &compare);

 for (i=0; i<SSIZE-1; i++)
 {
  rkey = bsearch( (array + i), array, SSIZE, sizeof(array[0]), &compare);
  if ( &array[i] != rkey)
  {
   printf("Not in array, array element %d\n",i);
   break;
  }
  else
   ++found;
 }

}

/* compare routine  */
compare(a,b)
int *a, *b;
{
 ccomp++;
 return (*a - *b);
}
```

cabs

The **cabs** function computes the Euclidean distance between two points as the square root of their respective squares. The **cabs** return is as follows:

```
sqrt(x*x + y*y)
```

Format

#include *math*

double cabs *(***cabs_t** *z);*

Description

The type **cabs_t** is defined in the standard include module *math.h* as follows:

```
typedef struct {double x,y;} cabs_t;
```

calloc

The **calloc** function allocates an area of memory.

Format

#include *stdlib*

void *calloc *(***size_t** *number,* **size_t** *size);*

Arguments

number
Specifies the number of items to be allocated.

size
Is the size of each item.

Description

The **calloc** function initializes the items to 0.

See also **malloc** and **realloc** in this section.

Return Values

0	Indicates an inability to allocate the space.
n	Indicates the address of the first byte, which is aligned on an octaword boundary.

ceil

The **ceil** function returns (as a **double**) the smallest integer that is greater than or equal to its argument.

Format

#include *math*

double ceil *(***double** *x);*

Arguments

x
Is a real value.

cfree

The **cfree** function makes available for reallocation the area allocated by a previous **calloc**, **malloc**, or **realloc** call.

Format

#include *stdlib*

void cfree *(***void** **ptr);*

Arguments

ptr
Is the address returned by a previous call to **malloc**, **calloc**, or **realloc**.

Description

The contents of the deallocated area are unchanged.

In VAX C, the **free** and **cfree** functions have the same function. However, for compatibility with other C implementations, use **free** with **malloc** or **realloc**, and **cfree** with **calloc**.

See also **free** in this section.

chdir

The **chdir** function changes the default directory.

Format

#include *stdlib*

int chdir *(***char** **dir_spec);*

Arguments

dir_spec
Is a NUL-terminated character string naming a directory in either a VMS or UNIX-style specification.

Description

If you call the **chdir** function 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, call **chdir** from SUPERVISOR, EXECUTIVE, or KERNEL mode.

Return Values

0	Indicates that the directory is successfully changed to the given name.
–1	Indicates that the change attempt has failed.

chmod

The **chmod** function changes the file protection of a file.

Format

#include *stdlib*

int chmod (**char** **file_spec*, **unsigned int** *mode*);

Arguments

file_spec
Is the name of a VMS or UNIX-style file specification.

mode
Is a file protection. Modes are constructed by performing a bitwise OR on any of the values shown in Table REF–2.

Table REF–2: File Protection Values and their Meanings

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

(continued on next page)

Table REF–2 (Cont.): File Protection Values and their Meanings

Value	Privilege
0001	WORLD:EXECUTE

When you supply a mode argument of 0, the **chmod** function gives the file the user's default file protection.

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

Description

You must have a WRITE privilege for the file specified to change the mode.

Return Values

0	Indicates that the mode is successfully changed.
–1	Indicates that the change attempt has failed.

chown

The **chown** function changes the owner User Identification Code (UIC) of the file.

Format

#include *stdlib*

int chown *(***char** **file_spec,*
unsigned int *owner,* **unsigned int** *group);*

Arguments

file_spec
Is the address of an ASCII file name.

owner
Is the owner name.

group
Is the group name.

Return Values

0	Indicates failure.
–1	Indicates success.

[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 **clear** macro acts on the stdscr window.

Format

#include *curses*

clear()

int wclear (**WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Return Values

ERR	Indicates an error.
1	Indicates success.

clearerr

The **clearerr** macro resets the error and end-of-file indications for a file (so that **ferror** and **feof** will not return a nonzero value).

Format

#include *stdio*

void clearerr *(***FILE** **file_ptr);*

Arguments

file_ptr
Is a file pointer.

Description

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

clearok

The **clearok** macro sets the clear flag for the window.

Format

#include *curses*

#define bool int

clearok **(WINDOW** **win,* **bool** *boolf);*

Arguments

win
Is the entire size of the terminal screen. You can use the windows stdscr and curscr with **clearok**.

boolf
Is a Boolean value of TRUE or FALSE. If the argument 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.

Description

Unlike the **clear** macro, the **clearok** macro does not alter the contents of the window. If the win argument 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.

clock

The **clock** function determines the CPU time (in 10-millisecond units) 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**.

Format

#include *time*

clock_t clock *(***void***);*

Description

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.

Return Values

n	Indicates the processor time used.
–1	Indicates that the processor time used is not available.

close

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

Format

#include *unixio*

int close *(***int** *file_desc);*

Arguments

file_desc
Is a file descriptor.

Description

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

Return Values

0	Indicates that the file is properly closed.
–1	Indicates that the file descriptor is undefined or an error occurred while the file was being closed (for example, if the buffered data cannot be written out).

Example

```
#include stdio.h
int fd;
    .
    .
    .
fd = open ("student.dat", 1);
    .
    .
    .
close(fd);
```

[w]clrattr

The **clrattr** macro and the **wclrattr** function deactivate the video display attribute attr within the window. The **clrattr** macro acts on the stdscr window.

Format

#include *curses*

clrattr *(attr);*

int wclrattr *(***WINDOW** **win,* **int** *attr);*

Arguments

win
Is a pointer to the window.

attr
Are video display attributes that can be blinking, boldface, reverse video, and underlining, and are represented by the defined constants _BLINK, _BOLD, _REVERSE, and _UNDERLINE. To clear multiple attributes, separate them with a bitwise OR operator (|) as follows:

```
clrattr(_BLINK | _UNDERLINE);
```

Description

The **clrattr** macro and the **wclrattr** function are VAX C specific and are not portable.

Return Values

1	Indicates success.
ERR	Indicates an error.

[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 **clrtobot** macro acts on the stdscr window.

Format

#include *curses*

clrtobot()

int wclrtobot *(***WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Return Values

1	Indicates success.
ERR	Indicates an error.

[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 **wclrtoeol** macro acts on the stdscr window.

Format

#include *curses*

clrtoeol( )

int wclrtoeol *(***WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Return Values

1	Indicates success.
ERR	Indicates an error.

cos

The **cos** function returns the cosine of its radian argument.

Format

#include *math*

double cos (**double** *x*);

Arguments

x
Is a radian expressed as a real value.

Description

If you use the *math* include module to declare **cos**, VAX C transforms the call into a direct call to MTH$DCOS_RT or MTH$GCOS_RT, depending on whether or not /G_FLOAT is specified on the CC command line.

cosh

The **cosh** function returns the hyperbolic cosine of its argument.

Format

#include *math*

double cosh *(***double** *x);*

Arguments

x
Is a real value.

creat

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

Format

#include *unixio*

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

Arguments

file_spec
Is a NUL-terminated string containing any valid file specification.

mode
Is 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.

You can construct modes by using the bitwise OR operator (|) to create 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 0, **creat** gives your default file protection.

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

...

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

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

Keyword is an RMS field in the file access block (FAB) or record access block (RAB); 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 REF–3 describes RMS keywords and values.

Table REF–3: RMS Valid Keywords and Values

Keyword	Value	Description
"alq = n"	decimal	Allocation quantity
"bls = n"	decimal	Block size
"ctx = bin"	string	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 = stm"	string	Force stream mode access
"deq = n"	decimal	Default extension quantity
"dna = filespec"	string	Default file-name string
"fop = val, val , . . . "		File-processing options:

(continued on next page)

Table REF–3 (Cont.): RMS Valid Keywords and Values

Keyword	Value	Description
	ctg	Contiguous
	cbt	Contiguous-best-try
	dlt	Delete file on close
	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
	rck	Read check compare operation
	wck	Write check compare operation
	mxv	Maximize version number
	rwo	Rewind file on open
	pos	Current position
	rwc	Rewind file on close
"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:

(continued on next page)

Table REF–3 (Cont.): RMS Valid Keywords and Values

Keyword	Value	Description
	fix	Fixed-length record format
	stm	RMS 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
	RAH	Read ahead
	WBH	Write behind
"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 on the type of access you want.

Description

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

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

Return Values

integer	Indicates a file descriptor.
–1	Indicates errors including protection violations, undefined directories, and conflicting file attributes.

[no]crmode

In the UNIX system environment, the **crmode** and **nocrmode** macros set and unset the terminal from cbreak mode. This mode of single character input is only supported with the Curses input routine **getch**. It also applies to any of the UNIX I/O, Terminal I/O, or Standard I/O routines.

Format

#include *curses*

crmode( )

nocrmode( )

Example

```
/* Exercise cbreak */

# include curses

main ()
{
 WINDOW  *win1;
 char    vert = '.', hor = '.', str[80];

 /* Initialize standard screen, turn echo off  */

 initscr ();
      noecho ();

 /* Define a user window  */

 win1 = newwin (22, 78, 1, 1);

 /* Turn on reverse video and draw a box on border */

 setattr (_REVERSE);
      box (stdscr, vert, hor);

 mvwaddstr (win1, 2, 2, "Test cbreak input");
 refresh ();
      wrefresh (win1);

 /*  Set cbreak do some input and output it */
```

```
 crmode();
 getstr (str);
      nocrmode();  /* Now turn off cbreak */

      mvwaddstr (win1, 5, 5, str);
 mvwaddstr (win1, 7, 7, "Type something to clear the screen");
      wrefresh (win1);

 /* Get another character then delete the window */

 getch ();
 wclear (win1);

 /* Redraw the standard window  */
      touchwin (stdscr);

 endwin ();
}
```

ctermid

The **ctermid** function returns a character string giving the equivalence string of SYS$COMMAND. This is the name of the controlling terminal.

Format

#include *stdlib*

char *ctermid **(char** **str);*

Arguments

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).

Return Values

pointer	Points to a character string.

ctime

The **ctime** function converts a time in seconds, since 00:00:00 January 1, 1970, to an ASCII string in the form generated by the **asctime** function.

Format

#include *time*

char *ctime **(const time_t** ***bintim);***

Arguments

bintim
Is a pointer to the time value to be converted.

Description

Successive calls to the **ctime** or **asctime** function 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
```

Return Values

pointer	Points to the 26-character ASCII string.

cuserid

The **cuserid** function returns a pointer to a character string containing the name of the user initiating the current process.

Format

#include *stdio*

char *cuserid **(char** **str);*

Arguments

str
If this argument 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 a null string, the function returns NULL.

Return Values

pointer	Points to a string.

[w]delch

The **delch** macro and the **wdelch** function delete the character on the specified window at the current position of the cursor.

Format

#include *curses*

delch( )

int wdelch **(WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Description

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.

Return Values

1	Indicates success.
ERR	Indicates an error.

delete

The **delete** function causes a file to be deleted.

Format

#include *stdio*

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

Arguments

file_spec
Is a pointer to the string that is a VMS or UNIX-style file specification.

Description

If you specify a directory in the file name and it is a search list that contains an error, VAX C interprets it as a file error.

The **remove** and **delete** functions are the same in the VAX C RTL.

Return Values

–1	Indicates that the operation has failed.
0	Indicates success.

[w]deleteln

The **deleteln** macro and the **wdeleteln** function delete the line at the current position of the cursor. The **deleteln** macro acts on the stdscr window.

Format

#include *curses*

deleteln( )

int wdeleteln **(WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Description

Every line below the deleted line moves up, and the bottom line becomes blank. The current (y,x) coordinates of the cursor remain unchanged.

Return Values

1	Indicates success.
ERR	Indicates an error.

delwin

The **delwin** function deletes the specified window from memory.

Format

#include *curses*

#define bool int

int delwin (**WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Description

If the window being deleted contains a subwindow, the subwindow is invalidated. Delete subwindows before deleting the underlying window. The **delwin** function refreshes all covered windows of the deleted window.

Return Values

1	Indicates success.
ERR	Indicates an error.

difftime

The **difftime** function computes the difference, in seconds, between the two times specified by the time1 and time2 arguments.

Format

#include *time*

double difftime (**time_t** *time2,* **time_t** *time1);*

Arguments

time2
Is of type **time_t**, which is defined in the standard include module *time.h*.

time1
Is of type **time_t**, which is defined in the standard include module *time.h*.

Return Values

n	Indicates time2—time1 in seconds expressed as a **double**.

div

The **div** function returns the quotient and the remainder after the division of its arguments.

Format

#include *stdlib*

div_t div *(***int** *numer,* **int** *denom)* **;**

Arguments

numer
Is a numerator of type **int**.

denom
Is a denominator of type **int**.

Description

The type **div_t** is defined in the standard include module *stdlib* as follows:

```
struct DIV_T
          {
           int quot
           int rem;
          };
typdef struct DIV_T div_t;
```

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**.

Format

#include *unixio*

int dup (**int** *file_desc1);*

int dup2 (**int** *file_desc1,* **int** *file_desc2);*

Arguments

file_desc1
Is the file descriptor being duplicated.

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

Description

The **dup2** function causes its second argument to refer to the same file as its first argument.

The argument file_desc1 is invalid if it does not describe an open file; file_desc2 is invalid if the new file descriptor cannot be allocated. If file_desc2 is connected to an open file, that file is closed.

Return Values

n	Indicates the new file descriptor.
–1	Indicates that there are invalid arguments.

[no]echo

The **echo** and **noecho** macros set the terminal so that characters may or may not be echoed on the terminal screen. This mode of single-character input is only supported with Curses.

Format

#include *curses*

echo( )

noecho( )

Description

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.

ecvt

The **ecvt** function 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.

Format

#include *unixlib*

char *ecvt (**double** *value,* **int** *ndigit,* **int** **decpt,* **int** **sign);*

Arguments

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

ndigit
Is 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, (the spaces being filled with zeros). A 0 value means that the decimal point is immediately to the left of the first digit in the returned string.

sign
Contains an integer 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 0 to the address specified by the argument sign.

Description

Repeated calls to the **ecvt** function overwrite any existing string.

See also **gcvt** and **fcvt** in this section.

Return Values

x	Is the value of the converted string.

endwin

The **endwin** function clears the terminal screen and frees any virtual memory allocated to Curses data structures.

Format

#include *curses*

#define bool int

void endwin *(***void***);*

Description

You must call the **endwin** function before exiting to restore the previous environment of the terminal screen.

[w]erase

The **erase** macro and the **werase** function erase the window by "painting" it with blanks. The **erase** macro acts on the stdscr window.

Format

#include *curses*

erase( )

int werase (**WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Description

Both the **erase** macro and the **werase** function 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).

Return Values

1	Indicates success.
ERR	Indicates an error.

execl

The **execl** function passes the name of an image to be activated in a child process.

Format

#include *processes*

int execl *(***char** **file_spec,* **char** **argn, . . .);*

Arguments

file_spec
Is the file specification (full) of a new image to be activated in the child process.

argn
Represents a sequence of pointers to NUL-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).

. . .
Represents a sequence of pointers to strings. At least one pointer must exist to terminate the list. This pointer may be the null pointer.

Description

To understand how the exec functions operate, consider how the VMS system 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]) contains the name of the image. The arguments are placed in subsequent elements of the array. The last element of the array is always the null pointer.

execl

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 as follows:

- Arguments can be passed in separate character strings.
- The environment can be explicitly taken from the parent's environment variable.

See also **execle**, **execlp**, **execv**, **execve**, and **execvp** in this section.

Return Values

-1	Indicates failure.

execle

The **execle** function passes the name of an image to be activated in a child process.

Format

#include *processes*

int execle *(***char** **file_spec,* **char** **argn, . . . ,* **char** **envp[ ]);*

Arguments

file_spec
Is the full file specification of a new image to be activated in the child process.

argn
Represents a sequence of pointers to NUL-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
Is an array of strings that specifies the program's environment. Each string in argument envp has the following form:

```
name = value
```

The name can be one of the following names and the value is a NUL-terminated string to be associated with the name:

- HOME—Your 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
Is an array of pointers to NUL-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.

Description

To understand how the exec functions operate, consider how the VMS system 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]) contains the name of the image. The arguments are placed in subsequent elements of the array. The last element of the array is 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 how they pass arguments and environment information to the child as follows:

- Arguments can be passed in separate character strings.
- The environment can be explicitly passed in an array.

See also **execl**, **execlp**, **execv**, **execve**, and **execvp** in this section.

Return Values

-1	Indicates failure.

execlp

The **execlp** function passes the name of an image to be activated in a child process.

Format

#include *processes*

int execlp *(***char** **file_name,* **char** **argn, . . .);*

Arguments

file_name
Is 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 searching the environment name VAXC$PATH.

argn
Represents a sequence of pointers to NUL-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).

. . .
Represents a sequence of pointers to strings. At least one pointer must exist to terminate the list. This pointer may be the null pointer.

Description

To understand how the exec functions operate, consider how the VMS system 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]) contains the name of the image. The arguments are placed in subsequent elements of the array. The last element of the array is 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 how they pass arguments and environment information to the child as follows:

- Arguments can be passed in separate character strings.
- The environment can be explicitly passed in an array (**execle** and **execve**) or taken from the parent's environment variable (**execl** and **execv**).

See also **execle**, **execl**, **execv**, **execve**, and **execvp** in this section.

Return Values

-1	Indicates failure.

execv

The **execv** function passes the name of an image to be activated in a child process.

Format

#include *processes*

int execv (**char** **file_spec,* **char** **argv[]);*

Arguments

file_spec
Is the full file specification of a new image to be activated in the child process.

argv
Is an array of pointers to NUL-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.

Description

To understand how the exec functions operate, consider how the VMS operating system 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]) contains the name of the image. The arguments are placed in subsequent elements of the array. The last element of the array is 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 how they pass arguments and environment information to the child as follows:

- Arguments can be passed in an array of character strings.
- The environment can be explicitly taken from the parent's environment variable.

See also **execle**, **execl**, **execlp**, **execve**, and **execvp** in this section.

Return Values

-1	Indicates failure.

execve

The **execve** function passes the name of an image to be activated in a child process.

Format

#include *processes*

int execve **(char** **file_spec,* **char** **argv[],* **char** **envp[]);*

Arguments

file_spec
Is the full file specification of a new image to be activated in the child process.

argv
Is an array of pointers to NUL-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.

envp
Is an array of strings that specifies the program's environment. Each string in argument envp has the following form:

```
name = value
```

The name can be one of the following names and the value is a NUL-terminated string to be associated with the name:

- HOME—Your 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.

Description

To understand how the exec functions operate, consider how the VMS operating system 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]) contains the name of the image. The arguments are placed in subsequent elements of the array. The last element of the array is the null pointer.

An exec function calls a child process in the same way that the run-time system calls any 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 how they pass arguments and environment information to the child as follows:

- Arguments can be passed in an array of character strings.
- The environment can be explicitly passed in an array.

See also **execle**, **execlp**, **execv**, **execl**, and **execvp** in this section.

Return Values

-1	Indicates failure.

execvp

The **execvp** function passes the name of an image to be activated in a child process.

Format

#include *processes*

int execvp **(char** **file_name,* **char** **argv[]);*

Arguments

file_name
Is 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 searching the environment name VAXC$PATH.

argv
Is an array of pointers to NUL-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.

Description

To understand how the exec functions operate, consider how the VMS operating system 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]) contains the name of the image. The arguments are placed in subsequent elements of the array. The last element of the array is the null pointer.

An exec function calls a child process in the same way that the run-time system calls any 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 how they pass arguments and environment information to the child as follows:

- 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** and **execve**) or taken from the parent's environment variable (**execl** and **execv**).

See also **execle**, **execlp**, **execv**, **execl**, and **execve** in this section.

Return Values

-1	Indicates failure.

exit, _exit

The **exit** and **_exit** functions terminate the process from which they are called.

Format

#include *stdlib*

void exit *(***int** *status);*

void _exit *(***int** *status);*

Arguments

status
Corresponds with an errno value. The errno values are defined in the *errno* definition module. A status value of 0 is translated to the VMS SS$_ NORMAL status code to return the VMS success value. Any other status value is left the same. The status value is passed to the parent process.

Description

If the program is invoked by the DIGITAL Command Language (DCL), the status is interpreted by DCL and a message is displayed. The two functions are identical; the **_exit** function is retained for reasons of compatibility with previous versions of VAX C.

exp

The **exp** function returns the base e raised to the power of the argument.

Format

#include *math*

double exp (**double** *x*);

Arguments

x
Is a real value.

Description

If an overflow occurs, the **exp** function returns the largest possible floating-point value and sets errno to ERANGE. The constant HUGE_VAL in the *math* definition file is defined to be the largest possible floating-point value.

Return Values

HUGE_VAL	Indicates that an overflow has occurred.

fabs

The **fabs** function returns the absolute value of a floating-point value.

Format

#include *math*

double fabs (**double** *x*);

Arguments

x
Is a real value.

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.

Format

#include *stdio*

int fclose (**FILE** **file_ptr*);

Arguments

file_ptr
Is a pointer to the file to be closed.

Description

When a program terminates normally, the **fclose** function is automatically called for all open files.

Return Values

0	Indicates success.
EOF	Indicates that the buffered data cannot be written to the file, or that the file control block is not associated with an open file. EOF is a preprocessor constant defined in the **#include** module *stdio*.

fcvt

The **fcvt** function converts its argument to a NUL-terminated string of ASCII digits and returns the address of the string.

Format

#include *unixlib*

char *fcvt *(***double** *value,* **int** *ndigit,* **int** **decpt,* **int** **sign);*

Arguments

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

ndigit
Is the number of ASCII digits after the decimal point to be used in the converted string.

decpt
Contains the position of the decimal point relative to the first character in the returned string. The returned string does not contain the actual decimal point. A negative **int** value means that the decimal point is decpt number of spaces to the left of the returned digits (the spaces are filled with zeros). A 0 value means that the decimal point is immediately to the left of the first digit in the returned string.

sign
Contains an integer value that indicates whether the argument value is positive or negative. If the value is negative, the **fcvt** function places a nonzero value at the address specified by the argument sign. Otherwise, the functions assign 0 to the address specified by the argument sign.

Description

Repeated calls to the **fcvt** function overwrite any existing string.

See also **gcvt** and **ecvt** in this section.

Return Values

x	Is a pointer to the converted string.

fdopen

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

Format

#include *stdio*

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

Arguments

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

a_mode
Is 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 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 the 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 first at the 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 first positioned at the end-of-file (writing). 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.

Description

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, see Chapter 1.

Return Values

pointer	Indicates that the operation has succeeded.
0	Indicates that an error has occurred.

feof

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

Format

#include *stdio*

int feof (**FILE** **file_ptr);*

Arguments

file_ptr
Is a file pointer.

Description

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

Return Values

nonzero integer	Indicates that the end-of-file has been reached.
0	Indicates that the end-of-file has not been reached.

ferror

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

Format

#include *stdio*

int ferror (**FILE** **file_ptr*);

Arguments

file_ptr
Is a file pointer.

Description

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

Return Values

nonzero integer	Indicates that an error has occurred.
0	Indicates success.

fflush

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

Format

#include *stdio*

int fflush (**FILE** **file_ptr);*

Arguments

file_ptr
Is a file pointer.

Description

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

The **fflush** function flushes the C RTL buffers. However, RMS has its own buffers. The **fflush** function does not guarantee that the file will be written to disk.

Return Values

0	Indicates that the operation is successful.
EOF	Indicates that the buffered data cannot be written to the file, or that the file control block is not associated with an output file (EOF is a preprocessor constant defined in the *stdio* definition module).

fgetc

The **fgetc** function returns characters from a specified file.

Format

#include *stdio*

int fgetc (**FILE** **file_ptr);*

Arguments

file_ptr
Is a pointer to the file to be accessed.

Description

The **fgetc** function and the **getc** macro are the same functions.

The file is left-positioned after the returned character, and the next **getc** call takes the character from that position. The value returned is a **char** converted to an **int**.

See the **getc** macro in this section for more information.

Return Values

x	Next character from the specified file.
EOF	Indicates the end-of-file or an error.

fgetname

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

Format

#include *stdio*

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

Arguments

file_ptr
Is a file pointer.

buffer
Is 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 **fgetname** function returns the file specification in VMS format. If you specify 0, **fgetname** returns the file specification in UNIX-style. If you do not specify this argument, **fgetname** returns the file name according to your current command language interpreter. For more information about UNIX-style file specifications, see Section 1.2.1.

Description

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

Return Values

n	Indicates the address of the buffer.
0	When an error occurs, **fgetname** returns 0.

fgetpos

The **fgetpos** function stores the current value of the file position indicator for the stream pointed to by the stream into the object pointed to by pos.

Format

#include *stdio*

int fgetpos (**FILE** **stream,* **fpos_t** **pos);*

Arguments

stream
Is a file pointer.

pos
Is a pointer to an implementation-defined structure. The **fgetpos** function fills this structure with information that can be used on subsequent calls to **fsetpos**.

Example

```
#include stdio

main()
{
FILE *fp;
int stat,i;
int character;
unsigned char ch, c_ptr[130], d_ptr[130];
fpos_t posit;

 /* Open a file for writing  */

 if ((fp = fopen ("file.dat","w+") ) == NULL)
     {
  perror ("open");
  exit(1);
 }

 /* Get the beginning position in the file  */
```

```
    if(fgetpos(fp, &posit) != 0)
perror ("fgetpos");

/* Write some data to the file */

    if (fprintf(fp,"this is a test\n") ==0)
{
 perror ("fprintf");
 exit(1);
}

/* Set the file position back to the beginning */

      if(fsetpos(fp, &posit) != 0)
 perror ("fsetpos");

fgets(c_ptr,130,fp);
     puts(c_ptr);  /* Should be "this is a test"  */

/* Close the file */

    if (fclose (fp) != 0)
{
 perror ("close");
 exit(1);
}

}
```

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.

Format

#include *stdio*

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

Arguments

str
Is the address where the fetched string will be stored.

maxchar
Specifies the maximum number of characters to fetch.

file_ptr
Is a file pointer.

Description

The **fgets** 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.

Return Values

x	Indicates the address of the first character in the line.
NULL	Indicates the end-of-file or an error. NULL is defined in the *stdio* definition module to be the null pointer value.

Example

```
#include stdio

main()
{
FILE *fp;
int stat,i;
int character;
unsigned char ch, c_ptr[130], d_ptr[130];

        /* open a file with some data -"THIS IS A TEST"   */

 if ((fp = fopen ("file.dat","r+") ) == NULL)
 {
 perror ("open error"),exit(1);

      fgets(c_ptr,130,fp);

      puts(c_ptr);  /* display what fgets got.  */

      close(fp);
}
```

fileno

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

Format

#include *stdio*

int fileno (**FILE** **file_ptr);*

Arguments

file_ptr
Is a file pointer.

Description

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

floor

The **floor** function returns (as a **double**) the largest integer that is less than or equal to its argument.

Format

#include *math*

double floor (**double** *x*);

Arguments

x
Is a real value.

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.

Format

#include *math*

double fmod *(***double** *x,* **double** *y);*

Arguments

x
Is a real value.

y
Is a real value.

Return Values

0	Indicates that y is 0.
x	Indicates 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.

fopen

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

Format

#include *stdio*

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

Arguments

file_spec
Is a character string containing a valid file specification.

a_mode
Is 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 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 the 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 first at the 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 first positioned at the end-of-file (writing). 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.

...
Represents optional file attribute arguments. The file attribute arguments are the same as those used in the **creat** function. For more information, see the **creat** function.

Description

If you specify a directory in the file name and it is a search list that contains an error, VAX C interprets it as a file open error.

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

Return Values

NULL	Indicates an error. The constant NULL is defined in the *stdio* definition module 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

fprintf

The **fprintf** function performs formatted output to a specified file.

Format

#include *stdio*

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

Arguments

file_ptr
Is a pointer to the file that you direct output to.

format_spec
Contains characters to be written literally to the output or converted as specified in the argument. For more information on conversion characters, see Chapter 2.

. . .
Are 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 left-to-right order.

Description

An example of a conversion specification is as follows:

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

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

Sample output (to the stdout file) 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, see Chapter 2.

Return Values

x	Indicates the number of characters written.
-1	Indicates that an error has occurred.

fputc

The **fputc** function writes characters to a specified file.

Format

#include *stdio*

int fputc *(***int** *character,* **FILE** **file_ptr);*

Arguments

character
Is an object of type **int**.

file_ptr
Is a file pointer.

Description

The **fputc** function writes a single character to a file and returns the character. The file pointer is left-positioned after the character. In VAX C, **fputc** and **putc** perform the same function.

Return Values

EOF	Indicates that an output error has occurred. EOF is defined in the *stdio* definition module.
character	Indicates success.

fputs

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

Format

#include *stdio*

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

Arguments

str
Is a pointer to a character string.

file_ptr
Is a file pointer.

Return Values

Nonnegative number	Indicates success.
EOF	Indicates an error.

fread

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

Format

#include *stdio*

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

Arguments

ptr
Is a pointer to the location, within memory, where you 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
Is the size of the items being read, in bytes.

number_items
Is the number of items to be read.

file_ptr
Is a pointer that indicates the file from which the items are to be read.

Description

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. You must also specify the size of an item, in bytes.

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

Return Values

n	Indicates the number of items read.
0	Indicates the end-of-file or an error.

free

The **free** function makes available for reallocation the area allocated by a previous **calloc**, **malloc**, or **realloc** call.

Format

#include *stdlib*

void free *(***void** **ptr);*

Arguments

ptr
Is the address returned by a previous call to **malloc**, **calloc**, or **realloc**.

Description

The contents of the deallocated area are unchanged. However, for compatibility with other C implementations, you should use **free** with **malloc** or **realloc**, and **cfree** with **calloc**.

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.

Format

#include *stdio*

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

Arguments

file_spec
Is a pointer to a string that contains a valid VMS or UNIX-style file specification. After the function call, the given file pointer is associated with this file.

a_mode
Is an access mode indicator. The **fdopen** function in this section describes a_mode.

file_ptr
Is a file pointer.

. . .
Represents optional file attribute arguments. The file attribute arguments are the same as those used in the **creat** function.

Description

Use **freopen** to associate one of the predefined names stdin, stdout, or stderr with a file. For more information about these predefined names, see Chapter 2.

Return Values

file_ptr	The file pointer, if **freopen** is successful.
NULL	Indicates that an error has occurred. The constant NULL is defined in the *stdio* definition module to be the null pointer value.

frexp

The **frexp** function calculates the fractional and exponent parts of a **double** value.

Format

#include *math*

double frexp *(***double** *value,* **int** **eptr);*

Arguments

value
Is an object of type **double**.

eptr
Is a pointer to an **int**, to which **frexp** returns the exponent.

Description

The **frexp** function converts *value* to the following form:

$$value = fraction * (2^{exp})$$

The fractional part is returned as the return value. The exponent is placed in the integer variable pointed to by *eptr*.

Example

```
main ()
{
   double val = 16.0, fraction;
   int exp;

   fraction = frexp(val, &exp);
   printf("fraction = %f\n",fraction);
   printf("exp = %d\n",exp);

}
```

In this example, **frexp** converts the value 16 to $.5*2^5$. The example produces the following output:

```
fraction = 0.500000
exp = 5
```

Return Values

x — The fractional part of the **double** value.

fscanf

The **fscanf** function performs formatted input from a specified file.

Format

#include *stdio*

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

Arguments

file_ptr
Is a pointer to the file that provides input text.

format_spec
Contains characters to be taken literally from the input or converted and placed in memory at the specified . . . argument. For more information on conversion characters, see Chapter 2.

. . .
Are optional expressions whose results correspond to conversion specifications given in the format specification. If no conversion specifications are given, you can omit the input pointers. 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 pointers. Conversion specifications are matched to input sources in left-to-right order.

Description

An example of a conversion specification is as follows:

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

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

Consider a file, designated by stdin, with the following contents:

```
4 17
```

Sample input from the previous example will then be as follows:

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

For a complete description of the format specification and the input pointers, see Chapter 2.

Return Values

x	Indicates the number of successfully matched and assigned input items.
EOF	Indicates that the end-of-file or the end of the string has been encountered. EOF is a preprocessor constant defined in the *stdio* definition module.

fseek

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

Format

#include *stdio*

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

Arguments

file_ptr
Is a file pointer.

offset
Is the offset specified, in bytes.

direction
Is 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).

Description

Direct the **fseek** function 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.

Return Values

0	Indicates successful seeks.
EOF	Indicates improper seeks. EOF is a preprocessor constant defined in the *stdio* definition module.

fsetpos

The **fsetpos** function sets the file position indicator for the stream according to the value of the object pointed to by pos.

Format

#include *stdio*

int fsetpos (**FILE** **stream,* **fpos_t** **pos);*

Arguments

stream
Is a file pointer.

pos
Is a pointer to an implementation-defined structure.

Description

Call the **fgetpos** function before using the **fsetpos** function.

fstat

The **fstat** function accesses information about the file descriptor or the file specification.

Format

#include *stat*

int fstat *(***int** *file_desc,* **stat_t** **buffer);*

Arguments

file_desc
Is a file descriptor.

buffer
Is a pointer to a structure of type **stat_t**, which is defined in the *stat* definition module. The argument receives information about that particular file. The members of the structure pointed to by buffer are as follows:

Member	Type	Definition
st_dev	unsigned	Pointer to a physical device name
st_ino[3]	unsigned short	Three words to receive the 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 0
st_size	unsigned	File size, in bytes
st_atime	unsigned	File access time; always the same as st_mtime

Member	Type	Definition
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 st_mode, structure member, is the status information mode and is defined in the *stat* definition module. The st_mode bits are listed 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

Description

The **fstat** function does not work on remote network files.

Return Values

0	Indicates successful completion.
–1	Indicates that there are errors.
–2	Indicates a protection violation.

ftell

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

Format

#include *stdio*

long int ftell *(***FILE** **file_ptr);*

Arguments

file_ptr
Is a file pointer.

Description

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, which repositions the file to where it was when **ftell** was called.

Return Values

EOF	Indicates that an error has occurred.

ftime

The **ftime** function returns the elapsed time since 00:00:00, January 1, 1970, in the structure pointed at by timeptr.

Format

#include *time*

void ftime (**timeb_t** **timeptr);*

Arguments

timeptr
Is a pointer to the structure **timeb_t**.

Description

The typedef **timeb_t** refers to a structure defined in the standard include module *time.h* as follows:

```
typedef struct timeb
   {
      time_t         time;
      unsigned short millitm;
      short          timezone;
      short          dstflag;
   };
```

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 0.

fwrite

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

Format

#include *stdio*

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

Arguments

ptr
Is a pointer to the memory location from which information is being written.

size_of_item
Is the size of the items being written, in bytes.

number_items
Is the number of items being written.

file_ptr
Is a file pointer that indicates the file to which the items are being written.

Description

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

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

Return Values

x	Indicates the number of items written. The number of records written depends upon the maximum record size of the file.

gcvt

The **gcvt** function 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.

Format

#include *unixlib*

char *gcvt *(***double** *value,* **int** *ndigit,* **char** **buffer);*

Arguments

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

ndigit
Is the number of ASCII digits to use in the converted string. If ndigit is less than 6, the value of 6 is used.

buffer
Is a storage location to hold the converted string.

Description

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

Repeated calls to this function overwrite any existing string.

See also **fcvt** and **ecvt** in this section.

Return Values

x	Indicates the address of the returned string.

getc

The **getc** macro returns characters from a specified file.

Format

#include *stdio*

int getc (**FILE** **file_ptr);*

Arguments

file_ptr
Is a pointer to the file to be accessed.

Description

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

```
fgetc(file_ptr)
```

Return Values

x	Indicates that the next character is an **int** from the specified file.
EOF	Indicates the end-of-file or an error.

[w]getch

The **getch** macro and the **wgetch** function get a character from the terminal screen and echo it on the specified window.

Format

#include *curses*

getch()

char wgetch **(WINDOW** ***win);***

Arguments

win
Is a pointer to the window.

Description

The **getch** macro and the **wgetch** function refresh the specified window before fetching a character. For more information, see the **scrollok** function in this section.

Return Values

x	Specifies the returned character.
ERR	Indicates that the function makes the screen scroll illegally.

getchar

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

Format

#include *stdio*

int getchar *(***void***);*

Description

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

Return Values

x	Is the next character from stdin.
EOF	Indicates the end-of-file or an error.

getcwd

The **getcwd** function returns a pointer to the file specification for the current working directory.

Format

#include *unixlib*

char *getcwd *(***char** **buffer,* **unsigned int** *size, . . .);*

Arguments

buffer
Is a pointer to a character string that is large enough to hold the directory specification.

If buffer is a null pointer, **getcwd** obtains size bytes of space using **malloc**. In this case, you can use the pointer returned by **getcwd** as the argument in a subsequent call to **free**.

size
Is the length of the directory specification to be returned.

. . .
Is an optional argument that can be either 1 or 0. If you specify 1, the **getcwd** function returns the directory specification in VMS format. If you specify 0, **getcwd** returns the directory specification (path name) in UNIX-style format. If you do not specify this argument, **getcwd** returns the file name according to your current command-language interpreter. For more information about UNIX-style directory specifications, see Section 1.2.1.

Description

If an error occurs, the **getcwd** function returns NULL with errno set to one of the following:

- ERANGE if size is not large enough
- EINVAL if size is 0
- ENOMEM if space for the returned string is not available for allocation

getegid

The **getegid** function returns, in VMS terms, the group number from the user identification code (UIC). For example, if the UIC is [313,031], 313 is the group number.

Format

#include *unixlib*

unsigned int getgid *(***void***);*

unsigned int getegid *(***void***);*

Description

In VAX C, **getgid** and **getegid** perform the same function. Both return the group number from the current UIC.

Return Values

x	Indicates the group number from the UIC.

getenv

The **getenv** function searches the environment array for the current process and returns the value associated with a specified environment name.

Format

#include *unixlib*

char *getenv **(const char** **name);*

Arguments

name
Can be one of the following values:

- HOME—Your 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

Description

In certain situations, the **getenv** function attempts 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 your environment array, **getenv** attempts to translate your argument as a logical name. All four logical name tables are searched in the standard order. If no logical names exist, **getenv** attempts to translate the argument string as a command-language interpreter (CLI) symbol; if it succeeds, it returns 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.

Return Values

x	Pointer to an array containing the translated symbol.
NULL	Indicates that the translation failed.

geteuid

The **geteuid** function returns, in VMS terms, the member number from the user identification code (UIC). For example, if the UIC is [313,031], 031 is the member number.

Format

#include *unixlib*

unsigned int geteuid *(***void***);*

Description

In VAX C, the **getuid** and **geteuid** functions both return the member number from the current UIC.

See the **getegid** or **getgid** functions in this section for the functions that return the group number.

Return Values

x	Indicates the member number from the current UIC.

getgid

The **getgid** function returns, in VMS terms, the group number from the user identification code (UIC). For example, if the UIC is [313,031], 313 is the group number.

Format

#include *unixlib*

unsigned int getgid *(***void***);*

Description

In VAX C, **getgid** and **getegid** perform the same function. Both return the group number from the current UIC. Similarly, **getuid** and **geteuid** both return the member number from the current UIC.

Return Values

x	Indicates the group number from the current UIC.

getname

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

Format

#include *unixio*

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

Arguments

file_desc
Is a file descriptor.

buffer
Is 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 UNIX-style 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 about UNIX-style file specifications, see Section 1.2.1.

Description

The **getname** function places the file specification in the area pointed to by buffer and returns that address. The area pointed to by buffer should be an array large enough to contain a fully qualified file specification (the maximum length is 256 characters).

Return Values

x	Is the address passed in the buffer argument. This indicates a successful completion.
0	Indicates an error.

getpid

The **getpid** function returns the process ID of the current process.

Format

#include *unixlib*

int getpid *(***void***);*

getppid

The **getppid** function returns the parent process ID of the calling process.

Format

#include *unixlib*

int getppid *(***void***);*

Return Values

x	Is the parent process ID.
0	Indicates that the calling process does not have a parent process.

gets

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

Format

#include *stdio*

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

Arguments

str
Is a pointer to a character string that is large enough to hold the information fetched from stdin.

Description

The newline character (\n) that ends the line is replaced by the function with an ASCII null character (\0). The function returns its argument, which is a pointer to a character string containing the acquired line.

Return Values

x	Is a pointer to the line read.
NULL	Indicates that an error has occurred or that the end-of-file was encountered before a newline was encountered.

[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 **getstr** macro works on the stdscr window.

Format

#include *curses*

getstr *(str)*

int wgetstr *(***WINDOW** **win,* **char** **str);*

Arguments

win
Is a pointer to the window.

str
Must be large enough to hold the character string fetched from the window.

Description

The **getstr** macro and the **wgetstr** function refresh the specified window before fetching a string. The newline terminator is stripped from the fetched string. For more information, see the **scrollok** macro in this section.

Return Values

1	Indicates success.
ERR	Indicates that the function makes the screen scroll illegally.

getuid

The **getuid** function returns, in VMS terms, the member number from the user identification code (UIC). For example, if the UIC is [313,031], 031 is the member number.

Format

#include *unixlib*

unsigned int getuid *(***void***);*

Description

In VAX C, **getuid** and **geteuid** perform the same function. Both return the member number from the current UIC.

Return Values

x	Indicates the member number from the current UIC.

getw

The **getw** function returns characters from a specified file.

Format

#include *stdio*

int getw *(***FILE** **file_ptr);*

Arguments

file_ptr
Is a pointer to the file to be accessed.

Description

The **getw** function returns the next four characters from the specified input file as an **int**. No conversion is performed.

Return Values

EOF	Indicates that the end-of-file was encountered during the retrieval of any of the four characters and all four characters were lost. Since EOF is an acceptable integer, use **feof** and **ferror** to check its success. EOF is a preprocessor constant defined in the **#include** module *stdio*.

getyx

The **getyx** macro puts the (y,x) coordinates of the current cursor position on win in the variables y and x.

Format

#include *curses*

getyx *(***WINDOW** **win,* **int** *y,* **int** *x);*

Arguments

win
Must be a pointer to the window.

y
Must be a valid VAX C lvalue.

x
Must be a valid VAX C lvalue.

gmtime

The **gmtime** function converts a given calendar time into a broken-down time, expressed as Greenwich Mean Time (GMT).

Format

#include *time*

struct tm *gmtime **(const time_t** ***timer);***

Arguments

timer
Is a pointer to an object of type **time_t**, which contains the calendar time.

Description

The **gmtime** function is provided to conform to the draft proposed ANSI standard for the C language. Since the VMS environment does not support GMT, this function returns a NULL.

Return Values

pointer	Is a null pointer because GMT is not available under the VMS operating system.

gsignal

The **gsignal** function generates a specified software signal. Generating a signal causes the action established by the **ssignal** function to be taken.

Format

#include *signal*

int gsignal *(***int** *sig, . . .);*

Arguments

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 REF–4 presents the various codes.

Table REF–4: SIGFPE Arithmetic Trap Signal Codes

Hardware Condition	Signal	Code
Arithmetic Traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by 0	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by 0	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP

(continued on next page)

Table REF–4 (Cont.): SIGFPE Arithmetic Trap Signal Codes

Hardware Condition	Signal	Code
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by 0 fault	SIGFPE	FPE_FLTDIV_FAULT
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 to 10, but the SIGILL codes are represented by the numbers 0 to 2. The code values are defined in the *signal* definition module.

Description

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 corresponding to the signal.

Return Values

0	Indicates a sig argument that is outside the range defined in the *signal* definition module, and the variable errno is set to EINVAL. See Chapter 4 for more information.
sig	Indicates that SIG_IGN (ignore signal) has been established as the action for the signal.
x	Indicates that **ssignal** has established an action function for the signal. That function is called, and that function's return value is returned by **gsignal**.

hypot

The **hypot** function returns the square root of the sum of the squares of two arguments. For example:

```
sqrt(x*x + y*y)
```

Format

#include *math*

double hypot *(***double** *x,* **double** *y);*

Arguments

x
Is a real value.

y
Is a real value.

[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 **inch** macro acts on the stdscr window.

Format

#include *curses*

inch()

char winch (**WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Return Values

x	Specifies the returned character.
ERR	Indicates an input error.

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.

Format

#include *curses*

void initscr *(***void***);*

[w]insch

The **insch** macro and the **winsch** function insert the character ch at the current cursor position in the specified window. The **insch** macro acts on the stdscr window.

Format

#include *curses*

insch (**char** *ch);*

int winsch (**WINDOW** **win,* **char** *ch);*

Arguments

win
Is a pointer to the window.

ch
Is the character to be inserted.

Description

After inserting the character, each character on the line shifts to the right, and Curses deletes the last character in the line. For more information, see the **scrollok** macro in this section.

Return Values

1	Indicates success.
ERR	Indicates that the function makes the screen scroll illegally.

[w]insertln

The **insertln** macro and the **winsertln** function insert a line above the line containing the current cursor position. The **insertln** macro acts on the stdscr window.

Format

#include *curses*

insertln( );

int winsertln **(WINDOW** ***win);**

Arguments

win
Is a pointer to the window.

Description

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. For more information, see the **scrollok** macro in this section.

Return Values

1	Indicates success.
ERR	Indicates that the function makes the screen scroll illegally.

[w]insstr

The **insstr** macro and the **winsstr** function insert a string at the current cursor position on the specified window. The **insstr** macro acts on the stdscr window.

Format

#include *curses*

insstr **(char** **str);*

int winsstr **(WINDOW** **win,* **char** **str);*

Arguments

win
Is a pointer to the window.

str
Is a pointer to the string to be inserted.

Description

Each character after the string shifts to the right, and the last character disappears. For more information, see the **scrollok** macro in this section. The macro and function are VAX C specific and are not portable.

Return Values

1	Indicates success.
ERR	Indicates that the function makes the screen scroll illegally.

isalnum

The **isalnum** macro returns a nonzero integer if its argument is one of the alphanumeric ASCII characters. Otherwise, it returns 0.

Format

#include *ctype*

int isalnum (**int** *character);*

Arguments

character
Is an object of type **int**.

isalpha

The **isalpha** macro returns a nonzero integer if its argument is an alphabetic ASCII character. Otherwise, it returns 0.

Format

#include *ctype*

int isalpha (**int** *character);*

Arguments

character
Is an object of type **int**.

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 about mailboxes, see Chapter 5.

Format

#include *unixio*

int isapipe (**int** *file_desc);*

Arguments

file_desc
Is a file descriptor.

Return Values

–1	Indicates an error (for example, if the file descriptor is not associated with an open file).
1	Indicates an association with a mailbox.
0	Indicates no association with a mailbox.

isascii

The **isascii** macro returns a nonzero integer if its argument is any ASCII character. Otherwise, it returns 0.

Format

#include *ctype*

int isascii (**int** *character);*

Arguments

character
Is an object of type **char**.

isatty

The **isatty** function returns 1 if the specified file descriptor is associated with a terminal, and 0 if it is not.

Format

#include *unixio*

int isatty **(int** *file_desc);*

Arguments

file_desc
Is a file descriptor.

Return Values

–1	Indicates an error (for example, if the file descriptor is not associated with an open file).
1	Indicates that the file descriptor is associated with a terminal.
0	Indicates that the file descriptor is not associated with a terminal.

iscntrl

The **iscntrl** macro returns a nonzero integer if its argument is an ASCII DEL character (177 octal) or any nonprinting ASCII character (a code less than 40 octal). Otherwise, it returns 0.

Format

#include *ctype*

int iscntrl *(***int** *character);*

Arguments

character
Is an object of type **int**.

isdigit

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

Format

#include *ctype*

int isdigit *(***int** *character);*

Arguments

character
Is an object of type **int**.

isgraph

The **isgraph** macro returns a nonzero integer if its argument is a graphic ASCII character. Otherwise, it returns 0.

Format

#include *ctype*

int isgraph (**int** *character*);

Arguments

character
Is an object of type **int**.

Description

Graphic ASCII characters have octal codes greater than or equal to 41 (!) and less than or equal to 176 (?~). They make up the set of characters you can print minus the space.

islower

The **islower** macro returns a nonzero integer if its argument is a lowercase alphabetic ASCII character. Otherwise, it returns 0.

Format

#include *ctype*

int islower (**int** *character);*

Arguments

character
Is an object of type **int**.

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 0.

Format

#include *ctype*

int isprint (**int** *character*);

Arguments

character
Is an object of type **int**.

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 0.

Format

#include *ctype*

int ispunct *(***int** *character);*

Arguments

character
Is an object of type **int**.

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 0.

Format

#include *ctype*

int isspace *(***int** *character);*

Arguments

character
Is an object of type **int**.

isupper

The **isupper** macro returns a nonzero integer if its argument is an uppercase alphabetic ASCII character. Otherwise, it returns 0.

Format

#include *ctype*

int isupper (**int** *character);*

Arguments

character
Is an object of type **int**.

isxdigit

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

Format

#include *ctype*

int isxdigit (**int** *character);*

Arguments

character
Is an object of type **int**.

kill

The **kill** function sends a signal to the process specified by a process ID. This function does not support the same functionality supported by UNIX systems.

Format

#include *signal*

int kill *(***int** *pid,* **int** *sig);*

Arguments

pid
Is the process ID.

sig
Is the signal code.

Description

Unless you have system privileges, the sending and receiving processes must have the same User Identification Code (UIC).

If pid is the process ID of the invoking process, then the **kill** function acts as if the **raise** function had been called.

If **kill** is successful, the receiving process is terminated. The termination status of the receiving process is the VMS error code corresponding to the value of the signal that was sent.

Return Values

0	Indicates that **kill** was successfully queued.
–1	Indicates errors. The receiving process may have a different UIC and you are not a system user, or the receiving process does not exist.

labs

The **labs** function returns the absolute value of an integer as a **long int**.

Format

#include *stdlib*

long int labs (**long int** *j*);

Arguments

j
Is a value of type **long int**.

ldexp

The **ldexp** function returns its first argument multiplied by 2 raised to the power of its second argument; that is, $x(2^e)$.

Format

#include *math*

double ldexp *(***double** *x,* **int** *e);*

Arguments

x
Is a base value, of type **double**, that is to be multiplied by 2^e.

e
Is the integer exponent value to which 2 is raised.

Description

If the calculation causes an overflow, the **ldexp** function sets errno to ERANGE and returns the value HUGE_VAL. The constant HUGE_VAL is defined in the *math* definition module to be the largest possible value of the appropriate sign.

Return Values

0	Indicates that underflow has occurred.
x	Indicates that overflow has occurred, and returns the largest possible value of the appropriate sign.

ldiv

The **ldiv** function returns the quotient and the remainder after the division of its arguments.

Format

#include *stdlib*

ldiv_t ldiv *(***long int** *numer,* **long int** *denom);*

Arguments

numer
Is a numerator of type **long int**.

denom
Is a denominator of type **long int**.

Description

The type **ldiv_t** is 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;
```

In VAX C, **ldiv** and **div** perform the same function.

See also **div** in this section.

leaveok

The **leaveok** macro signals Curses to leave the cursor at the current coordinates after an update to the window.

Format

#include *curses*

leaveok *(***WINDOW** **win,* **bool** *boolf);*

Arguments

win
Is a pointer to the window.

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, the cursor moves to the currently specified (y,x) coordinates of win. Values for boolf are defined in the *curses* definition module.

Description

The **leaveok** macro defaults to moving the cursor to the current coordinates of win.

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.

Format

#include *time*

struct tm *localtime **(const time_t** **bintim);*

Arguments

bintim
Is a pointer to the time in seconds relative to 00:00:00, January 1, 1970. You can generate this time by using the **time** function or you can supply a time.

Description

The converted time value is placed in a time structure defined in the *time* definition module with the tag tm. Table REF–5 describes the member names that are offsets into the structure.

Table REF–5: Member Names

tm_sec	Time in seconds
tm_min	Time in minutes
tm_hour	Time in hours (24)
tm_mday	Day of the month (1 to 31)

(continued on next page)

Table REF–5 (Cont.): Member Names

tm_mon	Month (0 to 11)
tm_year	Year (last two digits)
tm_wday	Day of the week (0 to 6)
tm_yday	Day of the year (0 to 365)
tm_isdst	Daylight savings time (always 0)

The member names are integers.

Successive calls to **localtime** overwrite the structure.

Return Values

pointer	Indicates a pointer to the time structure.

log, log10

The **log** and **log10** functions return the logarithm of their arguments.

Format

#include *math*

double log *(***double** *x);*

double log10 *(***double** *x);*

Arguments

x
Is a real number.

Return Values

Natural (base e) logarithm of the argument, which must be of type **double**	The returned value is also **double** for **log**.
Base 10 logarithm of its **double** argument	The returned value is **double** for **log10**.
0	Indicates that the argument is 0 or negative, and sets errno to EDOM.

longjmp

The **longjmp** function provides a way to transfer control from a nested series of function invocations back to a predefined point without returning normally; that is, by not using a series of **return** statements. The **longjmp** function restores the context of the environment buffer.

Format

#include *setjmp*

void longjmp (**jmp_buf** *env,* **int** *value);*

Arguments

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
Is passed from **longjmp** to **setjmp**, and then becomes the subsequent return value of the **setjmp** call. If value is passed as 0, it is converted to 1.

Description

When **setjmp** is first called, it returns the value 0. 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 you supply 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 the VMS system to unwind back to the desired destination. The VAX C RTL must be in control of signal handling for any VAX C image. For VAX C to be in control of signal handling, you must establish all exception handlers through a call to the **VAXC$ESTABLISH** function. See the **VAXC$ESTABLISH** function in this section for more information.

CAUTION

You cannot invoke the **longjmp** function from a VMS condition handler. However, you may invoke **longjmp** from a signal handler that has been established for any signal supported by the VAX C RTL, subject to the following nesting restrictions:

- 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.
- Do not invoke the **setjmp** function from a signal handler unless the associated **longjmp** is to be issued before the handling of that signal is completed.

longname

The **longname** function assigns the full terminal name to name, which must be large enough to hold the character string.

Format

#include *curses*

void longname *(***char** **termbuf,* **char** **name);*

Arguments

termbuf
Is a string containing the name of the terminal.

name
Is a character-string buffer with a minimum length of 64 characters.

Description

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 serves no function in the VMS environment. If portability is a concern, you must write a set of dummy routines to perform the functionality provided by the data base *termcap* in the UNIX system environment.

lseek

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

Format

#include *unixio*

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

Arguments

file_desc
Is an integer returned by **open**, **creat**, **dup**, or **dup2**.

offset
Is measured in bytes.

direction
Tells the **lseek** function where to begin the offset. The new position is relative either to the beginning of the file (direction=SEEK_SET), the current position (direction=SEEK_CUR), or the end of the file (direction=SEEK_END).

Description

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 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 0 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, SEEK_CUR)
```

You can then use the return value pos later in the program (perhaps after repositioning the file with **write** or **read**) to return to this position, as in the following example:

```
/* POSITION RELATIVE TO BEGINNING         */

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

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.

See also **open**, **creat**, **dup**, **dup2**, and **fseek** in this section.

Return Values

−1	Indicates that the file descriptor is undefined or a seek was attempted before the beginning of the file.

malloc

The **malloc** function allocates an area of memory.

Format

#include *stdlib*

void *malloc *(***size_t** *size);*

Arguments

size
Is the total number of bytes to be allocated.

Description

The **malloc** function allocates a contiguous area of memory whose size, in bytes, is supplied as an argument. The space is not initialized.

Return Values

0	Indicates that it is unable to allocate enough memory.
x	The address of the first byte, which is aligned on an octaword boundary.

memchr

The **memchr** function locates the first occurrence of the specified byte within the initial size bytes of a given object.

Format

#include *string*

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

Arguments

s1
Is a pointer to the object to be searched.

c
Is the byte value to be located.

size
Is the length of the object to be searched.

Description

Unlike **strchr**, the **memchr** function does not stop when it encounters a null character.

Return Values

pointer	Is 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 null pointer.

memcmp

The **memcmp** function compares two objects, byte by byte. The compare operation starts with the first byte in each object.

Format

#include *string*

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

Arguments

s1
Is a pointer to the first object.

s2
Is a pointer to the second object.

size
Is the length of the objects to be compared.

Description

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 null character is encountered.

See also **strcmp** in this section.

Return Values

x	Is 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.

memcpy

The **memcpy** function copies a specified number of bytes from one object to another.

Format

#include *string*

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

Arguments

s1
Is a pointer to the first object.

s2
Is a pointer to the second object.

size
Is the length of the object to be copied.

Description

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). Unlike the **strcpy** function, the **memcpy** function does not stop when a null character is encountered.

Return Values

x	Indicates the value of s1, which is a pointer.

memmove

The **memmove** function copies a specified number of bytes from one object to another.

Format

#include *string*

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

Arguments

s1
Is a pointer to the first object.

s2
Is a pointer to the second object.

size
Is the length of the object to be copied.

Description

In VAX C, **memmove** and **memcpy** perform the same function. Programs that require portability should use **memmove** if the area pointed at by s1 could overlap the area pointed at by s2.

Example

```
main(){
char *pdest =   "hello   there";
char *psource = "you are there";

 memmove( pdest, psource, 7);
 printf("%s\n", pdest);

}
```

memset

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

Format

#include *string*

void *memset *(***void** **s*, **int** *value*, **size_t** *size);*

Arguments

s
Is an array pointer.

value
Is the value to be placed in s.

size
Is the number of bytes to be placed in s.

Description

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

mkdir

The **mkdir** function creates a directory.

Format

#include *stdlib*

int mkdir *(***char** **dir_spec,* **unsigned** *mode, . . .);*

Arguments

dir_spec
Is a valid VMS or UNIX-style directory specification that may contain a device name. For example:

```
DBA0:[BAY.WINDOWS]      /*    VMS          */
/dba0/bay/windows       /*   UNIX-style    */
```

This specification cannot contain a node name, file name, file extension, file version, or a wildcard character. The same restriction applies to the UNIX-style directory specifications. For more information about the restrictions on UNIX-style specifications, see Chapter 1.

mode
Is a file protection. See the **chmod** function in this section for information about the specific file protections. All parent-directory defaults are applied to the new directory unless you override them.

. . .
Represents the following optional arguments:

uic
Is the user identification code (UIC) that identifies the owner of the created directory. If this argument is 0, 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
Is 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 0, 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 0, VAX C arbitrarily places the created directory within the volume set. This optional argument is VAX C specific and is not portable.

Description

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 where these intermediate directories must exist and will not be created.

VAX C implements this function using the VMS RTL routine LIB$CREATE_DIR. For more information, see the *VMS Run-Time Library Routines Volume*.

Return Values

0	Indicates success.
–1	Indicates failure.

mktemp

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

Format

#include *unixio*

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

Arguments

template
Is a pointer to a user-defined template. You supply the template in the form, namXXXXXX. The six trailing Xs are replaced by a unique series of characters. You may supply the first three characters.

Description

The use of **mktemp** is not recommended for new applications. See the **tmpnam** function for the preferable alternative.

Return Values

x	A pointer to the template, with the template modified to contain the created file name. If this value is a pointer to a null string, it indicates that a unique file name cannot be created.

modf

The **modf** function returns the positive fractional part of its first argument and assigns the integer part, expressed as an object of type **double**, to the object whose address is specified by the second argument.

Format

#include *math*

double modf **(double** *value,* **double** **iptr);*

Arguments

value
Must be an object of type **double**.

iptr
Is a pointer to an object of type **double**.

[w]move

The **move** macro and the **wmove** function change the current cursor position on the specified window to the coordinates (y,x). The **move** macro acts on the stdscr window.

Format

#include *curses*

move *(y,x);*

int wmove *(***WINDOW** **win,* **int** *y,* **int** *x);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

Description

For more information, see the **scrollok** macro in this section.

Return Values

1	Indicates success.
ERR	Indicates that the function makes the screen scroll illegally.

mvcur

The **mvcur** function moves the terminal's cursor from (lasty,lastx) to (newy,newx).

Format

#include *curses*

int mvcur *(***int** *lasty,* **int** *lastx,* **int** *newy,* **int** *newx);*

Arguments

lasty
Is the cursor position.

lastx
Is the cursor position.

newy
Is the resulting cursor position.

newx
Is the resulting cursor position.

Description

In VAX C, **mvcur** and **move** perform the same function.

See also **move** in this section.

Return Values

ERR	Indicates that moving the window put part or all of the window off the edge of the terminal screen. The terminal screen remains unaltered.

mvwin

The **mvwin** function moves the starting position of the window to the specified (y,x) coordinates.

Format

#include *curses*

mvwin (**WINDOW** **win,* **int** *y,* **int** *x);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

Description

When moving subwindows, the **mvwin** function does not rewrite the contents of the subwindow on the underlying window at the new position. If you write anything to the subwindow after the move, the function also writes to the underlying window.

Return Values

ERR	Indicates that moving the window put part or all of the window off the edge of the terminal screen. The terminal screen remains unaltered.

mv[w]addch

The **mvaddch** and **mvwaddch** macros move the cursor to coordinates (y,x) and add the character ch to the specified window. The **mvaddch** macro acts on the stdscr window.

Format

#include *curses*

mvaddch *(***int** *y,* **int** *x,* **char** *ch);*

mvwaddch *(***WINDOW** **win,* **int** *y,* **int** *x,* **char** *ch);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

ch
If this argument is a newline (\n), the **mvaddch** and **mvwaddch** macros 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 at the normal tabstop positions (every eight characters).

mv[w]addstr

The **mvaddstr** and **mvwaddstr** macros move the cursor to coordinates (y,x) and add the specified string, to which str points, to the specified window. The **mvaddstr** macro acts on the stdscr window.

Format

#include *curses*

mvaddstr (**int** *y*, **int** *x*, **char** **str);*

mvwaddstr (**WINDOW** **win*, **int** *y*, **int** *x*, **char** **str);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

str
Is the string that is displayed.

mv[w]delch

The **mvdelch** and **mvwdelch** macros move the cursor to coordinates (y,x) and delete the character on the specified window. The **mvdelch** macro acts on the stdscr window.

Format

#include *curses*

mvdelch (**int** *y*, **int** *x*);

mvwdelch (**WINDOW** **win*, **int** *y*, **int** *x*);

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

Description

Each of the following characters on the same line shifts to the left, and the last character becomes blank.

mv[w]getch

The **mvgetch** and **mvwgetch** macros move the cursor to coordinates (y,x), get a character from the terminal screen, and echo it on the specified window. The **mvgetch** macro acts on the stdscr window.

Format

#include *curses*

mvgetch *(***int** *y,* **int** *x);*

mvwgetch *(***WINDOW** **win,* **int** *y,* **int** *x);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

mv[w]getstr

The **mvgetstr** and **mvwgetstr** macros move the cursor to coordinates (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 **mvgetstr** macro acts on the stdscr window.

Format

#include *curses*

mvgetstr *(***int** *y,* **int** *x,* **char** **str);*

mvwgetstr *(***WINDOW** **win,* **int** *y,* **int** *x,* **char** **str);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

str
Is the string that is displayed.

Description

The **mvgetstr** and **mvwgetstr** macros strip the newline terminator (\n) from the string.

mv[w]inch

The **mvinch** and **mvwinch** macros move the cursor to coordinates (y,x) and return the character on the specified window without making changes to the window. The **mvinch** macro acts on the stdscr window.

Format

#include *curses*

mvinch *(***int** *y,* **int** *x);*

mvwinch *(***WINDOW** **win,* **int** *y,* **int** *x);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

mv[w]insch

The **mvinsch** and **mvwinsch** macros move the cursor to coordinates (y,x) and insert the character ch in the specified window. The **mvinsch** macro acts on the stdscr window.

Format

#include *curses*

mvinsch *(***char** *ch,* **int** *y,* **int** *x);*

mvwinsch *(***WINDOW** **win,* **int** *y,* **int** *x,* **char** *ch);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

ch
Is the character to be inserted at the window's coordinates.

Description

After inserting the character, each character on the line shifts to the right, and the last character disappears.

mv[w]insstr

The **mvinsstr** and **mvwinsstr** macros move the cursor to coordinates (y,x) and insert a string in the specified window. The **mvinsstr** macro acts on the stdscr window.

Format

#include *curses*

mvinsstr (**int** *y*, **int** *x*, **char** **str);*

mvwinsstr (**WINDOW** **win*, **int** *y*, **int** *x*, **char** **str);*

Arguments

win
Is a pointer to the window.

y
Is a window coordinate.

x
Is a window coordinate.

str
Is the string that is displayed.

Description

Each character after the string shifts to the right, and the last character disappears. The **mvinsstr** and **mvwinsstr** macros are VAX C specific and are not portable.

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.

Format

#include *curses*

WINDOW *newwin (**int** *numlines,* **int** *numcols,* **int** *begin_y,* **int** *begin_x);*

Arguments

numlines
If it is 0, the **newwin** function sets that dimension to LINES (begin_y). To get a new window of dimensions LINES by COLS, use the following line:

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

numcols
If it is 0, the **newwin** function sets that dimension to COLS (begin_x). Thus, to get a new window of dimensions LINES by COLS, use the following line:

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

begin_y
Is a window coordinate.

begin_x
Is a window coordinate.

Return Values

x	Indicates the address of the allocated window.
0	Indicates an error.

nice

The **nice** function increases or decreases process priority relative to the process base priority by the amount of the argument.

Format

#include *stdlib*

int nice (**int** *increment);*

Arguments

increment
As a positive argument decreases priority, and as a negative argument increases priority. The resulting priority cannot be less than 1 or greater than the process's base priority.

Description

When a process calls the **vfork** function, the resulting child inherits the parent's priority.

See also **vfork** in this section.

Return Values

0	Indicates success.
–1	Indicates failure.

[no]nl

The **nl** and **nonl** macros are provided only for UNIX software compatibility and have no function in the VMS environment.

Format

#include *curses*

nl()

nonl()

open

The **open** function opens a file for reading, writing, or editing. It positions the file at its beginning (byte 0).

Format

#include *unixio*

#include *file*

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

Arguments

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

flags
Are values defined in the *file* definition module as follows:

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_EXCL	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 1, 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_EXCL is set with O_CREAT and the file exists, the attempted open returns an error.

mode
Sets the file protection. You can construct modes by using the bitwise OR operator (|) to separate specified modes. The modes are described 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 0, **open** gives the file your default file protection.

The system is given the same access 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, . . . "
```

The keyword is a Record Management Services (RMS) 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 REF–6 lists the set of valid keywords and values.

Table REF–6: RMS Valid Keywords and Values

Keyword	Value	Description
"alq = n"	decimal	Allocation quantity
"bls = n"	decimal	Block size
"ctx = bin"	string	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 = stm"	string	Force stream-mode access
"deq = n"	decimal	Default extension quantity
"dna = filespec"	string	Default file-name string
"fop = val, val, . . . "		File-processing options:
	ctg	Contiguous
	cbt	Contiguous-best-try
	dlt	Delete file on close
	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
	rck	Read check compare operation
	wck	Write check compare operation
	mxv	Maximize version number
	rwo	Rewind file on open
	pos	Current position
	rwc	Rewind file on close
"fsz = n"	decimal	Fixed header size
"mbc = n"	decimal	Multiblock count
"mbf = n"	decimal	Multibuffer count

(continued on next page)

Table REF–6 (Cont.): RMS Valid Keywords and Values

Keyword	Value	Description
"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:
	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
		Variable-length record format
	var	Variable-length record with fixed control
	vfc	Undefined
	udf	
"rop = val"		Record-processing operations:
	asy	Asynchronous I/O
	tmo	Timeout I/O
	RAH	Read ahead
	WBH	Write behind
"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

Description

If you specify a directory in the file name and it is a search list that contains an error, VAX C interprets it as a file open error.

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 section.

Return Values

-1	Indicates that the file does not exist, it is protected against reading or writing, or the file, for another reason, cannot be opened.
x	Indicates a nonnegative file descriptor number.

Example

```
#include unixio
#include file
main()
{
int file,stat;
int flags;

 flags = O_RDWR ;     /*  open for read and write,
                       *  with user default file protection,
                       *  with a maximum fixed record size of 2048 bytes,
                              *  and a block size 2048 bytes
                             */
 file = open("file.dat",flags,0,"rfm=fix","mrs=2048","bls=2048");
 if (file == -1)
  perror ("OPEN error"), exit(1);

 close (file);
}
```

overlay

The **overlay** function nondestructively superimposes win1 on win2. The function writes the contents of win1 that will fit onto win2 beginning at the starting coordinates of both windows. Blanks on win1 leave the contents of the corresponding space on win2 unaltered. The **overlay** function copies as much of a window's box as possible.

Format

#include *curses*

int overlay (**WINDOW** **win1*, **WINDOW** **win2);*

Arguments

win1
Is a pointer to the window.

win2
Is a pointer to the window.

Return Values

1	Indicates success.
0	Indicates an error.

overwrite

The **overwrite** function destructively writes the contents of win1 on win2.

Format

#include *curses*

int overwrite (**WINDOW** **win1*, **WINDOW** **win2*);

Arguments

win1
Is a pointer to the window.

win2
Is a pointer to the window.

Description

The **overwrite** function writes the contents of win1 that will fit onto win2 beginning at the starting coordinates of both windows. Blanks on win1 are written on win2 as blanks. This function copies as much of a window's box as possible.

Return Values

1	Indicates success.
0	Indicates failure.

pause

The **pause** function causes its calling process to stop (hibernate) until the process receives a signal.

Format

#include *signal*

int pause *(***void***);*

Description

Control is not returned to the process that called **pause**. You may reawaken the process by using **kill** or **alarm**.

The **pause** function uses the $HIBER system service. Because of this, a call to the SYS$WAKE system service will also wake up a paused process.

See also **kill** and **alarm** in this section.

Return Values

x	Specifies the value of the VMS $HIBER system service routine.

perror

The **perror** function writes a short error message to stderr describing the last error encountered during a call to the VAX C RTL from a C program.

Format

#include *stdio*

void perror *(***const char** **str);*

Arguments

str
Typically contains the name of the program that brought on the error.

Description

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. See the description of *errno* in Chapter 4.

Example

```
main(argc,argv)
int argc;
char *argv[];
{
 FILE *fp;
 int status;
 int total_recs = -1;
```

```
 fp = fopen(argv[1],"r");                     /* Open an input file */
 if (fp < 0)
 {
  /*
   * perror will print out a diagnostic explaining why
   * the open failed.
   *
   */
  perror("open");
  exit();
 }

 /*   . . . etc.  */
}
```

pipe

The **pipe** function creates a temporary mailbox. You must use a mailbox to read and write data between the parent and child. The channels through which the processes communicate are called a *pipe*.

Format

#include *processes*

int pipe (**int** *array_fdscptr[2], . . .);*

Arguments

array_fdscptr
Is 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.

. . .
Represents two additional arguments, as follows:

flags
Is an optional argument that is identical to the same argument in the **open** function. The values for the argument are defined in the *file* definition module as follows:

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_EXCL	Error if attempting to create an 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 1, 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_EXCL is set with O_CREAT and the file exists, the attempted open returns an error.

Do not use O_CREAT, O_EXCL, and O_TRUNC with pipes. O_APPEND is ignored with pipes.

bufsize
Is 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.

Description

The mailbox 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 the end-of-file.

The mailbox is created by using the $CREMBX system service, specifying the following characteristics:

- A maximum message length of 512 characters
- A 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 you cannot write more than 512 characters 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 the VMS system is a record-oriented file with no carriage-control attributes. It is fully buffered by default in the VAX C RTL.

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. You can then use the **getname** function to return the name of the mailbox associated with the pipe, if this information is desired. The mailbox name returned by **getname** 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 about file descriptors, see Chapter 2.

File descriptors 0, 1, and 2 are open in a VAX C program for stdin (SYS$INPUT), stdout (SYS$OUTPUT), and stderr (SYS$ERROR), respectively. Therefore, if no other files are open 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. In this case, the pipe does not necessarily use adjacent file descriptors. For example, assume that 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 is assigned for writing and file descriptor 5 is 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 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 to 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.

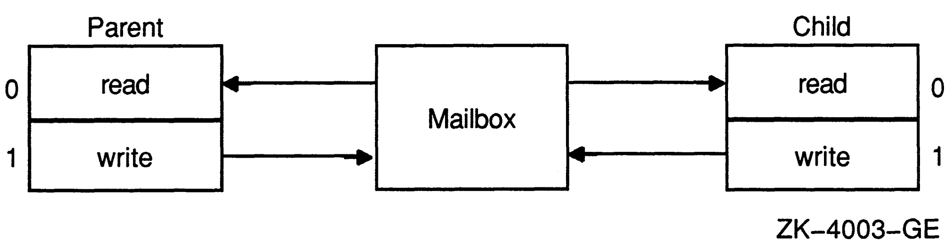
You can read and write through the pipe using the UNIX I/O functions **read** and **write**, specifying the appropriate file descriptors. As an alternative, you can issue **fdopen** calls to associate file pointers with these file descriptors so that you can use the Standard I/O functions (**fread** 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 0, then an end-of-file message is written to the mailbox.

Two separate file descriptors are used for reading from and writing to the pipe, but only one mailbox channel is used so 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 as shown in Figure REF–1.

Figure REF–1: Reading and Writing to a Pipe



Return Values

0	Indicates success.
–1	Indicates an error.

pow

The **pow** function returns the first argument raised to the power of the second argument.

Format

#include *math*

double pow (**double** *base*, **double** *exp);*

Arguments

base
Is a value of type **double** that is to be raised to a power.

exp
Is the exponent to which the power base is to be raised.

Description

Both arguments must be **double** and the returned value is **double**. If there is an overflow, the value HUGE_VAL is returned.

The constant HUGE is defined in the *math* definition module to be the largest possible value.

Return Values

The largest possible floating-point value	Indicates that the result overflowed. Errno is set to ERANGE.
0	Indicates success and sets errno to EDOM under the following conditions: • If both arguments are 0 • If exp is negative and not an integer • If base is negative and exp is not an integer

Example

```
#include stdio
#include math
main()
{
double x;

 errno=0;

 x = pow(-3.0, 2.0);
 printf("%d, %f\n", errno, x);

}
```

printf

The **printf** function performs formatted output from the standard output (stdout). See Chapter 2 for information on format specifiers.

Format

#include *stdio*

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

Arguments

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, you may omit the output sources. 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 left-to-right order.

Return Values

x	Indicates the number of characters written.
-1	Indicates that an output error has occurred.

[w]printw

The **printw** macro and **wprintw** function perform a **printf** (see **printf**) on the window starting at the current position of the cursor. The **printw** macro acts on the stdscr window. See Chapter 2 for information on format specifiers.

Format

#include *curses*

printw *(***char** **format_spec, . . .);*

int wprintw *(***WINDOW** **win,* **char** **format_spec, . . .);*

Arguments

win
Is a pointer to the window.

format_spec
Is a pointer to the format specification string.

. . .
Represents optional expressions whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, you may omit the output sources. 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 left-to-right order.

Description

The formatting specification (format_spec) and the other arguments are identical to those used with the **printf** function.

The **printw** macro and the **wprintw** function format and then print the resultant string to the window using the **addstr** macro. For more information, see the **printf** function and the **scrollok** macro in this section.

Return Values

1	Indicates success.
ERR	Indicates that the function makes the screen scroll illegally.

putc

The **putc** macro writes characters to a specified file.

Format

#include *stdio*

int putc *(***int** *character,* **FILE** **file_ptr);*

Arguments

character
Is an object of type **int**.

file_ptr
Is a file pointer.

Description

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

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

See also **fputc** and **putw** in this section.

Return Values

EOF	Indicates output errors.
character	Indicates success.

putchar

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

Format

#include *stdio*

int putchar *(***int** *character);*

Arguments

character
Is an object of type **int**.

Description

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

Return Values

EOF	Indicates output errors.
character	Indicates success.

puts

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

Format

#include *stdio*

int puts *(***char** **str);*

Arguments

str
Is a pointer to a character string to be written to stdout.

Description

The **puts** function does not copy the terminating null character to the output stream.

Return Values

Nonnegative value	Indicates success.
EOF	Indicates output errors.

putw

The **putw** function writes characters to a specified file.

Format

#include *stdio*

int putw *(***int** *integer,* **FILE** **file_ptr);*

Arguments

integer
Is an object of type **int** or **long**.

file_ptr
Is a file pointer.

Description

The **putw** function writes four characters to the output file as an **int**. No conversion is performed.

Return Values

EOF	Indicates output errors.
integer	Indicates success.

qsort

The **qsort** function sorts an array of objects in place. It implements the quick-sort algorithm.

Format

#include *stdlib*

void qsort (**void** **base,* **size_t** *nmemb,* **size_t** *size,* **int** *(*compar)* (**const void** **,* **const void** **));*

Arguments

base
Is a pointer to the first member of the array. The pointer should be of type pointer-to-element and cast to type pointer-to-character.

nmemb
Is the number of objects in the array.

size
Is the size of an object, in bytes.

compar
Is a pointer to the comparison function.

Description

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 then, equal to, or greater than 0.

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.

raise

The **raise** function generates a specified software signal. Generating a signal causes the action established by the **ssignal** function to be taken.

Format

#include *signal*

int raise (**int** *sig, . . .);*

Arguments

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 REF–7 presents the various codes.

Table REF–7: SIGFPE Signal Codes

Hardware Condition	Signal	Code
Arithmetic Traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by 0	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by 0	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP

(continued on next page)

Table REF–7 (Cont.): SIGFPE Signal Codes

Hardware Condition	Signal	Code
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by 0 fault	SIGFPE	FPE_FLTDIV_FAULT
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 to 10; the SIGILL codes are represented by the numbers 0 to 2. The code values are defined in the *signal* definition module.

Description

Calling the **raise** function has one of the following results:

- If **raise** specifies a sig argument that is outside the range defined in the signal module, then the specified function returns 0, and the variable errno is set to EINVAL. See Chapter 4 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 corresponding to the signal.
- If **ssignal** establishes SIG_IGN (ignore signal) as the action for the signal, then **raise** returns its argument, sig.
- **ssignal** must establish an action function for the signal. That function is called and its return value is returned by **raise**.

See also **ssignal** in this section.

rand

The **rand** function returns pseudorandom numbers in the range 0 to $2^{31} - 1$.

Format

#include *math*

int rand *(***void***);*

Description

The **rand** function uses a multiplicative congruential random number generator with a repeat factor (period) of 2^{31}.

See also **srand** in this section.

[no]raw

Like cbreak mode, raw mode only works with the Curses input routines **[w]getch** and **[w]getstr**. Raw mode is not supported with the VAX C RTL's emulation of UNIX I/O, Terminal I/O, or Standard I/O.

Format

include *curses*

raw( )

noraw( )

Description

Raw mode reads are satisfied on one of two conditions: after a minimum number (5) of characters are input at the terminal or after waiting a fixed time (10 seconds) from receipt of any characters from the terminal.

Example

```
/* example of standard and raw input  in curses package */

# include curses

main ()
{
 WINDOW *win1;
 char vert = '.', hor = '.', str[80];

 /* Initialize standard screen, turn echo off  */

 initscr ();
      noecho ();

 /* Define a user window  */

 win1 = newwin (22, 78, 1, 1);
        leaveok( win1, TRUE);
      leaveok (stdscr,TRUE);

      box (stdscr, vert, hor);
```

```
 /* Reset the video, refresh(redraw) both windows */

 mvwaddstr (win1, 2, 2, "test line terminated input");
      wrefresh (win1);

 /* Do some input and output it */
      nocrmode();
 wgetstr (win1, str);

     mvwaddstr (win1, 5, 5, str);
 mvwaddstr (win1, 7, 7, "Type something to clear screen");
     wrefresh (win1);

 /* Get another character then delete the window */

 wgetch (win1);
 wclear (win1);

 mvwaddstr (win1, 2, 2, "test raw input");
      wrefresh (win1);

 /* Do some raw input 5 chars or timeout - and output it */
      raw();
 getstr (str);
      noraw();
      mvwaddstr (win1, 5, 5, str);
 mvwaddstr (win1, 7, 7, "Raw input completed");
      wrefresh (win1);

 endwin ();
}
```

read

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

Format

#include *unixio*

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

Arguments

file_desc
Is a file descriptor. The specified file descriptor must refer to a file currently opened for reading.

buffer
Is the address of contiguous storage in which the input data is placed.

nbytes
Is the maximum number of bytes involved in the read operation.

Description

The **read** function returns the number of bytes 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

The **read** function does not span record boundaries in a record file and, therefore, reads only one record. A separate read must be done for each record.

Return Values

0	Indicates that the end-of-file was encountered.
–1	Indicates a read error, including physical input errors, illegal buffer addresses, protection violations, undefined file descriptors, and so forth.

Example

```
#include file
#include unixio

main()
{
int fd,i;
char buf[10];

       if ( (fd=open("test.txt",O_RDWR,0,"shr=upd")) <= 0 )
       {
        perror("open");
  exit();
 }

 /* read 2 characters  into buf  */

 if ( (i=read(fd, buf, 2)) < 0)
 {
  perror("read");
  exit();
 }
       else
  if ( i == -1) /* test for end of file  */
   exit();

 /* print out what was read  */

 if( i > 0)
  printf("buf='%c%c'\n",buf[0],buf[1]);

 close(fd);
}
```

realloc

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.

Format

#include *stdlib*

void *realloc (**void** **ptr,* **size_t** *size);*

Arguments

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.

Description

If ptr is the null pointer constant (NULL), the behavior of the **realloc** function is identical to the **malloc** function.

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 0.

See also **free**, **cfree**, **calloc**, and **malloc** in this section.

Return Values

x	Indicates the address of the area, since the area may have to be moved to a new address to reallocate enough space. If the area was moved, the space previously occupied is freed.
0	Indicates that space cannot be reallocated (for example, if there is not enough room).

[w]refresh

The **refresh** macro and the **wrefresh** function repaint the specified window on the terminal screen. The **refresh** macro acts on the stdscr window.

Format

#include *curses*

refresh( )

int wrefresh (**WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Description

The result of this process is that the portion of the window 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 the **refresh** macro or **wrefresh** function.

See also **touchwin** in this section.

Return Values

1	Indicates success.
ERR	Indicates an error.

remove

The **remove** function causes a file to be deleted.

Format

#include *stdio*

int remove *(***const char** **file_spec);*

Arguments

file_spec
Is a pointer to the string that is a VMS or a UNIX-style file specification.

Description

If you specify a directory in the file name and it is a search list that contains an error, VAX C interprets it as a file error.

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

See also **delete** in this section.

Return Values

0	Indicates success.
nonzero value	Indicates failure.

rename

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

Format

#include *stdio*

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

Arguments

old_file_spec
Is a pointer to a string that is the existing name of the file to be renamed.

new_file_spec
Is a pointer to a string that is the new name to be given to the file.

Description

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

Return Values

0	Indicates success.
nonzero value	Indicates failure.

rewind

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

Format

#include *stdio*

int rewind (**FILE** **file_ptr);*

Arguments

file_ptr
Is a file pointer.

Description

The **rewind** function is equivalent to **fseek** (file pointer, 0,0). You can use the **rewind** function with either record or stream files.

See also **fseek** in this section.

Return Values

0	Indicates success.
EOF	Indicates failure.

sbrk

The **sbrk** function determines the lowest virtual address that is not used with the program.

Format

#include *stdlib*

void *sbrk *(***unsigned long int** *incr);*

Arguments

incr
Specifies, to the **sbrk** function, the number of bytes to add to the current break address.

Description

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, **sbrk** is needed only by programs that have growing data areas.

Return Values

–1	Indicates that the program requests too much memory.
x	Indicates the old break address.

scanf

The **scanf** function performs formatted input from the standard input (stdin). See Chapter 2 for information on format specifiers.

Format

#include *stdio*

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

Arguments

format_spec
Contains characters to be taken literally from the input or converted and placed in memory at the specified input_sources. For a list of conversion characters, see Chapter 2.

. . .
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, you may omit these input pointers. 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 left-to-right order.

Return Values

x	Indicates the number of successfully matched and assigned input items.
EOF	Indicates that the end-of-file is encountered. EOF is a preprocessor constant defined in the *stdio* definition module.

[w]scanw

The **scanw** and **wscanw** functions perform a **scanf** on the window. The **scanw** function acts on the stdscr window.

Format

#include *curses*

int scanw (**char** **format_spec, . . .);*

int wscanw (**WINDOW** **win,* **char** **format_spec, . . .);*

Arguments

win
Is a pointer to the window.

format_spec
Is a pointer to the format specification string.

. . .
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, you may omit these input pointers. 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 left-to-right order.

[w]scanw

Description

The formatting specification (format_spec) and the other arguments are identical to those used with the **scanf** function.

The **scanw** and **wscanw** functions accept, format, and return a line of text from the terminal screen. For more information, see the **scrollok** macro and **scanf** function in this section.

Return Values

1	Indicates success.
ERR	Indicates that the function makes the screen scroll illegally or that the scan was unsuccessful.

scroll

The **scroll** function moves all the lines on the window up one line. The top line scrolls off the window and the bottom line becomes blank.

Format

#include *curses*

int scroll **(WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Return Values

1	Indicates success.
0	Indicates an error.

scrollok

The **scrollok** macro sets the scroll flag for the specified window.

Format

#include *curses*

#define bool int

scrollok (**WINDOW** **win,* **bool** *boolf);*

Arguments

win
Is a pointer to the window.

boolf
Is a Boolean TRUE or FALSE value. If boolf is FALSE, scrolling is not allowed. This is the default setting. The boolf argument is defined in the *curses* definition module.

[w]setattr

The **setattr** macro and the **wsetattr** function activate the video display attribute attr within the window. The **setattr** macro acts on the stdscr window.

Format

#include *curses*

setattr *(attr);*

int wsetattr *(***WINDOW** **win,* **int** *attr);*

Arguments

win
Is a pointer to the window.

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, respectively. You can set multiple attributes by separating them with a bitwise OR operator (|) as follows:

```
setattr(_BLINK | _UNDERLINE);
```

Description

The **setattr** macro and **wsetattr** function are VAX C specific and are not portable.

Return Values

1	Indicates success.
ERR	Indicates an error.

setbuf

The **setbuf** function associates a buffer with an input or output file.

Format

#include *stdio*

void setbuf (**FILE** **file_ptr,* **char** **buffer);*

Arguments

file_ptr
Is a pointer to a file.

buffer
Is a pointer to an array. I/O operations are 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, I/O operations will be completely unbuffered, and the pointer in buffer is ignored. Otherwise, I/O operations are performed using the array pointed to by buffer.

Description

You can use the **setbuf** function after a file is opened but you must use it before any I/O operations.

A common 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, see the **malloc** function and **setvbuf** function in this section.

setgid

The **setgid** function is implemented for program portability and serves no function. It returns 0 (to indicate success).

Format

#include *unixlib*

int setgid **(unsigned int** *group_number);*

Arguments

group_number
Is the group number.

setjmp

The **setjmp** function provides a way to transfer control from a nested series of function invocations back to a predefined point without returning normally. It does not use a series of **return** statements. The **setjmp** function saves the context of the calling function in an environment buffer.

Format

#include *setjmp*

int setjmp **(jmp_buf** *env);*

Arguments

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.

Description

When **setjmp** is first called, it returns the value 0. 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 you 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 that allows the VMS condition-handling facility to unwind back to the desired destination.

The VAX C RTL must be in control of signal handling for any VAX C image. For VAX C to be in control of signal handling, you must establish all exception handlers through a call to the **VAXC$ESTABLISH** function. See the **VAXC$ESTABLISH** function in this section for more information.

CAUTION

You cannot invoke the **longjmp** function from a VMS condition handler. However, you may invoke **longjmp** from a signal handler that has been established for any signal supported by the VAX C RTL, subject to the following nesting restrictions:

- The **longjmp** function will not work if you invoke it 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.
- Do not invoke the **setjmp** function from a signal handler unless the associated **longjmp** is to be issued before the handling of that signal is completed.

Return Values

See the Description section.

setuid

The **setuid** function is implemented for program portability and serves no function. It returns 0 (to indicate success).

Format

#include *unixlib*

int setuid **(unsigned int** *member_number);*

Arguments

member_number
Is the member number.

setvbuf

The **setvbuf** function associates a buffer with an input or output file.

Format

#include *stdio*

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

Arguments

file_ptr
Is a pointer to a file.

buffer
Is a pointer to an array. If either _IOFBF or _IOLBF is specified as a value for type, I/O operations are 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, I/O operations are done using the buffer automatically allocated by the VAX C RTL. If _IONBF is specified by type, I/O operations are completely unbuffered and the pointer in buffer is ignored.

type
Is a value that determines how the file will be buffered.

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

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

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

Description

You can use the **setvbuf** function after a file is opened but you must use it before any I/O operations.

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, see the **malloc** function and **setbuf** function in this section.

Return Values

0	Indicates success.
nonzero value	Indicates that an invalid value is given for type or size.

sigblock

The **sigblock** function causes the signals in mask to be added to the current set of signals being blocked from delivery.

Format

#include *signal*

int sigblock *(***int** *mask);*

Arguments

mask
Contains the signals that will be blocked.

Description

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 line:

```
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 by using the bitwise OR operator (|).

Return Values

x	Indicates the previous set of masked signals.

signal

The **signal** function allows you either to catch or to ignore a signal.

Format

#include *signal*

void **(*signal (int** *sig,* **void** *(*func)* **(int,** *. . .)))* **(int,** *. . .);*

Arguments

sig
Is the number or mnemonic associated with a signal. The sig argument is usually one of the mnemonics defined in the *signal* definition module.

func
Is either the action to take 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 that 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. Signals are reset to SIG_DFL after they are caught, except as shown in Chapter 4.)

Description

You must call the **signal** function each time you want to catch a signal.

To cause a VMS exception or signal to generate a UNIX-style signal, user condition handlers must return SS$_RESIGNAL upon receiving any exception that they do not want to handle. Returning SS$_NORMAL prevents the generation of a UNIX-style signal. UNIX signals are generated as if by an exception handler in the stack frame of the main C program. Not all VMS exceptions correspond to UNIX signals.

Return Values

x	Indicates the address of the function previously (or initially) established to handle the signal.
–1	Indicates that the sig argument is out of range. The variable errno is set to EINVAL.

sigpause

The **sigpause** function assigns mask to the current set of masked signals and then waits for a signal.

Format

#include *signal*

int sigpause (**int** *mask);*

Arguments

mask
Contains the signals that will be blocked.

Description

See the **sigblock** function in this section for information about the argument mask.

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.

A signal is usually 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.

Return Values

EINTR	Indicates an interrupt.

sigsetmask

The **sigsetmask** function establishes those signals that are blocked from delivery.

Format

#include *signal*

int sigsetmask *(***int** *mask);*

Arguments

mask
Contains the signals that will be blocked.

Description

See the **sigblock** function in this section for information about 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 by using the bitwise OR operator (|).

Return Values

x	Indicates the previous set of masked signals.

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.

Format

#include *signal*

int sigstack *(***struct sigstack** **ss,* **struct sigstack** **oss);*

Arguments

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 that will be stored to the current state of the signal stack.

Description

The sigstack structure is defined in the standard include module *signal* as follows:

```
struct sigstack
   {
      char     *ss_sp;
      int      ss_onstack;
   };
```

If the **sigvec** function specifies that the signal handler execute on the signal stack, the system checks to see if the process is currently executing 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 sigstack struture is defined in the *signal* definition module.

Return Values

0	Indicates success.
–1	Indicates failure.

sigvec

The **sigvec** function assigns a handler for a specific signal.

Format

#include *signal*

int sigvec (**int** *sigint*, **struct sigvec** **sv*, **struct sigvec** **osv);*

Arguments

sigint
Is the signal identifier.

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 you.

Description

The sigvec structure is defined in the standard include module *signal* as follows:

```
struct sigvec
   {
      int   (*handler)();
      int   mask;
      int   onstack;
   };
```

Return Values

0	Indicates that the call succeeded.
–1	Indicates that an error occurred. Upon error, the variable errno contains the value explaining the error. See Chapter 4 for more information.

sin

The **sin** function returns the sine of its radian argument.

Format

#include *math*

double sin (**double** *x);*

Arguments

x
Is a radian expressed as a real number.

Description

Both the argument and the returned sine value must be an object of type **double**. If you use the *math* include module to declare **sin**, VAX C transforms the call into a direct call to MTH$DSIN_RT or MTH$GSIN_RT, depending on whether or not /G_FLOAT is specified on the CC command line.

sinh

The **sinh** function returns the hyperbolic sine of its argument.

Format

#include *math*

double sinh (**double** *x);*

Arguments

x
Is a real number.

Description

Both the argument and the returned hyperbolic sine value must be an object of type **double**.

The value of sine, if it causes an overflow, is a **double** value with the largest possible magnitude and the appropriate sign.

sleep

The **sleep** function suspends the execution of the current process for at least the number of seconds indicated by its argument.

Format

#include *signal*

int sleep *(***unsigned** *seconds);*

Arguments

seconds
Is the number of seconds.

Return Values

x	Indicates the number of seconds that the process slept.
–1	Indicates that an error occurred.

sprintf

The **sprintf** function performs formatted output to a string in memory. See Chapter 2 for information on format specifiers.

Format

#include *stdio*

int sprintf *(***char** **str,* **const char** **format_spec, . . .);*

Arguments

str
Is the address of the string that will receive the formatted output.

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

. . .
Are optional expressions whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, you may omit the output sources. 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 left-to-right order.

Description

A null character is automatically appended to the end of the output string. An example of a conversion specification is as follows:

```
main()
{
   int  temp = 4, temp2 = 17;
   char s[80];

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

The contents of character string s are as follows:

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

For a complete description of the format specification and the output source, see Chapter 2.

Return Values

x	Are characters placed in the output string not including the final null character.

sqrt

The **sqrt** function returns the square root of its argument.

Format

#include *math*

double sqrt (**double** *x*);

Arguments

x
Is a real number.

Description

The argument and the returned value are both objects of type **double**.

Return Values

0	Indicates that x is negative. The function sets the errno to EDOM.

srand

The **srand** function returns pseudorandom numbers in the range 0 to $2^{31} - 1$.

Format

#include *math*

int srand *(***int** *seed);*

Arguments

seed
Is an integer.

Description

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.

sscanf

The **sscanf** function performs formatted input from a character string in memory. See Chapter 2 for information on format specifiers.

Format

#include *stdio*

int sscanf *(***char** **str,* **const char** **format_spec, . . .);*

Arguments

str
Is the address of the character string that provides the input text to **sscanf**.

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

. . .
Are optional expressions whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, you can omit the input pointers. 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 pointers. Conversion specifications are matched to input sources in left-to-right order.

Description

An example of a conversion specification is as follows:

```
main ()
{
   char str[] = "4 17";
   int   temp, temp2;

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

This example produces the following output:

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

For a complete description of the format specification and the input pointers, see Chapter 2.

Return Values

x	Indicates the number of successfully matched and assigned input items.
EOF	Indicates that the end-of-file (or the end of the string) was encountered. EOF is a preprocessor constant defined in the *stdio* definition module.

ssignal

The **ssignal** function allows you to specify the action to take when a particular signal is raised.

Format

#include *signal*

void *(****ssignal** *(***int** *sig,* **void** *(*func) (***int,** *. . .))) (***int,** *. . .) ;*

Arguments

sig
Is a number or mnemonic associated with a signal. The symbolic constants for signal values are defined in the *signal* definition module (see Chapter 4).

func
Represents the action to take when the signal is raised, or the address of a function that is executed when the signal is raised.

Description

The **ssignal** function calls the **signal** function with the same arguments; the only difference between the two is in their return value on detecting an error (usually an invalid signal argument).

Return Values

x	Indicates the address of the function previously established as the action for the signal. The address may contain the value SIG_DFL (0) or SIG_IGN (1).
0	Indicates errors. For this reason, there is no way to know whether a return status of 0 indicates failure, or whether it indicates that a previous action was SIG_DFL (0). The **signal** function returns –1 on error.

[w]standend

The **standend** macro and the **wstandend** function deactivate the boldface attribute for the specified window.

Format

#include *curses*

standend( )

int wstandend **(WINDOW** ***win);***

Arguments

win
Is a pointer to the window.

Description

The **standend** macro and **wstandend** function are equivalent to **clrattr** and **wclrattr** called with the attribute _BOLD.

Return Values

1	Indicates success.
ERR	Indicates an error.

[w]standout

The **standout** macro and the **wstandout** function activate the boldface attribute of the specified window. The **standout** macro acts on the stdscr window.

Format

#include *curses*

standout( )

int wstandout *(***WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Description

The **standout** macro and **wstandout** function are equivalent to **setattr** and **wsetattr** called with the attribute _BOLD.

Return Values

1	Indicates success.
ERR	Indicates an error.

stat

The **stat** function accesses information about the file descriptor or the file specification.

Format

#include *stat*

int stat *(***char** **file_spec,* **stat_t** **buffer);*

Arguments

file_spec
Is a valid VMS or UNIX-style file specification. Read, write, or execute permission of the named file is not required, but you must be able to reach all directories listed in the file specification leading to the file. For more information about UNIX-style file specifications, see Chapter 1.

buffer
Is a pointer to a structure of type stat_t that 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 described as follows:

Member	Type	Definition
st_dev	unsigned	Pointer to the physical device name
st_ino[3]	unsigned short	Three words to receive the 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 0
st_size	unsigned	File size, in bytes

Member	Type	Definition
st_atime	unsigned	File access time; always the 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 st_mode, structure member, is the status information mode defined in the *stat* definition module. The st_mode bits are described 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

Description

The **stat** function does not work on remote network files.

Return Values

0	Indicates success.
–1	Indicates failure.
–2	Indicates a protection violation.

strcat

The **strcat** function concatenates str_2 to the end of str_1.

Format

#include *string*

char *strcat *(***char** **str_1,* **const char** **str_2);*

Arguments

str_1, str_2
Must be NUL-terminated character strings.

Description

See also **strncat** in this section.

Return Values

x	Indicates the address of the first argument, str_1, which is assumed to be large enough to hold the concatenated result.

Example

```
#include string
/* This program tests the strcat string function */
#define S1LENGTH 10
#define S2LENGTH 8
#define FILL_CHAR 'a'
#define TRUE 1
```

```
main()
{
static char s1buf[S1LENGTH+S2LENGTH] = "abcmnexyz";
static char s2buf[] = " orthis";
static char null_buf[S1LENGTH+1] = "";
static char test1[] = "abcmnexyz orthis";
static char test3_5[] = "abcmnexyz";

int i, testnum;
char temp;
char *status;


/* this test uses static buffer s1buf,
 *   concatenates static buffer s2buf to it,
 *  and compares the answer in s1buf with
 *  the static answer in test1
 */
 testnum = 1;
 status = strcat(s1buf, s2buf);

 /* check for correct returned address */
 if (status == &s1buf)
 {
  for (i = 0; i <= S1LENGTH+S2LENGTH-2; i++)
  {
   /* check for correct returned string - test 1 */
   if (test1[i] != s1buf[i])
    printf("error in strcat");
  }
 }
    else
  printf("error in strcat");

}
```

strchr

The **strchr** function returns the address of the first occurrence of a given character in a NUL-terminated string.

Format

#include *string*

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

Arguments

str
Is a pointer to a NUL-terminated character string.

character
Is an object of type **int**.

Description

See also **strrchr** in this section.

Return Values

x	Indicates the address of the first occurrence of the specified character.
NULL	Indicates that the character does not occur in the string.

Example

```
#include stdio
#include string

main()
{
static char s1buf[] = "abcdefghijkl lkjihgfedcba";
static char s2buf[] = {'a','b','c',' ','\t','\n','z','\n','\t',' ','c','b','a'};
static char s3buf[] = "mnopqrstuvwxyz0123456789A";

int i, testnum;
char *status;


/*  this test checks the STRCHR function by incrementally going
 *  through a string that ascends to the middle and then
 *  descends towards the end
 */
 testnum = 1;
 for (i = 0; s1buf[i] != '\0' && s1buf[i] != ' '; i++)
 {
  status = strchr(s1buf, s1buf[i]);
  /* check for pointer to leftmost character - test 1 */
  if (status != &s1buf[i])
   printf("error in strchr");
 }
```

strcmp

The **strcmp** function compares two ASCII character strings and returns a negative, 0, 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.

Format

#include *string*

int strcmp *(***const char** **str_1,* **const char** **str_2);*

Arguments

str_1, str_2
Are pointers to character strings.

Description

The strings are compared until a null character is encountered or until the strings differ.

Return Values

< 0	Indicates that str1 is less than str2.
= 0	Indicates that str1 equals str2.
> 0	Indicates that str1 is greater than str2.

strcpy

The **strcpy** function copies all of str_2 into str_1.

Format

#include *string*

char *strcpy *(***char** **str_1,* **const char** **str_2);*

Arguments

str_1, str_2
Are pointers to character strings.

Description

The **strcpy** function copies str_2 into str_1, and stops after copying str_2's null character.

The behavior of this function is undefined if the area pointed to by str_1 overlaps the area pointed to by str_2.

Return Values

x	Indicates the address of str_1.

strcspn

The **strcspn** function returns the length of the prefix of a string which consists entirely of characters that are not in a specified set of characters.

Format

#include *string*

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

Arguments

str
Is a pointer to a character string. If the argument string is a null string, 0 is returned.

charset
Is a pointer to a character string containing the set of characters.

Description

The **strcspn** function scans the characters in the string, stops when it encounters a character found in charset, and returns the length of the string's initial segment formed by characters not found in charset.

If none of the characters match in the character strings pointed to by str and charset, **strcspn** returns the length of string.

Return Values

x	Indicates the length of the segment.

strerror

The **strerror** function maps the error number in error_code to an error message string.

Format

#include *string*

char *strerror *(***int** *error_code [,* **int** *vms_error_code]);*

Arguments

error_code
Is an error code.

vms_error_code
Is a VMS error code.

Description

If the first argument is the *errno* value EVMSERR and there is a second argument, the **strerror** function calls the $GETMSG system service to translate the error code into the VMS message text. Otherwise, the UNIX type message is returned. Use of the second argument is not portable.

Return Values

x	Indicates a pointer to a buffer containing the appropriate error message. Do not modify this buffer in your programs. Moreover, calls to the **strerror** function may overwrite this buffer with a new message.
NULL	Indicates that the argument errnum does not correspond to a known RTL error code.

strlen

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

Format

#include *string*

size_t strlen **(const char** **str);*

Arguments

str
Is a pointer to the character string.

Return Values

x	Indicates the length of the string.

strncat

The **strncat** function concatenates str_2 to the end of str_1.

Format

#include *string*

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

Arguments

str_1, str_2
Must be NUL-terminated character strings.

maxchar
Specifies the number of characters to concatenate from str_2, unless the **strncat** first encounters a null terminator in str_2. If maxchar is 0 or negative, no characters are copied from str_2.

Description

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

Return Values

x	Indicates the address of the first argument, str_1, which is assumed to be large enough to hold the concatenated result.

strncmp

The **strncmp** function compares two ASCII character strings and returns a negative, 0, 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.

Format

#include *string*

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

Arguments

str_1, str_2
Are 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 0 or negative, no comparison is performed and 0 is returned (the strings are considered equal).

Description

The strings are compared until a null character is encountered, the strings differ, or maxchar is reached. The comparison is terminated when a NUL character is encountered in one of the strings.

Return Values

< 0	Indicates that str1 is less than str2.
= 0	Indicates that str1 equals str2.
> 0	Indicates that str1 is greater than str2.

strncpy

The **strncpy** function copies all or part of str_2 into str_1.

Format

#include *string*

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

Arguments

str_1, str_2
Are 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 null terminator of str_2.

Description

The **strncpy** function 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.

NOTE

The str_1 argument may not be terminated by a null character after a call to **strncpy**.

Return Values

x	Indicates the address of str_1.

strpbrk

The **strpbrk** function searches a string for the occurrence of one of a specified set of characters.

Format

#include *string*

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

Arguments

str
Is a pointer to a character string. If the argument string is a null string, 0 is returned.

charset
Is a pointer to a character string containing the set of characters for which the function will search.

Description

The **strpbrk** function scans the characters in the string, stops when it encounters a character found in charset, and returns the address of the first character in the string that appears in the character set.

Return Values

x	Indicates the address of the first character in the string that is in the set.
NULL	Indicates that no character is in the set.

strrchr

The **strrchr** function returns the address of the last occurrence of a given character in a NUL-terminated string.

Format

#include *string*

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

Arguments

str
Is a pointer to a NUL-terminated character string.

character
Is an object of type **int**.

Description

See also **strchr** in this section.

Return Values

x	Indicates the address of the last occurrence of the specified character.
NULL	Indicates that the character does not occur in the string.

strspn

The **strspn** function returns the length of the prefix of a string that consists entirely of characters from a set of characters.

Format

#include *string*

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

Arguments

str
Is a pointer to a character string. If the argument string is a null string, 0 is returned.

charset
Is a pointer to a character string containing the characters for which the function will search.

Description

The **strspn** function scans the characters in the string, stops when it encounters a character not found in charset, and returns the length of the string's initial segment formed by characters found in charset.

Return Values

x	Indicates the length of the segment.

strstr

The **strstr** function locates the first occurrence in the string pointed to by s1 of the sequence of characters in the string pointed to by s2.

Format

#include *string*

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

Arguments

s1
Is a string.

s2
Is a string.

Return Values

Pointer	Is a pointer to the located string.
null pointer	Indicates that the string was not found.

Example

```
#include stdio

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

main()
    {
    static char lookin[]="that this is a test was at the end";
```

strstr

```
putchar('\n');
printf("String: %s\n", &lookin[0] );
putchar('\n');
printf("Addr: %s\n", &lookin[0] );
printf("this: %s\n", strstr( &lookin[0] ,"this") );
printf("that: %s\n", strstr( &lookin[0] , "that" ) );
printf("NULL: %s\n", strstr( &lookin[0], "" ) );
printf("was: %s\n", strstr( &lookin[0], "was" ) );
printf("at: %s\n", strstr( &lookin[0], "at" ) );
printf("the end: %s\n", strstr( &lookin[0], "the end") );
putchar('\n');

exit();
};
```

strtod

The **strtod** function converts a given string to a double-precision number.

This function recognizes 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 used to interpret floating constants.

Format

#include *stdlib*

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

Arguments

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

endptr
Is the address of an object where the function can store 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.

strtod

Description

The **strtod** function returns the converted value. For **strtod,** overflows are accounted for as follows:

- If the correct value causes an 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 causes an underflow, 0 is returned and errno is set to ERANGE.

If the string starts with an unrecognized character, *endptr is set to nptr and a 0 value is returned.

Return Values

x	Specifies the converted string.
0	Indicates an error.

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. This function keeps track of its position in the string between calls and, as successive calls are made, the function works through the string, identifying the text token following the one identified by the previous call.

Format

#include *string*

char *strtok *(***char** **s1,* **const char** **s2);*

Arguments

s1
Is a pointer to a string containing 0 or more text tokens.

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

Description

The first call to the **strtok** function returns a pointer to the first character in the first token and writes a null 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.

Tokens in s1 are delimited by null characters inserted into s1 by the **strtok** function. Therefore, s1 cannot be a **const** object. The **strtok** function is nonreentrant since you must use a static global variable to maintain the starting address within s1 of subsequent calls to **strtok** with a null first argument.

Return Values

x	Specifies a pointer to the first character of a token.
NULL	Indicates that no token was found.

strtol

The **strtol** function converts strings of ASCII characters to the appropriate numeric values.

Format

#include *stdlib*

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

Arguments

nptr
Is a pointer to the character string to be converted to a **long**.

endptr
Is the address of an object where the function can store 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
Is the value, 2 through 36, to use 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 0 indicates octal conversion, a leading 0x or 0X indicates hexadecimal conversion, and any other combination of leading characters indicates decimal conversion.

Description

The **strtol** function recognizes strings in various formats, depending on the value of the base. This 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.

Truncation from **long** to **int** can take place after assignment or by an explicit cast (arithmetic exceptions not withstanding). The function call **atol (str)** is equivalent to **strtol (str, (char**)0, 10)**.

Return Values

x	Indicates the converted value.
LONG_MAX or LONG_MIN	Indicate that the correct value will cause an overflow (according to the sign of the value). Errno is set to ERANGE. These values are defined in the *limits* standard include module.
0	Indicates that the string starts with an unrecognized character. *endptr is set to nptr.

strtoul

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

Format

#include *stdlib*

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

Arguments

nptr
Is a pointer to the character string to be converted to a **long**.

endptr
Is the address of an object where the function can store 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
Is the value, 2 through 36, to use 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 0 indicates octal conversion, a leading 0x or 0X indicates hexadecimal conversion, and any other combination of leading characters indicates decimal conversion.

Return Values

x	Indicates the converted value.
0	Indicates that no conversion was performed.
ULONG_MAX	Indicates that an overflow occurred; errno is set to erange. ULONG_MAX is defined in the *limits* standard include module.

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.

Format

#include *curses*

WINDOW *subwin (**WINDOW** **win,* **int** *numlines,* **int** *numcols,* **int** *begin_y,* **int** *begin_x);*

Arguments

win
Is a pointer to the window.

numlines
If it is 0, then the function sets that dimension to LINES (begin_y). To get a new window of dimensions LINES by COLS, use the following format:

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

numcols
If it is 0, then the function sets that dimension to COLS (begin_x). To get a new window of dimensions LINES by COLS, use the following format:

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

begin_y
Is a window coordinate.

begin_x
Is a window coordinate.

subwin

Description

When creating the subwindow, begin_y and begin_x are relative to the entire terminal screen. If either numlines or numcols is 0, then the **subwin** 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.

Return Values

x	Specifies a pointer to an instance of the structure window.
0	Indicates an error.

system

The **system** function passes a given string to the host environment to be executed by a command processor.

Format

#include *processes*

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

Arguments

string
Is a pointer to the string to be executed.

Description

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.

Return Values

nonzero value	If the string is a null pointer, then the **system** function is supported.
0	If you get a 0 value, then the **system** function is not supported.

Example

```
#include processes

main ()
{
 int status, fd;

 fd = creat ("system.test", 0);
 write (fd, "this is an example of using system", 34);
 close (fd);

 status = system ("DIR/NOHEAD/NOTRAIL/SIZE SYSTEM.TEST");
 printf ("system status = %d\n", status);
}
```

tan

The **tan** function returns a **double** value that is the tangent of its radian argument.

Format

#include *math*

double tan *(***double** *x);*

Arguments

x
Is a radian expressed as a real number.

Description

The value of tan(x) at its singular points (. . . $-3\pi/2,-\pi/2,\pi/2$ . . .) is the largest possible **double** value HUGE_VAL, defined in the *math* include module; the value of errno is set to ERANGE when x is a singular point.

tanh

The **tanh** function returns a **double** value that is the hyperbolic tangent of its **double** argument.

Format

#include *math*

double tanh (**double** *x*);

Arguments

x
Is a real number.

Description

If you use the *math* include module to declare **tanh**, VAX C transforms the call into a direct call to MTH$DTANH or MTH$GTANH, depending on whether or not /G_FLOAT is specified on the CC command line.

time

The **time** function returns the time elapsed since 00:00:00, January 1, 1970, in seconds.

Format

#include *time*

time_t time (**time_t** **time_location);*

Arguments

time_location
Is either NULL or a pointer to the place where the returned time is also stored.

Return Values

x	Specifies the time elapsed past epoch.
0	Indicates an error.

times

The **times** function passes back the accumulated times of the current process and its terminated child processes.

Format

#include *time*

void times (**tbuffer_t** **buffer*);

Arguments

buffer
Is a pointer to the terminal buffer.

Description

The type **tbuffer_t** is defined in the standard include module *time.h* as follows:

```
struct tbuffer
    {
       int proc_user_time;
       int proc_system_time;
       int child_user_time;
       int child_system_time;
    }typedef struct tbuffer tbuffer_t;
```

For both process and children times, the structure breaks down the time by user and system time. Since the VMS system does not differentiate between system and user time, all system times are returned as 0. Accumulated CPU times are returned in 10-millisecond units.

tmpfile

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

Format

#include *stdio*

FILE *tmpfile *(***void***);*

Description

The file exists only for the duration of the process and is preserved across vforks.

Return Values

x	Indicates the address of a FILE pointer (defined in the *stdio* definition module).
NULL	Indicates an error.

tmpnam

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

Format

#include *stdio*

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

Arguments

name
Is a character string containing a name to use in place of file-name arguments in other functions or macros. Successive calls to **tmpnam** with a null argument cause the function to overwrite the current name.

Return Values

x	If the name argument is the null pointer value NULL, **tmpnam** returns the address of an internal storage area. If the name is not NULL, then it is considered the address of an area of length L_tmpnam (defined in the *stdio* definition module). In this case, **tmpnam** returns the name argument as the result.

toascii

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

Format

#include *ctype*

int toascii *(***char** *character);*

Arguments

character
Is an object of type **char**.

Return Values

x	Specifies a 7-bit ASCII character.

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.

Format

#include *ctype*

int tolower *(***char** *character);*

int _tolower *(***char** *character);*

Arguments

character
Is an object of type **char**.

Description

The **_tolower** macro should not be used with arguments that contain side-effect operations. For instance, the following example will not return the expected result:

```
d = _tolower (c++);
```

touchwin

The **touchwin** function places the most recently edited version of the specified window on the terminal screen.

Format

#include *curses*

int touchwin (**WINDOW** **win);*

Arguments

win
Is a pointer to the window.

Description

The **touchwin** function is normally used only to refresh overlapping windows.

Return Values

1	Indicates success.
0	Indicates an error.

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.

Format

#include *ctype*

int toupper (**char** *character);*

int _toupper (**char** *character);*

Arguments

character
Is an object of type **char**.

Description

You only have to include the *ctype* definition module if you are using the **_toupper** macro.

The **_toupper** macro should not be used with arguments that contain side-effect operations. For instance, the following example will not return the expected result:

```
d = _toupper (c++);
```

ttyname

The **ttyname** function returns a pointer to the NUL-terminated name of the terminal device associated with file descriptor 0, the default input device (stdin).

Format

#include *unixio*

char *ttyname (void);

Description

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

Return Values

x	Specifies a pointer to a NUL-terminated string.
0	Indicates that SYS$INPUT is not a TTY device.

umask

The **umask** function creates a file protection mask that is used when a new file is created, and returns the previous mask value.

Format

#include *stdlib*

int umask **(unsigned int** *mode_complement);*

Arguments

mode_complement
Shows which bits to turn off when a new file is created. See the description of **chmod** to determine what the bits represent.

Description

Initially, the file protection mask is set from the current process's default file protection. This is done when the C main program starts up or when **VAXC$CRTL_INIT** is called. You can change this for all files created by your program by calling **umask** or you can use **chmod** to change the file protection on individual files. The file protection of a file created by **open** or **creat** is the bitwise AND of the **open** and **creat** mode argument with the complement of the value passed to **umask** on the previous call.

Return Values

x	Indicates the old mask value.

ungetc

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

Format

#include *stdio*

int ungetc (**int** *character,* **FILE** **file_ptr);*

Arguments

character
Is a value of type **int**.

file_ptr
Is a file pointer.

Description

When using the **ungetc** function, the character is pushed back onto the file, since it is returned by the next **getc** call.

One push-back is guaranteed, even if there has been no previous activity on the file. The **fseek** function erases all memory of pushed-back characters. The pushed-back character is not written to the underlying file. If the character to be pushed back is EOF, the operation fails, the input stream is left unchanged, and EOF is returned.

See also the **fseek** and **getc** functions in this section.

Return Values

x	Indicates the push-back character.
EOF	Indicates it cannot push the character back.

VAXC$CALLOC_OPT

The **VAXC$CALLOC_OPT** function allocates an area of memory.

Format

#include *stdlib*

void *VAXC$CALLOC_OPT *(***size_t** *number,* **size_t** *size);*

Arguments

number
Specifies the number of items to be allocated.

size
Is the size of each item.

Description

The **VAXC$CALLOC_OPT** function initializes the items to 0. For more information, see the **VAXC$MALLOC_OPT** function in this section.

Return Values

0	Indicates an inability to allocate the space.
n	Indicates the address of the first byte, which is aligned on an octaword boundary.

VAXC$CFREE_OPT

The **VAXC$CFREE_OPT** function makes available for reallocation the area allocated by a previous **VAXC$CALLOC_OPT**, **VAXC$MALLOC_OPT**, or **VAXC$REALLOC_OPT** call.

Format

#include *stdlib*

int VAXC$CFREE_OPT **(void** **ptr);*

Arguments

ptr
Is the address returned by a previous call to **VAXC$MALLOC_OPT**, **VAXC$CALLOC_OPT**, or **VAXC$REALLOC_OPT**.

Description

The contents of the deallocated area are unchanged. For more information, see the **VAXC$MALLOC_OPT** function in this section.

Return Values

0	Indicates that the area is successfully freed.
1	Indicates an error.

VAXC$CRTL_INIT

The **VAXC$CRTL_INIT** function allows you to call the VAX C RTL from other languages or to use the VAX C RTL when your main function is not in C. It initializes the run-time environment and establishes both an exit and condition handler.

Description

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.
```

It is not recommended that you make multiple calls to the **VAXC$CRTL_INIT** function. A shareable image should only call this function if it contains a VAX C function for exception handling, environment variables, or a default file protection mask.

VAXC$ESTABLISH

The **VAXC$ESTABLISH** function establishes a special VAX C RTL exception handler that catches all RTL-related exceptions and passes on all others to your handler. This routine is necessary when using certain VAX C RTL UNIX emulation routines.

Format

#include *signal*

void VAXC$ESTABLISH **(int** *(*exception_handler)***(void** **sigarr,* **void** **mecharr));*

Arguments

exception_handler
Is the name of the function to establish as a VMS condition handler. You pass the address of a function as an argument to **VAXC$ESTABLISH.**

sigarr
Is a pointer to the signal array.

mecharr
Is a pointer to the mechanism array.

Description

You can only invoke the **VAXC$ESTABLISH** function from a VAX C function, as it relies on the allocation of data space on the run-time stack by the VAX C compiler. Calling the VMS system library routine LIB$ESTABLISH directly from a VAX C function results in undefined results by the **setjmp** and **longjmp** functions.

VAXC$ESTABLISH must be used in place of LIB$ESTABLISH when programs use the VAX C RTL routines **setjmp** or **longjmp**. See the **setjmp** and **longjmp** functions in this section.

To cause a VMS exception or signal to generate a UNIX-style signal, user condition handlers must return SS$_RESIGNAL upon receiving any exception that they do not want to handle. Returning SS$_NORMAL prevents the generation of a UNIX-style signal. UNIX signals are generated as if by an exception handler in the stack frame of the main C program. Not all VMS exceptions correspond to UNIX signals.

VAXC$FREE_OPT

The **VAXC$FREE_OPT** function makes available for reallocation the area allocated by a previous **VAXC$CALLOC_OPT**, **VAXC$MALLOC_OPT**, or **VAXC$REALLOC_OPT** call.

Format

#include *stdlib*

int VAXC$FREE_OPT *(**void** *ptr);*

Arguments

ptr
Is the address returned by a previous call to **VAXC$MALLOC_OPT**, **VAXC$CALLOC_OPT**, or **VAXC$REALLOC_OPT**.

Description

The contents of the deallocated area are unchanged. For more information, see the **VAXC$MALLOC_OPT** function in this section.

Return Values

0	Indicates that the area is successfully freed.
–1	Indicates an error.

VAXC$MALLOC_OPT

The **VAXC$MALLOC_OPT** function allocates an area of memory.

Format

#include *stdlib*

void *VAXC$MALLOC_OPT **(size_t** *size);*

Arguments

size
Is the total number of bytes to be allocated.

Description

The **VAXC$MALLOC_OPT** function allocates a contiguous area of memory whose size, in bytes, is supplied as an argument. This routine takes advantage of memory-management routines (LIB$GET_VM and LIB$FREE_VM zone allocation) that are in the VMS RTL. The performance and function of these routines are an improvement to the previous functionality provided. The zone algorithm used is first fit with no boundary tag. Each allocation is zero filled and aligned on an octaword boundary. This implementation may change in a future release of VAX C.

The malloc_opt routine makes no attempt to support the previous behavior of malloc. An example of such behavior is to sequence a freeing of dynamic memory followed by an access of that memory.

An easy way to use these routines without rewriting the function calls is to include the following macro definitions at the beginning of your program:

```
#define malloc VAXC$MALLOC_OPT
#define calloc VAXC$CALLOC_OPT
#define free VAXC$FREE_OPT
#define cfree VAXC$CFREE_OPT
#define realloc VAXC$REALLOC_OPT
```

These functions are not interchangeable with **malloc**, **calloc**, **free**, **cfree**, and **realloc**.

Return Values

0	Indicates that it is unable to allocate enough memory.
x	The address of the first byte, which is aligned on an octaword boundary.

VAXC$REALLOC_OPT

The **VAXC$REALLOC_OPT** function changes the size of the area pointed to by the first argument to the number of bytes given by the second argument.

Format

#include *stdlib*

void *VAXC$REALLOC_OPT *(***void** **ptr,* **size_t** *size);*

Arguments

ptr
May point to an allocated area only.

size
Specifies the new size of the allocated area.

Description

This function will not reallocate memory that has been previously freed by **VAXC$FREE_OPT** or by **VAXC$CFREE_OPT**.

See the **VAXC$MALLOC_OPT** function in this section for more information.

Return Values

x	Indicates 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.
0	Indicates that it is unable to reallocate the space (for example, if there is not enough room).

va_arg

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

Format

#include *stdarg*

#include *varargs*

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

Arguments

ap
Is a variable list containing the next argument to be obtained.

type
Is 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.

Description

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

NOTE

On VMS systems, all items in an argument list are aligned on the longword boundary. If you try 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 about the VAX Calling Standard, see the *Guide to VAX C*.

va_count

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

Format

#include *varargs*

void va_count *(***int** *count);*

Arguments

count
Is an integer variable name in which the number of longwords is returned.

Description

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

va_end

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

Format

#include *stdarg* **or**

#include *varargs*

void va_end *(***va_list** *ap);*

Arguments

ap
Is the object used to traverse the argument list length. You must declare and use the argument ap as shown in the format section.

Description

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

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.

Format

#include *varargs*

void va_start *(***va_list** *ap);*

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

Arguments

ap
Is an object pointer. You must declare and use the argument ap as shown in the format section.

offset
Represents the number of bytes that 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.

Description

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 that 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 about the VAX Calling Standard, see the *Guide to VAX C*.

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

Format

#include *stdargs*

void va_start *(***va_list** *ap, parmN);*

Arguments

ap
Is an object pointer. You must declare and use the argument ap as shown in the format section.

parmN
Is the name of the last of the known fixed arguments.

Description

The pointer ap is initialized to point to the first of the optional arguments that follow parmN in the argument list. Always use this version of **va_start** in conjunction with functions that are declared and defined with function prototypes.

vfork

The **vfork** function creates an independent child process.

Format

#include *processes*

int vfork *(***void***);*

Description

The **vfork** function provided by VAX C differs from the **fork** function provided by other C implementations. The two major differences are shown in Table REF–8.

Table REF–8: The vfork and fork Functions

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 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 (before the call to an exec function is made). After the corresponding exec function call is 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. Unless the exec function fails, control appears 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.

Return Values

0	Indicates successful creation of context.
nonzero	Indicates the process ID (PID) of the child process.
–1	Indicates an error—failure to create the child process.

vfprintf

The **vfprintf** function prints formatted output based on an argument list.

This function is the same as the **fprintf** function except that instead of being called with a variable number of arguments, it is called with an argument list that has been initialized by the macro **va_start** (and possibly subsequent va_arg calls).

See Chapter 2 for information on format specifiers.

Format

#include *stdio*

#include *stdarg*

int vfprintf (**FILE** **file_ptr*, **const char** **format*, **va_list** *arg);*

Arguments

file_ptr
Is a pointer to a file.

format
Is a pointer to a string containing the format specification.

arg
Is a list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

See the **vprintf** and **vsprintf** functions in this section.

Return Values

x	Indicates the number of characters transmitted.
EOF	Indicates an output error.

vprintf

The **vprintf** function prints formatted output based on an argument list.

This function is the same as the **printf** function except that instead of being called with a variable number of arguments, it is called with an argument list that has been initialized by the macro **va_start** (and possibly subsequent va_arg calls).

See Chapter 2 for information on format specifiers.

Format

#include *stdio*

#include *stdarg*

int vprintf *(***const char** **format,* **va_list** *arg);*

Arguments

format
Is a pointer to the string containing the format specification.

arg
Is a variable list of the items needed for output.

Description

See the **vfprintf** and **vsprintf** functions this section.

Return Values

x	Indicates the number of characters transmitted.
EOF	Indicates an output error.

vsprintf

The **vsprintf** function prints formatted output based on an argument list.

This function is the same as the **sprintf** function except that instead of being called with a variable number of arguments, it is called with an argument list that has been initialized by the macro **va_start** (and possibly subsequent va_arg calls).

Format

#include *stdio*

#include *stdarg*

int vsprintf *(***char** **str,* **const char** **format,* **va_list** *arg);*

Arguments

str
Is a pointer to a string.

format
Is a format specification.

arg
Is a list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Return Values

x	Indicates the number of characters transmitted.
EOF	Indicates an output error.

wait

The **wait** function checks the status of the child process before exiting. A child process is terminated when the parent process terminates.

Format

#include *processes*

int wait *(***int** **status);*

Arguments

status
Is the address of a location to receive the final status of the terminated child. The child can set the status with the **exit** function and the parent can retrieve this value by specifying status.

Description

The **wait** function suspends the parent process until a value is returned from the child. This value is the final status of the child.

Return Values

x	Indicates the process ID (PID) of the terminated child. If more than one child process was created, **wait** will return the PID of the terminated child that was most recently created. Subsequent calls will return the PID of the next most recently created, but terminated, child.

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 serves no function in the VMS environment.

Format

#include *curses*

#define bool int

wrapok **(WINDOW** **win,* **bool** *boolf);*

Arguments

win
Is a pointer to the window.

boolf
Is a Boolean TRUE or FALSE value. If boolf is FALSE, scrolling is not allowed. This is the default setting. The boolf argument is defined in the *curses* definition module.

write

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

Format

#include *unixio*

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

Arguments

file_desc
Is a file descriptor. The specified file descriptor must refer to a file currently opened for writing or updating.

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

nbytes
Is the maximum number of bytes involved in the write operation.

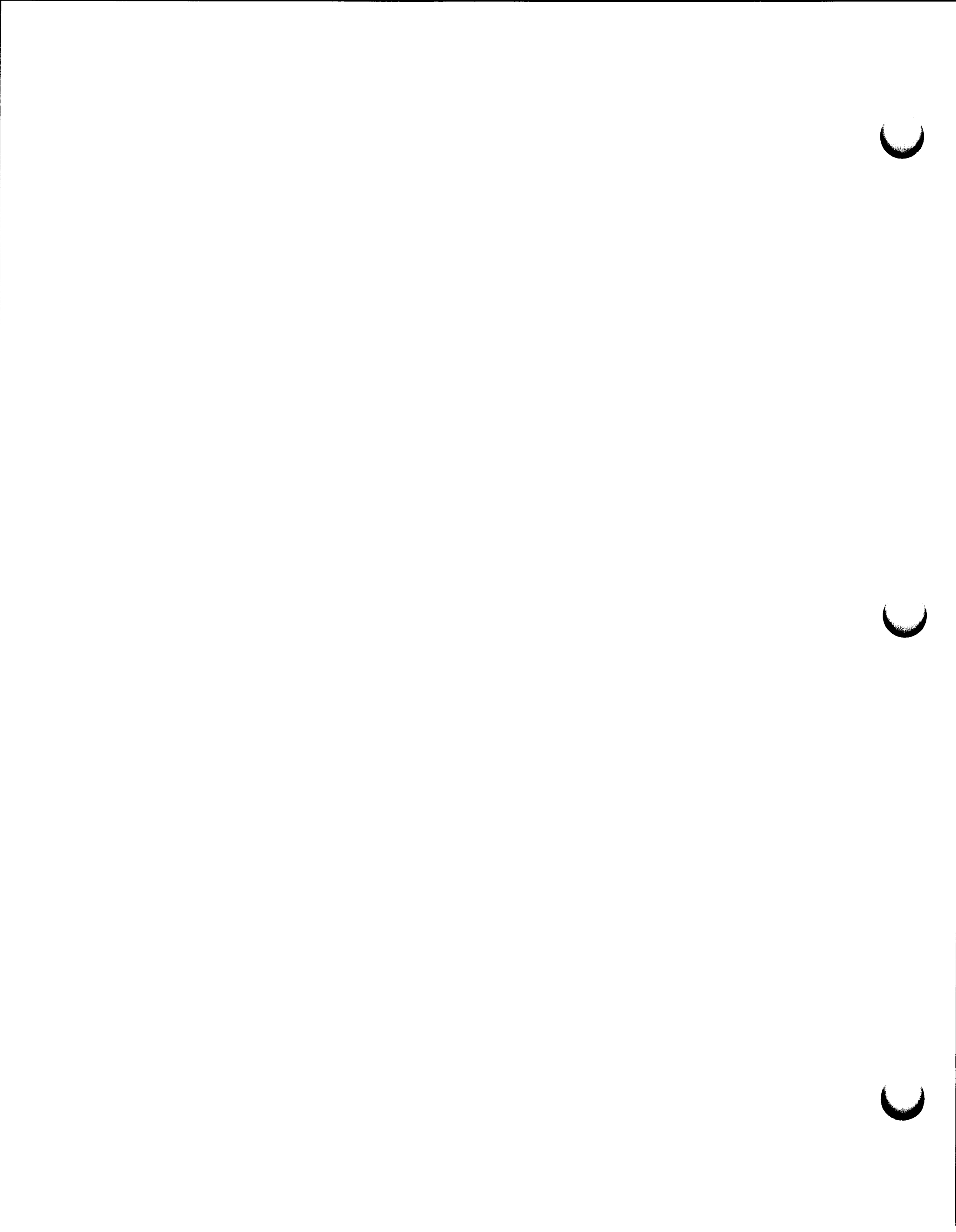
Description

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 will be written to the file. The **write** function always generates at least one record.

If the write is to a mailbox and the third argument, nbytes, specifies a length of 0, an end-of-file message is written to the mailbox. For more information, see Chapter 5.

Return Values

x	Indicates the number of bytes written.
–1	Indicates errors, including undefined file descriptors, illegal buffer addresses, and physical I/O errors.



Appendix A

VAX C RTL and RTLs of Other C Implementations

Most implementations of the C programming language provide some form of the run-time functions and macros found 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

C Function	VAX C Implementation
abort	Not equivalent.
	The VMS system does not generate a core dump.
abs	Equivalent functionality.
access	Equivalent functionality.
acct	Not provided.
	Not provided in the VAX C RTL. 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	Equivalent functionality.
[w]addch	Equivalent functionality.
[w]addstr	Equivalent functionality.
alarm	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
asctime	Equivalent functionality.
asin	Equivalent functionality.
assert	Equivalent functionality.
atan	Equivalent functionality.
atan2	Equivalent functionality.
atexit	Defined in the Draft Proposed ANSI C Standard.
atof	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	See **atof**.
atol	See **atof**.
box	Equivalent functionality.
brk	See **sbrk**.
cabs	Equivalent functionality.
calloc	Equivalent functionality.
ceil	Equivalent functionality.
cfree	Equivalent functionality.
chdir	Not equivalent. The VAX C version changes the default directory for your program only. You will still have the same default directory as before the call. On VMS systems, use the DCL SET DEFAULT command.
chmod	Not equivalent. VMS systems have no equivalent to the "set user ID", "set group ID", or "save text" file attributes. You can individually specify group and system read, write, and execute protection. Perform a **chmod** to 1000 ("save text") on VMS systems using the INSTALL utility.
chown	Equivalent functionality.
circle	Not provided.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
[w]clear	Equivalent functionality.
clearerr	Equivalent functionality.
clearok	Equivalent functionality.
clock	Equivalent functionality.
close	Equivalent functionality.
closepl	Not provided.
[w]clrattr	VAX C specific.
[w]clrtobot	Equivalent functionality.
[w]clrtoeol	Equivalent functionality.
cont	Not provided.
cos	Equivalent functionality.
cosh	Equivalent functionality.
creat	Not equivalent.
	VAX C adds optional file attributes that let you create files with RMS formats other than stream.
[no]crmode	Equivalent functionality.
crypt	Not provided.
ctermid	Equivalent functionality.
ctime	Equivalent functionality.
cuserid	Equivalent functionality.
dbm	Not provided.
[w]delch	Equivalent functionality.
delete	VAX C specific.
[w]deleteln	Equivalent functionality.
delwin	Equivalent functionality.
difftime	Defined in the Draft Proposed ANSI C Standard.
div	Equivalent functionality.
dup	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
dup2	Equivalent functionality.
[no]echo	Equivalent functionality.
ecvt	Equivalent functionality.
endfsent	Not provided.
endgrent	Not provided.
endpwent	Not provided.
endwin	Equivalent functionality.
[w]erase	Equivalent functionality.
exec	See **execve**.
execl	See **execve**.
execlp	See **execve**.
execle	See **execve**.
execv	See **execve**.
execve	Not equivalent. The principle of process overlaying is not used in VMS systems. 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	See **execve**.
exit	Not equivalent. If you invoke the process with the DCL command interpreter, the VMS system interprets the return value and prints a DCL message.
exp	Equivalent functionality.
fabs	Equivalent functionality.
fclose	Equivalent functionality.
fcvt	Equivalent functionality.
fdopen	Equivalent functionality.
feof	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
ferror	Equivalent functionality.
fflush	Equivalent functionality.
fgetc	Equivalent functionality.
fgetname	Not equivalent. VAX C returns either the VMS file specification or the DEC/Shell file specification.
fgets	Equivalent functionality.
fileno	Equivalent functionality.
floor	Equivalent functionality.
fmod	Equivalent functionality.
fopen	Not equivalent. VAX C adds optional file attributes that let you create files with RMS formats other than stream.
fork	Not provided (see **vfork**).
fprintf	Equivalent functionality.
fputc	Equivalent functionality.
fputs	Equivalent functionality.
fread	Equivalent functionality.
free	Equivalent functionality.
freopen	Not equivalent. VAX C adds optional file attributes that let you create files with RMS formats other than stream.
frexp	Equivalent functionality.
fscanf	Not equivalent. VAX C provides the following conversion characters: hd, ho, hx, ld, lo, lx, le, lf, i, n, and p.
fseek	Not equivalent. When using record files, input from **ftell** is required for VAX C.
fstat	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
ftell	Not equivalent. When using record files, VAX C returns the position of the current record.
ftime	Equivalent functionality.
fwrite	Equivalent functionality.
gamma	Not provided.
gcvt	Equivalent functionality.
getc	Equivalent functionality.
[w]getch	Equivalent functionality.
getchar	Equivalent functionality.
getcwd	Equivalent functionality.
getegid	See **getuid**.
getenv	Equivalent functionality.
geteuid	See **getuid**.
getfsent	Not provided.
getfsfile	Not provided.
getfsspec	Not provided.
getgid	See **getuid**.
getgrent	Not provided.
getgrgid	Not provided.
getgrnam	Not provided.
getlogin	Not provided.
getname	Not equivalent. VAX C returns either the VMS file specification or the DEC/Shell file specification.
getpass	Not provided.
getpgrp	Not provided.
getpid	Equivalent functionality.
getppid	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
getpw	Not provided.
getpwent	Not provided.
getpwnam	Not provided.
getpwuid	Not provided.
getrgid	Not provided.
gets	Equivalent functionality.
[w]getstr	Equivalent functionality.
getuid	Not equivalent. VAX C returns the group and member codes from the UIC; VMS systems do not distinguish between real and effective user IDs.
getw	Equivalent functionality.
getyx	Equivalent functionality.
gmtime	Provided with no functionality.
gsignal	VAX C specific.
hypot	Equivalent functionality.
[w]inch	Equivalent functionality.
index	Not provided.
initscr	Equivalent functionality.
[w]insch	Equivalent functionality.
[w]insertln	Equivalent functionality.
[w]insstr	VAX C specific.
ioctl	Not provided.
isalnum	Equivalent functionality.
isalpha	Equivalent functionality.
isapipe	Equivalent functionality.
isascii	Equivalent functionality.
isatty	Equivalent functionality.
iscntrl	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
isdigit	Equivalent functionality.
isgraph	Equivalent functionality.
islower	Equivalent functionality.
isprint	Equivalent functionality.
ispunct	Equivalent functionality.
isspace	Equivalent functionality.
isupper	Equivalent functionality.
isxdigit	Equivalent functionality.
j0, j1, jn	Not provided.
kill	Not equivalent. VMS systems require 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	Equivalent functionality.
ldiv	Equivalent functionality.
leaveok	Equivalent functionality.
link	Not provided.
line	Not provided.
linemod	Not provided.
localtime	Not equivalent. On VAX C, daylight savings time always equals 0.
log, log10	Equivalent functionality.
longjmp	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
longname	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	Not equivalent. The VAX C function positions on record boundaries for RMS record files.
ltol3	Not provided.
malloc	Not equivalent. VAX C aligns the area returned on an octaword boundary.
memchr	Equivalent functionality.
memcmp	Equivalent functionality.
memcpy	Equivalent functionality.
memmove	Equivalent functionality.
memset	Equivalent functionality.
mkdir	Not equivalent. VAX C includes VMS-specific optional arguments to specify the UIC, the maximum file version number, and the relative volume number.
mknod	Not provided.
mktemp	Equivalent functionality.
modf	Equivalent functionality.
monitor	Not provided.
mount, umount	Not provided.
[w]move	Equivalent functionality.
mpx	Not provided.
mv[w]addch	Equivalent functionality.
mv[w]addstr	Equivalent functionality.
mvcur	Equivalent to the function **move**.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
mv[w]delch	Equivalent functionality.
mv[w]getch	Equivalent functionality.
mv[w]getstr	Equivalent functionality.
mv[w]inch	Equivalent functionality.
mv[w]insch	Equivalent functionality.
mv[w]insstr	VAX C specific.
mvwin	Equivalent functionality.
newwin	Equivalent functionality.
nice	Not equivalent. On VMS systems, the resulting priority cannot be greater than the process base priority.
[no]nl	Provided without functionality.
nlist	Not provided. You can obtain this information from the linker load map.
open	Not equivalent. VAX C requires mode = 2 when randomly writing to files.
openpl	Not provided.
overlay	Equivalent functionality.
overwrite	Equivalent functionality.
pause	Not equivalent. On VMS systems, processes can also be awakened with the SYS$WAKE system service.
pclose	Not provided.
perror	Equivalent functionality.
pipe	Not equivalent. VAX C specifies optional arguments for buffer size and asynchronous read operations.
point	Not provided.
popen	Not provided.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
pow	Equivalent functionality.
printf	Equivalent functionality.
[w]printw	Equivalent functionality.
profil	Not provided.
ptrace	Not provided.
putc	Equivalent functionality.
putchar	Equivalent functionality.
puts	Equivalent functionality.
putw	Equivalent functionality.
qsort	Equivalent functionality.
raise	Defined in the Draft Proposed ANSI C Standard (equivalent to the **gsignal** function).
rand	Equivalent functionality.
[no]raw	Equivalent functionality.
read	Equivalent functionality.
realloc	Not equivalent. On VAX C, you can reallocate only the last freed area. For example, if you make two calls to **free**, only the second area is reallocated.
reboot	Not provided.
[w]refresh	Equivalent functionality.
remove	Defined in the Draft Proposed ANSI C Standard (equivalent to the **delete** function).
rename	Equivalent functionality.
rewind	Equivalent functionality.
re_comp	Not provided.
re_exec	Not provided.
rindex	Not provided.
rint	Not provided.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
sbrk	Not equivalent. The VAX C version rounds the break address to the next higher multiple of 512 bytes.
scanf	Not equivalent. VAX C provides the following conversion characters: hd, ho, hx, ld, lo, lx, le, lf, i, n, and p.
[w]scanw	Equivalent functionality.
scroll	Equivalent functionality.
scrollok	Equivalent functionality.
[w]setattr	VAX C specific.
setbuf	Defined by the Draft Proposed ANSI C Standard.
setgid	Provided without functionality.
setgrent	Not provided.
setjmp	Equivalent functionality.
setpgrp	Not provided.
setpwent	Not provided.
setsfent	Not provided.
setuid	Provided without functionality.
setvbuf	Not equivalent.
sigblock	Equivalent functionality.
sighold	Not provided. See the VAX C **ssignal** and **gsignal** functions.
sigignore	Not provided. See the VAX C **ssignal** and **gsignal** functions.
signal	Equivalent functionality.
sigpause	Equivalent functionality.
sigsetmask	Equivalent functionality.
sigstack	Equivalent functionality.
sigvec	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
sigrelse	Not provided. See the VAX C **ssignal** and **gsignal** functions.
sigset	Not provided. See the VAX C **ssignal** and **gsignal** functions.
sigsys	Not provided. See the VAX C **ssignal** and **gsignal** functions.
sin	Equivalent functionality.
sinh	Equivalent functionality.
sleep	Equivalent functionality.
space	Not provided.
sprintf	Equivalent functionality. VAX C also provides the conversion characters n and p. See the **fprintf** and **printf** functions for more information.
sqrt	Equivalent functionality.
srand	Equivalent functionality.
sscanf	Not equivalent. VAX C provides the following conversion characters: h, ho, hx, ld, lo, lx, le, and lf.
ssignal	VAX C specific.
[w]standend	Equivalent functionality.
[w]standout	Equivalent functionality.
stat	Equivalent functionality.
stime	Not provided.
strcat	Equivalent functionality.
strchr	Equivalent functionality.
strcmp	Equivalent functionality.
strcpy	Equivalent functionality.
strcspn	Equivalent functionality.
strerror	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
strlen	Equivalent functionality.
strncat	Equivalent functionality.
strncmp	Equivalent functionality.
strncpy	Equivalent functionality.
strpbrk	Equivalent functionality.
strrchr	Equivalent functionality.
strspn	Equivalent functionality.
strstr	Equivalent functionality.
strtod	Equivalent functionality.
strtok	Equivalent functionality.
strtol	Equivalent functionality.
strtoul	Equivalent functionality.
subwin	Equivalent functionality.
swab	Not provided.
sync	Not provided.
syscall	Not provided.
system	Equivalent functionality.
tan	Equivalent functionality.
tanh	Equivalent functionality.
tgetent	Not provided.
tgetflag	Not provided.
tgetnum	Not provided.
tgetstr	Not provided.
tgoto	Not provided.
time	Not equivalent. VAX C does not return timezone or daylight fields.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
times	Not equivalent. VMS systems do not distinguish between system and user times. VAX C returns the time in 10-millisecond units.
timezone	Not provided.
tmpfile	Equivalent functionality.
tmpnam	Equivalent functionality.
toascii	Equivalent functionality.
tolower	Equivalent functionality.
touchwin	Equivalent functionality.
toupper	Equivalent functionality.
tputs	Not provided.
ttyname	Not equivalent. VAX C returns a pointer to the null-terminated path name of the terminal device associated with file descriptor 0 (standard input, stdin).
umask	Not equivalent. The default values of the **umask** function are set from RMS default file protection.
umount	Not provided.
ungetc	Equivalent functionality.
unlink	Not provided. This functionality is not provided in the VMS environment. You can create temporary files using the RMS extensions to **creat**. (See the **delete** and **remove** functions.)
vadvise	Not provided.
valloc	Not provided.
va_arg	Equivalent functionality.
va_count	VAX C specific.
va_end	Equivalent functionality.
va_start	Equivalent functionality.

(continued on next page)

Table A–1 (Cont.): Relationship of VAX C RTL Functions and Macros to Other C RTL Functions and Macros

C Function	VAX C Implementation
va_start_1	VAX C specific.
vfork	VAX C specific. This function is equivalent to the **fork** function in other implementations of the C language.
vfprintf	Equivalent functionality.
vhangup	Not provided.
vlimit	Not provided.
vprintf	Equivalent functionality.
vread	Not provided.
vsprintf	Equivalent functionality.
vswapon	Not provided.
vwrite	Not provided.
wait	Equivalent functionality.
wait3	Not provided.
wrapok	Provided without functionality.
write	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 procedures 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.
C$ATAN2	Arc tangent math function.

(continued on next page)

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
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.
C$GETPID	Get the process identification.

(continued on next page)

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
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 and 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$PRINTW	Curses **printf** function for window.
C$QSORT	Rapid sort function.
C$RAND	Random-number generator.
C$RMS_PROTOTYPES	Definition of RMS data structures.

(continued on next page)

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
C$SCANW	Curses **scanf** for window.
C$SCROLL	Curses scroll window function.
C$SETGID	Set group identification.
C$SETJMP	Nonlocal goto functions (**setjmp and longjmp**).
C$SETUID	Set user identification.
C$SIGNAL	Manipulate signal data base.
C$SIGVEC	Signal function.
C$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$STRSTR	Search for a string in a string.
C$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 or convert real-time values.
C$TMPFILE	Create a temporary file.

(continued on next page)

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 string 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.

(continued on next page)

Table B–1 (Cont.): VAX C Run-Time Modules

Module	Description
C$WREFRESH	Curses refresh window function.
C$WSCANW	Curses **scanf** function 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$MATCH_WILD	Expand file-name wildcards.
VAXC$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.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
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.
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 startup 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.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
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$RMS_PROTOTYPES	Indexed file key extended attribute block prototype.
cc$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$RMS_PROTOTYPES	Terminal characteristics of the extended attribute block.
ceil	C$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.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
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$DIVIDE	Compute quotient and remainder.
dup	C$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.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
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.
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.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
getname	C$VAXCIO	Get a file-name string.
getpid	C$GETPID	Get the process identification.
getppid	C$GETPPID	Get the parent process ID of the calling process.
gets	C$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$INITSCR	Begin Curses session.
isapipe	C$VAXCIO	Check for a mailbox.
isatty	C$VAXCIO	Check for a terminal file.
insstr	C$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$DIVIDE	Compute long integer quotient and remainder.
localtime	C$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 a terminal name.
lseek	C$VAXCIO	Seek to a position in a file.
malloc	C$MALLOC	Allocate memory.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
memchr	C$MEMFUNC	Locate first occurrence of a character.
memcmp	C$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.

(continued on next page)

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$CLI_NAME	Determine user's command-language interpreter.
shell$fix_time	SHELL$FIX_TIME	Translate time to a UNIX format.
shell$from_vms	SHELL$FROM_VMS	Translate VMS file specifications to DEC/Shell specifications.
shell$get_argv	SHELL$CLINT	Interface to argument lists under the Shell.
shell$is_shell	SHELL$CLI_NAME	Determine CLI name.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
shell$match_wild	SHELL$MATCH_ WILD	Wildcard expansion to infinite names.
shell$to_vms	SHELL$TO_VMS	Translate DEC/Shell file specifications to VMS specifications.
shell$translate_ vms	SHELL$TO_VMS	Translate DEC/Shell file specifications to DEC/Shell specifications.
sigblock	C$SIGVEC	Block signals from delivery.
signal	C$SIGNAL	Set a signal.
sigpause	C$SIGVEC	Pause and wait for a signal.
sigsetmask	C$SIGVEC	Block signals from delivery.
sigstack	C$SIGVEC	Define alternate signal stack.
sigvec	C$SIGVEC	Assign a handler function for a specific 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.
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 a string for a character.
strerror	C$PERROR	Translate an error message code.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
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$STRSPN	Search a string for a character.
strstr	C$STRSTR	Search a string in a string.
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$SUBWIN	Create a subwindow.
system	C$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.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
umask	C$VAXCIO	Set a file's protection mask.
ungetc	C$VAXCIO	Push a character back into the stream.
utime	C$VAXCIO	Set the access and modification times for a file.
vaxc$crtl_init	C$$MAIN	Initialize VAX 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 a **sigstack** function.
va_arg	VAXC$VARARGS	Return the next argument.
va_count	VAXC$VARARGS	Count the number of arguments.
va_end	VAXC$VARARGS	Terminate the processing of variable argument lists.
va_start	VAXC$VARARGS	Initialize to the beginning of an argument list.
va_start_1	VAXC$VARARGS	Initialize to the beginning of an argument list.
vfork	C$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$WCLEAR	Erase a window.
wclrattr	C$WCLRATTR	Turn off a screen attribute.
wclrtobot	C$CLRTOBOT	Erase a window to the bottom.
wclrtoeol	C$CWCLRTOEOL	Erase a window to the end of the current line.

(continued on next page)

Table B–2 (Cont.): VAX C Run-Time Entry Points

Entry Point	Module	Description
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$WINSCH	Insert a character on a window.
winsertln	C$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.

(continued on next page)

Table B-3 (Cont.): Run-Time Library Procedures Called by VAX C

Procedure	Description
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.

You can examine the contents of these modules in the appropriate definition file. All definition files have the file extension .H and 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.
acedef	Access control list entry structure definitions.
acldef	Access control list definitions.
acrdef	Accounting record definitions.
argdef	Argument descriptors definitions.
armdef	Access rights definitions.
assert	Assert macro definition.
atrdef	File attribute definitions.
basdef	Message definitions for BASIC.

(continued on next page)

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
brkdef	Breakthrough system service definitions.
chfdef	Structure definitions for condition handlers.
chkpntdef	Flags for calls to create processes with check points.
chpdef	Definitions for the $CHKPRO (check protection) service.
clidef	Command-language interface definitions.
climsgdef	Command-language interpreter error code definitions.
cliservdef	CLI service request codes.
cliverbdef	CLI generic codes for verbs.
clsdef	Security classification mask block definitions.
cobdef	Message definitions for COBOL.
cqualdef	Qualifier definitions.
crdef	Card reader status bits definitions.
credef	Create options table definitions.
crfdef	CRF$INSRTREF argument list definitions.
crfmsg	Return status codes for cross-reference program.
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.
dibdef	Device information block definitions.
dmpdef	Layout of the header block of the system dump file.
dmtdef	$DISMOU (dismount) system service definitions.
dstdef	Debug Symbol Table definitions.
dtk$routines	DECtalk routine definitions.
dtkdef	Definitions for RTL DECtalk Management.
dtkmsg	Message definitions for DECtalk.
dvidef	$GETDVI system service request code definitions.

(continued on next page)

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
envdef	Define/reference environment definitions.
eomdef	End-of-module record (EOM) definition.
eomwdef	End-of-module record with word of psect (EOMW) definition.
epmdef	GSD entry - Entry point definition, normal symbols.
epmmdef	GSD entry - Entry point definition, version mask symbols.
epmvdef	GSD entry - Entry point definition, vectored symbols.
epmwdef	GSD entry - Entry point definition with word of psect value.
eradef	Erase type codes definitions.
errno	Error number definitions.
errnodef	VAX C error message constants.
fab	File access block definitions.
faldef	Message definitions for the FAL (DECnet File Access Listener).
fchdef	File characteristics definitions.
fdldef	FDL call interface definitions.
fibdef	File information block definitions.
fiddef	FID (File ID) structure definitions.
file	Symbol definitions for the **open** function.
float	Macro definitions that provide implementation-specific, floating-point restrictions.
fmldef	Formal arguments structure definitions.
fordef	Message definitions for FORTRAN.
fscndef	SYS$FILESCAN descriptor codes.
gpsdef	GSD entry - Psect definition.
gsdef	Global symbol definition record (GSD) definitions.
gsydef	GSD entry - Symbol definition.
hlpdef	Definitions for help processing.
iacdef	Image activation control flags definitions.
idcdef	Random entity ident consistency check definitions.
iodef	I/O function code definitions.

(continued on next page)

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
jbcmsgdef	Message definitions for Job Controller.
jpidef	$GETJPI system service request code definitions.
kgbdef	Key Grant Block definitions for rights database.
ladef	LPA-11 characteristics definitions.
latdef	Message definitions for the LAT facility.
lbrctltbl	Library control table use by Librarian.
lbrdef	Librarian argument definitions.
lckdef	Lock manager definitions.
lepmdef	GSD entry - Module local entry point definition.
lhidef	Library header information array offsets.
lib$routines	Library (LIB$) routine definitions.
libclidef	Definitions for LIB$ CLI callback procedures.
libdcfdef	Definitions for LIB$DECODE_FAULT.
libdef	Definitions of LIB$ return codes.
libdtdef	Interface definitions for LIB$DT (date/time) package.
libvmdef	Interface definitions for LIB$VM package.
limits	Macro definitions that provide implementation-specific constraints.
lkidef	Lock information data identifier information.
lmfdef	License Management Facility definitions.
lnkdef	Linker Options Record (LNK).
lnmdef	Logical name flag definitions.
lpdef	Line printer characteristics definitions.
lprodef	GSD entry - Module Local Procedure definition.
lsdfdef	Module-local Symbol definition.
lsrfdef	Module-local Symbol reference.
lsydef	LSY - Module-Local symbol definition.
maildef	Definitions needed for mail that can be called.
math	Math function definitions.

(continued on next page)

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
mhddef	Object module header definitions.
mhdef	Module header record (MHD).
mntdef	Flag bits for the $MOUNT system service.
msgdef	System mailbox message type definitions.
mt2def	Extended magtape characteristics definitions.
mtadef	Magtape accessibility routine code definitions.
mtdef	Magtape status definitions.
mthdef	Message definitions for the math library.
nam	Name block definitions.
ncs$routines	Definitions for routines for working with national character sets.
ncsdef	Message definitions for the NCS facility.
nfbdef	DECnet file access definitions.
nsarecdef	Security Auditing record definitions.
objrecdef	Object file record definitions.
opcdef	OPCOM request code definitions.
opdef	Instruction opcode definitions.
oprdef	Operator communications message types and values.
ots$routines	Common object library routine definitions.
otsdef	Message definitions for common object library.
pccdef	Printer/terminal carriage-control specifiers.
perror	PERROR function-related definitions.
plvdef	Privileged library vector definition.
ppl$def	Definitions for RTL Parallel Processing Facility.
ppl$routines	Routine definitions for the Parallel Processing Facility.
ppldef	Message definitions for the Parallel Processing Facility.
pqldef	Process quota code definitions.
prcdef	Create process (SYS$CREPRC) system service status flags.
prdef	Processor register definitions.
processes	Prototype definitions for subprocess functions.

(continued on next page)

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
prodef	GSD entry - Procedure definition, normal symbols.
promdef	GSD entry - Procedure definition, version mask symbols.
provdef	GSD entry - Procedure definition, vectored symbols.
prowdef	GSD entry - Procedure definition with word of psect value.
prtdef	Protection field definitions.
prvdef	Privilege mask bit definitions.
psldef	Processor status longword definitions.
psmmsgdef	Message definitions for print symbiont.
pswdef	Processor Status Word definitions.
quidef	Get Queue Information Service ($GETQUI) definitions.
rab	Record access block definitions.
rmedef	RMS escape definitions.
rms	All RMS structures and return status value definitions.
rmsdef	RMS return status value definitions.
sbkdef	Statistics block definitions.
scrdef	Screen package request types.
sdfdef	Object symbol definitions.
sdfmdef	Object symbol definition for version mask symbols.
sdfvdef	Object symbol definition for vectored symbols.
sdfwdef	Object symbol definition with word of psect value.
secdef	Image section flag bit and match constant definitions.
setjmp	State buffer definition for the **setjmp** and **longjmp** functions.
sfdef	Stack call frame definitions.
sgpsdef	GSD entry - Psect definition in shareable image.
shrdef	Definition file for shared messages.
signal	Signal value definitions.
sjcdef	Send to Job Controller Service ($SNDJBC) definitions.
smg$routines	Screen Management Facility routine definitions.
smgdef	Curses Screen Management interface definitions.

(continued on next page)

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
smgmsg	Message definitions for Screen Management Facility.
smgtrmptr	Terminal Capability Pointers for RTL SMG$ facility.
smrdef	Symbiont manager request codes definitions.
sor$routines	Sort/Merge routine definitions.
sordef	Message definitions for Sort/merge.
srfdef	Object symbol reference.
srmdef	Hardware symbol definitions.
ssdef	System service return status value definitions.
starlet	System routine definitions.
stat	STAT and FSTAT function-related definitions.
stdarg	Variable argument list access definitions.
stddef	Common useful definitions.
stdio	Standard I/O definitions.
stdlib	Definitions of miscellaneous C functions.
str$routines	Routine definitions for dealing with strings.
strdef	Message definitions for VMS string functions.
string	C string function definitions.
stsdef	System service status code format definitions.
syidef	Definitions for the Get System-Wide Information (SYS$GETSYI) system service.
time	Definitions for the **localtime** function.
timeb	Definitions for the **ftime** function.
tirdef	Object file text, information and relocation record (TIR).
tpadef	TPARSE control block definitions.
trmdef	Define symbols for the item list QIO format.
tt2def	Terminal definitions.
ttdef	Terminal definitions.
types	Type definitions.
uaidef	Get User Authorization Information Data Identifier definitions.

(continued on next page)

Table C–1 (Cont.): VAX C Definition Modules

Module	Description
uicdef	Format of UIC (user identification code).
unixio	UNIX I/O functions.
unixio	UNIX I/O emulation functions.
unixlib	Miscellaneous UNIX emulation functions.
unixlib	UNIX emulation functions.
usgdef	Disk usage accounting file produced by ANALYZE/DISK_ STRUCTURE utility.
usridef	User image bit definitions.
varargs	Variable argument list access definitions.
xab	Extended attribute block definitions.
xwdef	System definitions for DECnet DDCMP.

Appendix D

VAX C Socket Routine Reference

D.1 Introduction

This appendix describes the aspects of the VAX C language that pertain to the writing of Internet application programs for the VMS/ULTRIX Connection product. For a description of Internet details, such as protocols, protocol types, and sockets, refer to the *VMS / ULTRIX Connection Programming Manual*. For more information on how to write socket programs, refer to the *ULTRIX Supplementary Documents, System Manager*.

D.2 Porting Considerations

This section contains information that you should consider when writing Internet application programs for the VMS/ULTRIX Connection. These considerations will help to make your programs more portable.

D.2.1 Calling an IPC Routine from an AST State

Calls to various IPC routines use a static area within which they return information. The VMS environment allows an AST routine to interrupt an IPC routine during its execution. In addition, the ASTs of more privileged modes can interrupt ASTs of less privileged modes. Therefore, caution needs to be observed when calling an IPC routine from AST state , while a similar IPC routine is being called from non-AST state or a less privileged mode.

The IPC routines that use a static area are:

- GETHOSTBYADDR
- GETHOSTBYNAME

- GETNETBYADDR
- GETNETBYNAME

In VMS Version 5.2, sockets should not be created or destroyed within ASTs.

D.2.2 Calling from KERNEL or EXEC Modes

Several IPC routines access files in order to retrieve their information. These routines should not be called from either the KERNEL or EXEC modes when ASTs are disabled. These IPC routines are:

- GETHOSTBYADDR
- GETHOSTBYNAME
- GETNETBYADDR
- GETNETBYNAME

D.2.3 Standard I/O

You cannot use Standard I/O with sockets; the fdopen function does not support sockets.

D.2.4 Event Flags

IPC routines may use event flags during their operation. The event flags are assigned by using the library routine LIB$GET_EF and are released when the routine no longer needs them.

D.2.5 Suppressing VAX C Compilation Warnings

Certain parameters to the IPC routines may require type casting to suppress VAX C compilation warnings. Type casting is required because of parameter prototyping, which the VAX C header (.h) files have in order to be ANSI compliant. These header files are unlike ULTRIX header files, whose IPC routines are not parameter prototyped.

D.2.6 Header Files

It is acceptable to specify in header files on a VMS system without angle brackets (< >) or double quotes (" "). For example, **# include types** would be acceptable. This is possible on the VMS system because all the header files are located in a text library in SYS$LIBRARY. In contrast, on an ULTRIX system the header files must be specified with angle brackets (< >) or double quotes (" ") and any subdirectories that are needed to locate a header file. For example, to specify the header file **types.h** , you would specify it by **#include <sys/types.h>**.

D.3 Linking an Internet Application Program

You link Internet application programs with the LINK command. For example:

```
$ LINK/MAP/FULL MAIN, SYS$LIBRARY:UCX$IPC/LIB, SYS$INPUT/OPTIONS
SYS$SHARE:VAXCRTL.EXE/SHARE
```

Use the OPTIONS qualifier for executable images. UCX$IPC.OLD contains the transfer vectors used to resolve the socket routine references to the VAXCRTL.

D.4 VAX C Structures

This section describes the structures used in writing Internet applications for the VMS/ULTRIX Connection product.

D.4.1 hostent Structure

The **hostent** structure, defined in the netdb.h header file, is used to specify or obtain a host name, a list of aliases associated with the network, and the network's number as specified in an Internet address from the host database. An entry in the host database is created with the command : **UCX> SET HOST xxxx**. See the *System Manager's Guide to VMS / ULTRIX Connection* for a description of the host database.

```
struct  hostent {
        char    *h_name;❶          /* official name of host */
        char    **h_aliases;❷      /* alias list */
        int     h_addrtype;❸       /* host address type */
        int     h_length;❹         /* length of address */
        char    **h_addr_list;❺    /* list of addresses from name server */
#define h_addr  h_addr_list[0]❻    /* address, for backward compatibility */
};
```

The members of the **hostent** structure are:

❶ **h_name** is a pointer to a NULL-terminated character string that is the official name of the host.

❷ **h_aliases** is a NULL-terminated array of alternate names for the host.

❸ **h_addrtype** is the type of address being returned; currently always AF_INET.

❹ **h_length** is the length, in bytes, of the address.

❺ **h_addr_list** is a pointer to a list of pointers to the network addresses for the host. Each host address is represented by a series of bytes in network order. They are not ASCII strings.

❻ **h_addr** is defined as the first address in the h_addr_list. This is used for backward compatibility.

D.4.2 in_addr Structure

The **in_addr** structure, defined in the in.h header file, is used to specify or obtain an Internet address. The address format can be any of the supported Internet address notations. Refer to the *VMS/ULTRIX Connection Programming Manual* for information on the Internet address notations.

```
struct in_addr {
       union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;
}      S_un;
#define s_addr  S_un.S_addr     /* can be used for most tcp & ip code */
#define s_host  S_un.S_un_b.s_b2        /* host on imp */
#define s_net   S_un.S_un_b.s_b1        /* network */
#define s_imp   S_un.S_un_w.s_w2        /* imp */
#define s_impno S_un.S_un_b.s_b4        /* imp # */
#define s_lh    S_un.S_un_b.s_b3        /* logical host */
};
```

D.4.3 iovec Structure

In ULTRIX, the **iovec** structure is defined in the UIO.H header file; in VMS it is defined in the socket.h header file.

The **iovec** structure describes one scatter/gather buffer. Multiple scatter/gather buffer descriptors are stored as an array of **iovec** elements.

```
struct iovec {
       char *iov_base;❶
       int  iov_len;❷
}
```

❶ **iov_base** field is a pointer to a buffer.

❷ **iov_len** field contains the size of the buffer to which **iov_base** points.

D.4.4 linger Structure

The linger structure, defined in the socket.h header file, specifies the setting or resetting of the **socket opt** for the time interval that the socket will **linger** for data. **linger** is supported only by STREAM type sockets.

```
struct  linger {
        int     l_onoff;❶        /* option on/off */
        int     l_linger;❷       /* linger time */
};
```

❶ **l_onoff** is a value of 1 sets the **linger**, while a value of 0 resets the **linger**.

❷ **l_linger** is the number of seconds to **linger** (default 120 seconds).

D.4.5 msghdr Structure

The **msghdr** structure, defined in the socket.h header file, is used to specify the buffer parameter of **recvmsg** and **sendmsg**. It allows specifying an array of scatter/gather buffers. **recvmsg** scatters the data to several user receive buffers. **msghdr** gathers data from several user transmit buffers before being transmitted.

```
struct msghdr {
        char      *msg_name;❶          /* optional address */
        int       msg_namelen;❷        /* size of address */
        struct    iov *msg_iov;❸       /* scatter/gather array */
        int       msg_iovlen;❹         /* # elements in msg_iov */
        char      *msg_accrights;❺     /* access rights sent/received */
        int       msg_accrightslen;❻
};
```

The members of the **msghdr** structure are:

❶ **msg_name** is the address of the destination socket if the socket is unconnected. If no address is required, this field may be set to NULL.

❷ **msg_namelen** is the length of the message name field.

❸ **msg_iov** is an array of I/O buffer pointers of the **iovec** structure form. See Section D.4.3 for a description of the **iovec** structure.

❹ **msg_iovlen** is the number of buffers in the **msg_iov array**.

❺ **msg_accrights** points to a buffer containing access rights sent with the message.

❻ **msg_accrightslen** is the length of the **msg_accrights** buffer.

D.4.6 netent Structure

The **netent** structure, defined in the netdb.h header file, is used to specify or obtain a network name, a list of aliases associated with the network, and the network's number specified as an Internet address from the network database. An entry in the network database is created with the command : **UCX> SET NETWORK xxxx**. See the *System Manager's Guide to VMS / ULTRIX Connection* for a description of the network database.

```
struct  netent {
        char     *n_name;❶          /* official name of net */
        char     **n_aliases;❷      /* alias list */
        int      n_addrtype;❸       /* net address type */
        long     n_net;❹            /* net number */
};
```

The members of the **netent** structure are:

❶ **n_name** is the official name of the network.

❷ **n_aliases** is a NULL-terminated list of pointers to alternate names for the network.

❸ **n_addrtype** is the type of the network number returned. Currently always AF_INET.

❹ **n_net** is the network number. It is returned in host byte order.

D.4.7 sockaddr Structure

The **sockaddr** structure, defined in the socket.h header file, specifies a general address family.

```
struct sockaddr {
        u_short sa_family;❶  /* address family */
        char sa_data[14];❷   /* up to 14 bytes of direct address */
};
```

The members of this structure are:

❶ **sa_family** is the address family or domain in which the socket was created.

❷ **sa_data** is the data string of up to 14 bytes of direct address.

D.4.8 sockaddr_in Structure

sockaddr_in structure, defined in the in.h header file, specifies an Internet address family.

```
struct sockaddr_in {
        short   sin_family;❶        /* address family */
        u_short sin_port;❷          /* port number */
        struct  in_addr sin_addr;❸ /* Internet address */
        char    sin_zero[8];❹       /* 8-byte field of all zeroes */
};
```

The members of sockaddr_in structure are:

❶ **sin_family** is the address family (Internet domain (AF_INET)).

❷ **sin_port** is the port number in network order.

❸ **sin_addr** is the Internet address in network order.

❹ **sin_zero** is an 8-byte field containing all zeroes.

D.4.9 timeval Structure

The **timeval** structure, defined in the socket.h header file, is used to specify times.

```
struct timeval {
    long tv_sec;❶
    long tv_usec;❷
};
```

❶ **tv_sec** field specifies the number of seconds to wait.

❷ **tv_usec** specifies the number of microseconds to wait.

D.5 Internet Protocols

The Internet protocol family is a collection of protocols layered on the Internet Protocol (IP) transport layer, and using the Internet address format. This section describes the Transmission Control Protocol and User Datagram Protocol.

D.5.1 Transmission Control Protocol

The Transmission Control Protocol (TCP) provides a reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the **SOCK_STREAM** abstraction. TCP uses the standard Internet address format and, in addition, provides a per host collection of **port addresses**. Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets utilizing the TCP protocol are either **active** or **passive**. Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the **listen** system call must be used after binding the socket with the **bind** system call. Only passive sockets may use the **accept** call to accept incoming connections. Only active sockets may use the **connect** call to initiate connections.

Passive sockets may **underspecify** their location to match incoming connection requests from multiple networks. This technique, called **wildcard addressing**, allows a single server to provide service to clients on multiple networks. To create a socket that listens to all hosts on any network, the Internet address **INADDR_ANY** must be bound. The TCP port must be specified at this time. If the Internet address is not INADDR and the port is not specified, the system will assign a port. Once a connection has been established, the socket's address is fixed by the peer entity's location. The

address assigned to the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity's network.

TCP supports one socket option that is set with **setsockopt** and tested with **getsockopt**. Under most circumstances, TCP sends data when it is presented; when outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acnowledgement is received. For a small number of clients, such as window systems that send a stream of mouse events that receive no replies, this packetization may cause significant delays. Therefore, TCP provides a Boolean option, **TCP_NODELAY** (from <netinet/tcp.h>), to defeat this algorithm. The option level for the **setsockopt** call is the protocol number for TCP, available from **getprotobyname**.

D.5.2 User Datagram Protocol

User Datagram Protocol (UDP) is a simple, unreliable datagram protocol used to support the **SOCK_DGRAM** abstraction for the Internet protocol family. UDP sockets are connectionless and are normally used with the **sendto** and **recvfrom** calls, though the **connect** call may also be used to fix the destination for future packets (in which case the **recv** or **read** or **write** system calls may be used).

UDP address formats are identical to those used by TCP. In particular, UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (for example, a UDP port may not be **connected** to a TCP port). Also, broadcast packets may be sent (assuming the underlying network supports this) by using a reserved **broadcast address**; this address is network interface dependent. The **SO_BROADCAST** option must be set on the socket and the process must have the SYSPRV or BYPASS privilege for broadcasting to succeed.

D.6 errno Values

errno is an external variable whose value is set whenever an error occurs during a call to any of the VAX C RTL routines. This value can be used to obtain a more detailed description of the error. **errno** is not cleared on successful calls, so its value should be checked only when an error has been indicated.

Most calls to the VAX C RTL routines have one or more returned values. Any error condition is indicated by an otherwise impossible return value. This is almost always -1; the individual routine descriptions specify the details.

All return codes and values from routines are of type integer unless otherwise noted. An error number is also made available in the external variable **errno**, which is not cleared on successful calls. The **errno** values may be translated to a message, similar to that found in UNIX systems, by using the **perror** function. **vaxc$errno** may also be returned as an error.

NOTE

The notation **[...]** is used in this manual to denote an **errno** error.

Table D–1 lists the **errno** values.

Table D–1: errno Values

EINPROGRESS	Operation now in progresss An operation that takes a long time to complete, such as **connect**, was attempted on a non-blocking object.
EALREADY	Operation already in progress An operation was attempted on a non-blocking object that already had an operation in progress.
ENOTSOCK	Socket operation on non-socket
EDESTADDRREQ	Destination address required A required address was omitted from an operation on a socket.
EMSGSIZE	Message too long A message sent on a socket was larger than the internal message buffer.

(continued on next page)

Table D–1 (Cont.): errno Values

EPROTOTYPE	Protocol wrong type for socket A protocol was specified that does not support the semantics of the socket type requested. For example you cannot use the ARPA Internet UDP protocol with type **SOCK_STREAM**.
ENOPROTOOPT	Protocol not available A bad option was specified in a **getsockopt** or **setsocketopt** call.
EPROTONOSUPPORT	Protocol not supported The protocol has not been configured into the system or no implementation for it exists.
ESOCKTNOSUPPORT	Socket type not supported The support for the socket type has not been configured into the system or no implementation for it exists.
EOPNOTSUPP	Error-operation not supported For example, tyring to accept a connection on a datagram socket.
EPFNOSUPPORT	Protocol family not supported The protocol family has not been configured into the system or no implementation for it exists.
EAFNOSUPPORT	Address family not supported by protocol family An address incompatible with the requested protocol was used.
EADDRINUSE	Address already in use Each address can be used only once.
EADDRNOTAVAIL	Cannot assign requested address Normally, results from an attempt to create a socket with an address not on this machine.

(continued on next page)

Table D–1 (Cont.): errno Values

ENETDOWN	Network is down A socket operation encountered a dead network.
ENETUNREACH	Network is unreachable A socket operation was attempted to an unreachable network.
ENETRESET	Network dropped connection on reset The host you were connected to crashed and rebooted.
ECONNABORTED	Software caused connection abort A connection abort was caused internal to your host machine.
ECONNRESET	Connection reset by peer A connection was forcibly closed by a peer. This usually results from the peer executing a **shutdown** call.
ENOBUFS	No buffer space available An operation on a socket or pipe was not performed because the system lacked sufficient buffer space.
EISCONN	Socket is already connected A **connect** request was made on an already connected socket; or, a **sendto** or **sendmsg** request on a connected socket specified a destination other than the connected party.
ENTOTCONN	Socket is not connected Request to send or receive data was disallowed because the socket is not connected.
ESHUTDOWN	Cannot send after socket shutdown A request to send data was disallowed because the socket had already been shut down with a previous **shutdown** call.
ETOOMANYREFS	Too many references: cannot splice

(continued on next page)

Table D–1 (Cont.): errno Values

ETIMEDOUT	Connection timed out A **connect** request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.) A **connect** request or remote file operation failed because the connected party did not properly respond after a period of time that is dependent on the communication protocol.
ECONNREFUSED	Connection refused No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.
ELOOP	Too many levels of symbolic links A path name lookup involved more than eight symbolic links.
ENAMETOOLONG	File name too long A component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.
EHOSTDOWN	Host is down A socket operation failed because the destination host was down.
EHOSTUNREACH	No route to host A socket operation was attempted to an unreachable host.
EVMSERR	VMS-specific error code that is non-translatable

D.7 Basic Communication Routines

This section contains the basic communication routines that make up the **building blocks** of Internet programs. These calls are listed in Table D–2.

Table D–2: Basic Communication Routines

Routine	Description
Accept	Accepts a connection on a socket.
Bind	Binds a name to a socket.
Close	Closes a connection and deletes a socket descriptor.
Listen	Set the maximum limit of outstanding connection requests for a socket.
Read	Reads bytes from a file or socket and places them into a buffer.
Readv	Not implemented.
Recv	Receives bytes from a socket and places them into a buffer.
Recvfrom	Receives bytes for a socket from any source.
Recvmsg	Receives bytes from a socket and places them into scattered buffers.
Select	Allows the polling or checking of a group of sockets.
Send	Sends bytes through a socket to a connected peer.
Sendmsg	Sends gathered bytes through a socket to any other socket.
Sendto	Sends bytes through a socket to any other socket.
Shutdown	Shuts down all or part of a bidirectional socket.
Socket	Creates an endpoint for communication by returning a socket descriptor.
Write	Writes bytes from a buffer to a file or socket.
Writev	Not implemented.

accept

Accepts a connection on a socket.

Format

#include *types*

#include *socket*

int accept *(***int** *s,* **struct sockaddr** **addr,* **int** **addrlen);*

Arguments

s
Is a socket descriptor that has been returned by **socket**, subsequently bound to an address with **bind**, and that is listening for connections after a **listen**.

addr
Is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the structure to which the address parameter points is determined by the domain in which the communication is occurring. This version of VAX C supports only the Internet domain (AF_INET).

addrlen
Is a value-result parameter; it should initially contain the size of the structure pointed to by **addr**. On return it will contain the actual length (in bytes) of the structure that has been filled in by the communication layer. See Section D.4.7 for a description of the **sockaddr** structure.

accept

Description

The **accept** routine completes the first connection on the queue of pending connections, creates a new socket with the same properties as **s** and allocates and returns a new descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as nonblocking, **accept** blocks the caller until a connection request is present. If the socket is marked nonblocking by using a **setsockopt** call and no pending connections are present on the queue, **accept** returns an error. The accepted socket may not be used to accept connections. The original socket **s** remains open (listening) for other connection requests. This call is used with connection-based socket types, currently with **SOCK_STREAM**.

It is possible to **select** a socket for the purposes of performing an **accept** by selecting it for read.

See also **bind**, **connect**, **listen**, **select**, and **socket**.

Return Values

-1	Indicates that the call failed and is further specified in the global **errno**.
x	A nonnegative integer that is a descriptor for the accepted socket.
[EBADF]	The socket descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[EOPNOTSUPP]	The reference socket is not of type **SOCK_STREAM**.
[EFAULT]	The **addr** parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked nonblocking and no connections are present to be accepted.

bind

Binds a name to a socket.

Format

#include *types*

#include *socket*

int bind (**int** *s*, **struct sockaddr** **name*, **int** *namelen);*

Arguments

s
Is a socket descriptor that has been created with **socket**.

name
Address of a structure used to assign a name to the socket in the format specific to the family (AF_INET) socket address. See Section D.4.7 for description of the **sockaddr** structure.

namelen
Is the size in bytes of the structure pointed to by **name**.

Description

The **bind** routine assigns a name to an unnamed socket. When a socket is created with **socket** it exists in a name space (address family) but has no name assigned. The **bind** routine requests that a name be assigned to the socket.

See also **connect**, **getsockname**, **listen**, and **socket**

Return Values

0	Indicates success.
-1	Indicates an error and is further specified in the global **errno**.
[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified Internet address and ports are already in use.
[EINVAL]	The socket is already bound to an address.
[EACCESS]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The name parameter is not a valid part of the user address space.

close

Closes a connection and deletes a socket descriptor.

Format

#include *unixio*

int close *(s);*

Argument

s
Is a socket descriptor.

Description

The **close** deletes a descriptor from the per-process object reference table. If this is the last reference to the underling object, then it will be deactivated.

See also **accept**, **socket**, and **write**.

Return Values

0	Indicates success.
-1	Indicates an error and is further specified in the global **errno**.
[EBADF]	The socket descriptor is invalid.

connect

Initiates a connection on a socket.

Format

#include *types*

#include *socket*

int connect *(***int** *s,* **struct sockaddr** **name,* **int** *namelen);*

Arguments

s
Is a socket descriptor that has been created with **socket**.

name
Is the address of a structure that specifies the name of the remote socket in the format specific to the address family (AF_INET).

namelen
Is the size in bytes of the structure pointed to by name.

Description

If **s** is a socket descriptor of type **SOCK_DGRAM**, then this call permanently specifies the peer to which data is to be sent. If it is of type **SOCK_STREAM**, then this call attempts to make a connection to another socket.

Each communications space interprets the name parameter in its own way. This argument specifies the socket to which the socket specified in **s** is to be connected.

See also **accept**, **select**, **socket**, **getsockname**, and **shutdown**.

Return Values

0	Indicates success.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EAFNOSUPPORT]	Address in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network is not reachable from this host.
[EADDRINUSE]	The specified Internet address and ports are already in use.
[EFAULT]	The name parameter is not a valid part of the user address space.
[EWOULDBLOCK]	The socket is nonblocking and the connection cannot be completed immediately. It is possible to **select** the socket while it is connecting by selecting it for writing.

listen

Sets the maximum limit of outstanding connection requests for a socket that is connection-oriented.

Format

int listen (**int** *s*, **int** *backlog*);

Arguments

s
Is a socket descriptor of type **SOCK_STREAM** that has been created using **socket.**

backlog
Specifies the maximum number of pending connections that may be queued on the socket at any given time. The maximum cannot exceed 5.

Description

This routine simply creates a queue for pending connection requests on socket **s** with a maximum size of **backlog**. Connections may then be accepted with **accept**.

If a connection request arrives with the queue full (more than backlog connection requests pending), the client will receive an error with an **errno** indication of **ECONNREFUSED**.

See also **accept**, **connect**, and **socket**

Return Values

0	Indicates success.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The socket descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation **listen**.

read

Reads bytes from a socket or file and places them in a buffer.

Format

#include *unixio*

int read *(***int** *d,* **void** **buffer,* **int** *nbytes);*

Arguments

d
Is a descriptor. The specified descriptor must refer to a socket or file currently opened for reading.

buffer
Is the address of contiguous storage in which the input data is placed.

nbytes
Is the maximum number of bytes involved in the read operation.

Description

If the end-of-file is not reached, the **read** routine returns **nbytes**. If the end-of-file occurs during the **read** routine, it returns the number of bytes read.

Upon successful completion, **read** returns the number of bytes actually read and placed in the buffer.

See also **socket**.

Return Values

x	Indicates end-of-file has been reached.
-1	Indicates an error and is further specified in the global **errno**.
[EBADF]	The descriptor is invalid.
[EFAULT]	The **buf** points outside the allocated address space.
[EINVAL]	The **nbytes** argument is negative.
[EWOULDBLOCK]	The **NBIO** socket option (nonblocking) flag is set for the socket or file descriptor and the process would be delayed in the read operation.

recv

Receives bytes from a connected socket and places them into a buffer.

Format

#include *types*

#include *socket*

int recv *(***int** *s,* **char** **buf,* **int** *len,* **int** *flags);*

Arguments

s
Is a socket descriptor that was created as the result of a call to **accept** or **connect**.

buf
Is a pointer to a buffer into which received data will be placed.

len
Specifies the size of the buffer pointed to by **buf**.

flags
Is a bit mask that may contain one or more of: **MSG_OOB** and **MSG_PEEK**. It is built by **or**ing the appropriate values together.

The **MSG_OOB** flag allows out-of-band data to be received. If out-of-band data is available, it will be read before any other data that is available. If no out-of-band data is available, the **MSG_OOB** flag is ignored. Out-of-band data can be sent using **send**, **sendmsg**, and **sendto**.

The **MSG_PEEK** flag allows you to **peek** at the data that is next in line to be received without actually removing it from the system's buffers.

Description

This routine receives data from a connected socket. To receive data on an unconnected socket, use the **recvfrom** or **recvmsg** routines. The received data is placed in the buffer **buf**.

Data is sent by the socket's peer using the **send**, **sendmsg**, or **sendto** routines.

The **select** call may be used to determine when more data arrives.

If no data is available at the socket, the receive call waits for data to arrive, unless the socket is nonblocking in which case a -1 is returned with the external variable **errno** set to **EWOULDBLOCK**.

See also **read**, **send**, **sendmsg**, **sendto**, and **socket**.

Return Values

x	The number of bytes received and placed in **buf**.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The socket descriptor is invalid.
[EINVAL]	The **nbytes** argument is negative.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EPIPE]	An attempt was made to write to a socket that is not open for reading by any process.
[EWOULDBLOCK]	The **NBIO** (non_blocking) flag is set for the socket descriptor and the process would be delayed in the write operation.
[EFAULT]	The data was specified to be received into non-existent or protected part of the process address space.

recvfrom

Receives bytes from a socket from any source.

Format

#include *types*

#include *socket*

int recvfrom (**int** *s,* **char** **buf,* **int** *len,* **int** *flags,* **struct sockaddr** **from,* **int** **fromlen)* ;

Arguments

s
Is a socket descriptor that has been created with **socket** and bound to a name using **bind** or as a result of **accept**.

buf
Is a pointer to a buffer into which received data will be placed.

len
Specifies the size of the buffer pointed to by **buf**.

flags
Is a bit mask that may contain one or more of: **MSG_OOB** and **MSG_PEEK**. It is built by **or**ing the appropriate values together.

The **MSG_OOB** flag allows out of band data to be received. If out-of-band data is available, it will be read before any other data that is available. If no out-of-band data is available, the **MSG_OOB** flag is ignored. Out-of-band data can be sent using **send**, **sendmsg**, and **sendto**.

The **MSG_PEEK** flag allows you to **peek** at the data that is next in line to be received without actually removing it from the system's buffers.

from
If **from** is nonzero, **from** is a buffer into which **recvfrom** places the address (structure) of the socket from which the data is received. If **from** was zero, the address will not be returned.

fromlen
Points to an integer containing the size of the buffer pointed to by **from**. On return, the integer is modified to contain the actual length of the socket address structure returned.

Description

This routine allows a named, unconnected socket to receive data. The data is placed in the buffer pointed to by **buf**, and the address of the sender of the data is placed in the buffer pointed to by **from** if **from** is non-NULL. The structure that **from** points to is assumed to be as large as the **sockaddr** structure. See Section D.4.7 for description of **sockaddr** structure.

To receive bytes from any source, the sockets need not be connected to another socket.

The **select** call may be used to determine if data is available.

If no data is available at the socket, the receive call waits for data to arrive, unless the socket is nonblocking in which case a -1 is returned with the external variable **errno** set to **EWOULDBLOCK**.

See also **read**, **send**, **sendmsg**, **sendto**, and **socket**.

Return Values

x	Is the number of bytes of data received and placed in **buf**.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The socket descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.

[EPIPE]	An attempt was made to write to a socket that is not open for reading by any process.
[EWOULDBLOCK]	The **NBIO** (non_blocking) flag is set for the socket descriptor and the process would be delayed in the write operation.
[EINVAL]	The **nbytes** argument is negative.
[EFAULT]	The data was specified to be received into non-existent or protected part of the process address space.

recvmsg

Receives bytes on a socket and places them into scattered buffers..

Format

#include *types*

#include *socket*

int recvmsg *(***int** *s,* **struct msghdr** *msg[],* **int** *flags);*

Arguments

s
Is a socket descriptor that has been created with **socket**.

msg
Specifies a **msghdr** structure. See Section D.4.5 for a description of the **msghdr** structure.

flags
Is a bit mask that may contain one or more of: **MSG_OOB** and **MSG_PEEK**. It is built by **or**ing the appropriate values together.

The **MSG_OOB** flag allows out-of-band data to be received. If out-of-band data is availiable, it will be read before any normal data that is available. If no out-of-band data is available, the **MSG_OOB** flag is ignored. Out-of-band data can be sent using **send**, **sendmsg**, and **sendto**.

The **MSG_PEEK** flag allows you to **peek** at the data that is next in line to be received without actually removing it from the system's buffers.

Description

This routine may be used with any socket, whether it is in a connected state or not. It receives data sent by a call to **sendmsg**, **send**, or **sendto**. The message is scattered into several user buffers if such buffers are specified.

To receive data, the socket need not be connected to another socket.

When the **iovec[iovcnt]** array specifies more than one buffer, the input data is scattered into **iovcnt** buffers as specified by the members of the **iovec array: iov[0], iov[1], ..., iov[iovcnt]**.

When a message is received, it is split among the buffers by filling the first buffer in the list, then the second, and so on, until either all of the buffers are full or there is no more data to be placed in the buffers.

When a message is sent, the first buffer is copied to a system buffer and then the second buffer is copied, followed by the third buffer and so on, until all the buffers are copied. After the data is copied, the protocol will send the data to the remote host at the appropirate time, depending upon the protocol.

The **select** call may be used to determine when more data arrives.

See also **read**, **send**, and **socket**.

Return Values

x	Number of bytes returned in the **msg_iov** buffers.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The socket descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EPIPE]	An attempt was made to write to a socket that is not open for reading by any process.

[EWOULDBLOCK]	The **NBIO** (non_blocking) flag is set for the socket descriptor and the process would be delayed in the write operation.
[EINVAL]	The **nbytes** argument is negative.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	The data was specified to be received into non-existent or protected part of the process address space.

select

Allows the user to poll or check a group of sockets for I/O activity. It can check what sockets are ready to be read or written, or what sockets have a pending exception.

Format

#include *time*

int select *(***int** *nfds,* **int** **readfds,* **int** **writefds,* **int** **execptfds,* **struct timeval** **timeout);*

Arguments

nfds
Specifies the highest numbered socket descriptor to search for. That is, it specifies the highest numbered bit +1 in **readfds**, **writefds**, and **exceptfds** that should be examined. Descriptor **s** is represented by **1<<s** (1 shifted to the left **s** number of times).

This argument is used only to improve efficiency. If you are unsure what the highest numbered socket descriptor is, **nfds** can safely be set to a number lower than 32.

The VAX C **select** routine only examines the longwords referenced by the **readfds**, **writefds**, and **exceptfds** arguments. Note that this means that no program that uses the VAX C **select** routine may ever have more than 32 files and sockets opened simultaneously.

readfds
Is a pointer to an array of bits, organized as integers (each integer describing 32 descriptors), that should be examined for read readiness. If bit **n** of the longword is set, socket descriptor **n** will be checked to see if it is ready to be read. All bits set in the bit mask must correspond to the file descriptors of sockets. The **select** routine cannot be used on normal files.

On return, the longword to which **readfds** points contains a bit mask of the sockets that are ready for reading. Only bits that were set on entry to **select** could possibly be set on exit.

writefds
Is a pointer to a longword bit mask of the socket descriptors that should be examined for write readiness. If bit **n** of the longword is set, socket descriptor n will be checked to see if it is ready to be written to. All bits set in the bit mask must correspond to socket descriptors.

On return, the longword that **writefds** points to contains a bit mask of the sockets that are ready for writing. Only bits that were set on entry to **select** will be set on exit.

exceptfds
Is a pointer to a longword bit mask of the socket descriptors that should be examined for exceptions. If bit **n** of the longword is set, socket descriptor **n** will be checked to see if it has any pending exceptions. All bits set in the bit mask must correspond to the file descriptors of sockets.

On return, the longword **exceptfds** pointer contains a bit mask of the sockets that have exceptions pending. Only bits that were set on entry to **select** could possibly be set on exit.

timeout
Specifies how long **select** should examine the sockets before returning. If one of the sockets specified in the **readfds**, **writefds**, and **exceptfds** bit masks is ready for I/O, **select** will return before the timeout period has expired.

The **timeout** structure points to a **timeval** structure. See Section D.4.9 for a description of the **timeval** structure.

Description

This routine determines the I/O status of the sockets specified in the various mask arguments. It returns either when a socket is ready to be read or written, or when the **timeout** period expires. If **timeout** is a nonzero integer, it specifies a maximum interval to wait for the selection to complete.

If the **timeout** argument is NULL, **select** will block indefinitely. In order to effect a poll, timeout should be non-NULL, and should point to a zero-valued structure.

If a process is blocked on a **select** while waiting for input from a socket and the sending process closes the socket, the **select** notes this as an event and will unblock the process. The descriptors are always modified on return if **select** returns because of the **timeout**.

NOTE

When the socket option SO_OOBINLINE is set on the device_ socket, a **select** on both read and exception events returns the socket mask set on both the read and exception mask. Otherwise, only the exception mask is set.

See also **accept**, **connect**, **read**, **recv**, **recvfrom**, **recvmsg**, **send**, **sendmsg**, **sendto**, and **write**.

Return Values

n	The number of sockets that were ready for I/O or that had pending exceptions. This value matches the number of returned bits that are set in all output masks.
0	Indicates that **select** timed out before any socket became ready for I/O.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	One of the bit masks specified an invalid descriptor.
[EINVAL]	The specified time limit is unacceptable. One of its components is negative or too large.

send

Sends bytes though a socket to its connected peer.

Format

#include *types*

#include *socket*

int send *(***int** *s,* **char** **msg,* **int** *len,* **int** *flags);*

Arguments

s
Is a socket descriptor that was created with **socket**, and that has been connected to another socket using **accept** or **connect**.

msg
Is a pointer to a buffer containing the data to be sent.

len
Specifies the length in bytes of the data pointed to by **msg**.

flags
May be either 0 or **MSG_OOB**. If it is equal to **MSG_OOB**, the data will be sent **out-of-band**. This means that the data can be received before other pending data on the receiving socket if the receiver also specifies a **MSG_OOB** in the flag parameter of the call.

Description

The **send** routine may only be used on connected sockets. To send data on an unconnected socket, use the **sendmsg** or **sendto** routines. The send routine simply passes data along to its connected peer, which may receive the data by using **recv**.

If there is no space available to buffer the data being sent on the receiving end of the connection, **send** will normally block until buffer space becomes available. If the socket is defined as nonblocking, however, **send** will fail with an **errno** indication of **EWOULDBLOCK**. If the message is too large to be sent in one piece and the socket type requires that messages be sent atomically **(SOCK_DGRAM)**, **send** will fail with an **errno** indication of **EMSGSIZE**.

No indication of failure to deliver is implicit in a **send**. All errors (except **EWOULDBLOCK**) are detected locally. The **select** routine may be used to determine when it is possible to send more data.

See also **read**, **recv**, **recvmsg**, **recvfrom**, **getsocketopt**, and **socket**.

Return Values

n	The number of bytes sent. This value will normally equal **len**.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The socket descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	Blocks if the system does not have enough space for buffering the user data.

sendmsg

Sends gathered bytes through a socket to any other socket.

Format

#include *types*

#include *socket*

int sendmsg *(***int** *s,* **struct msghdr** *msg[],* **int** *flags);*

Arguments

s
Is a socket descriptor that has been created with **socket**.

msg
Is a pointer to a **msghdr** structure containing the message to be sent. See Section D.4.5 for a description of the **msghdr** structure.

The **msg_iov** field of the **msghdr** structure is used as a series of buffers from which data is read in order until **msg_iovlen** bytes have been obtained.

flags
May be either 0 or **MSG_OOB**. If it is equal to **MSG_OOB**, the data will be sent **out-of-band**. This means that the data can be received before other pending data on the receiving socket if the receiver also specifies a flag of **MSG_OOB**.

sendmsg

Description

The **sendmsg** routine may be used on any socket to send data to any named socket. The data in the **msg_iovec** field of the **msg** structure is sent to the socket whose address is specified in the **msg_name** field of the structure. The receiving socket gets the data using either **read**, **recv**, or **recvfrom**, **recvmsg** routine. When the **iovec** array specifies more than one buffer, the data is gathered from all specified buffers before being sent. See Section D.4.3 for a description of the **iovec** structure.

If there is no space available to buffer the data being sent on the receiving end of the connection, **sendmsg** will normally block until buffer space becomes available. If the socket is defined as nonblocking, however, **sendmsg** will fail with an **errno** indication of **EWOULDBLOCK**. If the message is too large to be sent in one piece and the socket type requires that messages be sent atomically (SOCK_DGRAM), **sendmsg** will fail with an **errno** indication of **EMSGSIZE**.

If the address specified is a **INADDR_BROADCAST** address, the **SO_ BROADCAST** option must be set and the process must have SYSPRV or BYPASS privilege for the I/O operation to succeed.

No indication of failure to deliver is implicit in a **sendmsg**. All errors (except **EWOULDBLOCK**) are detected locally. The **select** routine may be used to determine when it is possible to send more data.

See also **read**, **recv**, **recvfrom**, **recvmsg**, **getsockopt**, and **socket**.

Return Values

n	The number of bytes sent.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.

[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	Blocks if the system does not have enough space for buffering the user data.

sendto

Sends bytes through a socket to any other socket.

Format

#include *types*

#include *socket*

int sendto (**int** *s,* **char** **msg,* **int** *len,* **int** *flags,* **struct sockaddr** **to,* **int** *tolen);*

Arguments

s
Is a socket descriptor that has been created with **socket**.

msg
Is a pointer to a buffer containing the data to be sent.

len
Specifies the length of the data pointed to by **msg**.

flags
May be either 0 or **MSG_OOB**. If it is equal to **MSG_OOB**, the data will be sent **out-of-band**. This means that the data can be received before other pending data on the receiving socket if the receiver also specifies a **MSG_OOB** in its flag parameter of the call.

to
Points to the address structure of the socket to which the data is to be sent.

tolen
Specifies the length of the address structure **to** points to.

Description

The **sendto** routine may be used on any socket to send data to any named socket. The data in the **msg** buffer is sent to the socket whose address is specified in **to**, and the address of socket **s** is provided to the receiving socket. The receiving socket gets the data using either **read**, **recv**, **recvfrom**, or **recvmsg** routine.

If there is no space available to buffer the data being sent on the receiving end of the connection, **sendto** will normally block until buffer space becomes available. If the socket is defined as nonblocking, however, **sendto** will fail with an **errno** indication of **EWOULDBLOCK**. If the message is too large to be sent in one piece and the socket type requires that messages be sent atomically (SOCK_DGRAM), **sendto** will fail with an errno indication of **EMSGSIZE**.

No indication of failure to deliver is implicit in a **sendto**. All errors (except **EWOULDBLOCK**) are detected locally. The **select** routine may be used to determine when it is possible to send more data.

If the address specified is a **INADDR_BROADCAST** address, **SO_BROADCAST** option must be set and the process must have SYSPRV or BYPASS privilege for the I/O operation to succeed.

See also **getsockopt**, **read**, **recv**, **recvfrom**, **recvmsg**, and **socket**.

Return Values

n	The number of bytes sent. This value will normally equal **len**.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.

[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	Blocks if the system does not have enough space for buffering the user data.

shutdown

Shuts down all or part of a bidirectional connection on a socket. It can disallow further receives, further sends, or both.

Format

#include *socket*

shutdown *(***int** *s,* **int** *how);*

Arguments

s
Is a socket descriptor that is in a connected state as a result of a previous call to either **connect** or **accept**.

how
Specifies how the socket is to be shut down. It may have any of the following values:

0	Further calls to **recv** on the socket are to be disallowed.
1	Further calls to **send** on the socket are to be disallowed.
2	Further calls to both **send** and **recv** are to be disallowed.

Description

This routine allows communications on a socket to be shut down one piece at a time rather than all at once. It can be used to create unidirectional connections rather than the normal bidirectional (full-duplex) connections.

See also **connect** and **socket**.

Return Values

0	Indicates success.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The socket descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[ENOTCONN]	The specified socket is not connected.

socket

Creates an endpoint for communication by returning a special kind of file descriptor called a socket descriptor, which is associated with a VMS/ULTRIX Connection socket device channel.

Format

#include *types*

#include *socket*

int socket (**int** *af,* **int** *type,* **int** *protocol);*

Arguments

af

Specifies the address format to be used in later references to the socket. Addresses specified in subsequent operations using the socket are interpreted according to this format. Currently, only AF_INET (Internet style) addresses are supported.

type

Specifies the semantics of communication. The type may be **SOCK_STREAM**, **SOCK_DGRAM**, or **SOCK_RAW**.

SOCK_STREAM type sockets provide sequenced, reliable, two-way connection based byte streams with an available out-of-band data transmission mechanism.

SOCK_DGRAM sockets support datagrams (connectionless, unreliable data transmission mechanism).

SOCK_RAW sockets provide access to internal network interfaces, and are available only to users with SYSPRV privilege.

protocol
Specifies the protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type using a given address format. However, it is possible that many protocols may exist, in which case a particular protocol must be specified with this argument. The protocol number to use is particular to the **communication domain** in which communication is to take place.

Description

This routine provides the primary mechanism for creating sockets. The type and protocol of the socket affect the way the socket behaves and how it can be used.

The operation of sockets is controlled by socket-level options, defined in the file socket.h. The calls **setsockoption** and **getsockoption** are used to set and get options. Options other than SO_LINGER take an integer parameter that should be nonzero if the option is to be enabled, or zero if it is to be disabled. SO_LINGER uses a **linger** structure parameter defined in socket.h. This structure specifies the desired state of the option and the linger interval in the following manner:

SO_REUSEADDR — allow local address reuse
SO_KEEPALIVE — keep connections alive
SO_DONTROUTE — do not apply routing on outgoing messages
SO_LINGER — linger on close if data present
SO_BROADCAST — permit sending of broadcast messages

SO_REUSEADDR indicates the rules used in validating addresses supplied in a **bind** call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified through the error code SS$_LINKDISCON.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the actions taken when unsent messages are queued on the socket and a **close** is performed. When using the **setsockopt** to set the linger values, the option value for the SO_LINGER command is the address of a linger structure:

```
struct  linger {
        int     l_onoff;        /* option on/off */
        int     l_linger;       /* linger time */
};
```

If the socket promises reliable delivery of data and l_onoff is nonzero, the system will block the process on the attempt until it is able to transmit the data or until it decides it is unable to deliver the information. A timeout period, called the linger interval, is specified in l_linger. If l_onoff is set to zero and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

SO_BROADCAST is used to enable or disable broadcasting on the socket.

See also **accept**, **bind**, **connect**, **getsockname**, **getsockopt**, **listen**, **read**, **recv**, **recvfrom**, **recvmgs**, **select**, **send**, **sendmsg**, **sendto**, **shutdown**, and **write**.

Return Values

x	Is a file descriptor that refers to the socket descriptor.
-1	Indicates an error and is further specified in the global **errno**.
[EAFNOSUPPORT]	The specified address family is not supported in this version of the system.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.
[EPROTONOSUPPORT]	The specified protocol is not supported.

[EPROTOTYPE]	Request for a type of socket for which there is no supporting protocol.
[EMFILE]	The per-process descriptor table is full.
[ENOBUFS]	No buffer space is available. The socket cannot be created.

write

Writes a buffer of data to a socket or file.

Format

#include *unixio*

int write *(***int** *d,* **void** **buffer,* **int** *nbytes);*

Arguments

d
Is a descriptor. The specified descriptor must refer to a socket or file.

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

nbytes
Is the maximum number of bytes involved in the write operation.

Description

The **write** call attempts to write a buffer of data to a socket or file.

See also **socket**.

Return Values

x	Number of bytes written to the socket or file.
0	Indicates an error.
-1	Indicates an error and is further specified in the global **errno**.

[EBADF]	The **d** argument is not a valid descriptor open for writing.
[EPIPE]	An attempt was made to write to a socket that is not open for reading by any process.
[EFAULT]	Part of the array pointed to by **iov** or data to be written to the file points outside the process's allocated address space.
[EWOULDBLOCK]	The **NBIO** (non_blocking) flag is set for the socket descriptor and the process would be delayed in the write operation.
[EINVAL]	The **nbytes** argument is negative.

D.8 Auxiliary Communication Routines

This section describes auxiliary communication routines. These routines are used to provide information about a socket and to set the options on a socket. See Table D–3 for a description of these routines.

Table D–3: Auxiliary Communication Routines

Routine	Description
getpeername	Returns the name of the connected peer.
getsockname	Returns the name associated with a socket.
getsockopt	Returns the options set on a socket.
setsockopt	Sets options on a socket.

getpeername

Returns the name of the connected peer.

Format

#include *types*

#include *socket*

getpeername *(***int** *s,* **struct sockaddr** **name,* **int** **namelen);*

Arguments

s
Is a socket descriptor that has been created using **socket**.

name
Is a pointer to a buffer within which the peer name is to be returned.

namelen
Is an address of an integer that specifies the size of the **name** buffer. On return, it will be modified to reflect the actual length (in bytes) of the **name** returned.

Description

The **getpeername** routine returns the name of the peer connected to the socket descriptor specified.

See also **bind**, **getsockname**, and **socket**.

Return Values

0	Indicates success.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Resources were insufficient in the system to perform the operation.
[EFAULT]	The name parameter is not a valid part of the user address space.

getsockname

Returns the name associated with a socket.

Format

#include *types*

#include *socket*

int getsockname *(***int** *s,* **struct sockaddr** **name,* **int** **namelen);*

Arguments

s
Is a socket descriptor created with **socket** and bound to the socket name with **bind**.

name
Is a pointer to the buffer in which **getsockname** should return the socket name.

namelen
Is a pointer to an integer specifying the size of the buffer pointed to by name. On return, the integer contains the actual size of the name returned (in bytes).

Description

The **getsockname** routine returns the current name for the specified socket descriptor. The name is a format specific to the address family (AF_INET) assigned to the socket.

Bind makes the association of the name to the socket, not **getsockname**.

See also **bind** and **socket**.

Return Values

0	Indicates success.
-1	Indicates that an error has occurred.
[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[ENOBUFS]	Resources were insufficient in the system to perform the operation.
[EFAULT]	The name parameter is not a valid part of the user address space.

getsockopt

Returns the options set on a socket.

Format

#include *types*

#include *socket*

int getsockopt (**int** *s,* **int** *level,* **int** *optname,* **char** **optval,* **int** **optlen);*

Arguments

s
Is a socket descriptor created by **socket**.

level
Specifies the protocol level for which the socket options are desired. It may have one of the following values:

SOL_SOCKET	Get the options at the socket level.
p	Any protocol number. Get the options for protocol level p. See the in.h file for the various IPPROTO values.

optname
Is interpreted by the protocol that is specified in the level. Options at each protocol level are documented with the protocol. See **setsockopt** for socket level options.

optval
Points to a buffer in which the value of the specified option should be placed by **getsockopt**.

optlen
Points to an integer containing the size of the buffer pointed to by **optval**. On return, the integer will be modified to contain the actual size of the option value returned.

Description

This routine gets information on socket options. See the appropriate protocol for information on available options at each protocol level.

Return Values

0	Indicates success.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[ENOPROTOOPT]	The option is unknown or the protocol is unsupported.
[EFAULT]	The name parameter is not a valid part of the user address space.

setsockopt

Sets options on a socket.

Format

#include *types*

#include *socket*

int setsockopt (**int** *s*,**int** *level*, **int** *optname*,**char** **optval*, **int** **optlen*);

Arguments

s
Is a socket descriptor created by **socket**.

level
Specifies the protocol level for which the socket options are to be modified. It may have one of the following values:

SOL_SOCKET	Set the options at the socket level.
p	Any protocol number. Set the options for protocol level p. See the in.h file for the various IPPROTO values.

optname
Is interpreted by the protocol specified in **level**. Options at each protocol level are documented with the protocol.

The options available at the socket level are:

SO_REUSEADDR	Allow local address reuse.
SO_KEEPALIVE	Keep connections alive (TCP/IP).
SO_DONTROUTE	Do not apply routing on outgoing messages.

SO_LINGER	Linger on close if data present (TCP/IP).
SO_BROADCAST	Permit sending of broadcast messages.

SO_REUSEADDR indicates the rules used in validating addresses supplied in a **bind** call should allow reuse of local addresses.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified through an EPIPE error.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER delays the internal socket deletion portion of **close** until either the data has been transmitted, or the device times out (approximately eight minutes).

SO_BROADCAST is used to enable or disable broadcasting on the socket.

optval
Points to a buffer containing the parameters of the specified option.

All socket level options other than **SO_LINGER** take an integer parameter that should be nonzero if the option is to be enabled, or zero if it is to be disabled.

SO_LINGER uses a **linger** structure parameter defined in the socket.h file. This structure specifies the desired state of the option and the linger interval. The option value for the **SO_LINGER** command is the address of a **linger** structure. See Section D.4.4 for a description of the **linger** structure.

If the socket promises reliable delivery of data and **l_onoff** is nonzero, the system will block the process on the **close** attempt until it is able to transmit the data or until it decides it is unable to deliver the information. A timeout period, called the linger interval, is specified in **l_linger**.

If **l_onoff** is set to zero and a **close** is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

optlen
Points to an integer containing the size of the buffer pointed to by **optval**.

setsockopt

Description

The **setsockopt** routine manipulates options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as **SOL_SOCKET**. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option must be supplied. For example, to indicate an option is to be interpreted by the TCP protocol, level should be set to the protocol number (IPPROTO_TCP) of TCP. See in.h file for the various IPPROTO values.

Return Values

0	Indicates success.
-1	Indicates that an error has occurred.
[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The socket descriptor references a file, not a socket.
[ENOPROTOOPT]	The option is unknown.
[EFAULT]	The name parameter is not a valid part of the user address space.

D.9 Communication Support Routines

The communication support routines perform operations, such as searching databases, converting byte order of network and host addresses, reading records, and returning Internet addresses. Refer to Table D–4 for a description of these routines.

Table D–4: Supported Communication Routines

Routine	Description
gethostbyaddr	Searches the host database for a host record with a given address.
gethostbyname	Searches the host database for a host record with given name or alias.
gethostent	Reads the next record in the host database.
gethostname	Returns the name of the current host.
getnetbyaddr	Searches the network database for a network record with a given address.
getnetbyname	Searches the network database for a network record with a given name or alias.
getnetent	Reads the next record in the network database.
htonl	Converts longwords from network to host byte order.
htons	Converts short integers from network to host byte order.
ntohl	Converts longwords from host to network byte order.
ntohs	Converts short integers from host to network byte order.
inet_addr	Converts Internet addresses in text form into numeric Internet addresses.
inet_lnaof	Returns the local network address portion of an Internet address.
inet_makeaddr	Returns an Internet address given a network address and a local address on that network.
inet_netof	Returns the Internet network address portion of an Internet address.

(continued on next page)

Table D–4 (Cont.): Supported Communication Routines

Routine	Description
inet_network	Converts a NULL-terminated text string representing an Internet network address into a network address in network byte order.
inet_ntoa	Converts an Internet address into an ASCIZ (NULL-terminated) string.
vaxc$get_sdc	Returns the socket device's VAX/VMS I/O channel associated with a socket descriptor.

gethostbyaddr

Searches the host database sequentially from the beginning of the database for a host record with a given address.

Format

#include *netdb*

struct hostent *gethostbyaddr *(***char** **addr,* **int** *len,* **int** *type);*

Arguments

addr
Specifies a pointer to a series of bytes in network order specifying the address of the host sought. This argument does not point to an ASCII string.

len
Specifies the number of bytes in the address pointed to by the **addr** argument.

type
Specifies the type of address format being sought. Currently, only AF_INET is supported.

Description

The **gethostbyaddr** routine finds the first host record in the host database with the given address.

The **gethostent**, **gethostbyaddr**, and **gethostbyname** routines all use a common static area for their return values. This means that subsequent calls to any of these routines will overwrite any existing host entry. You must make a copy of the host entry if you wish to save it.

Return Values

NULL	Indicates an error.
x	A pointer to an object with the **hostent** structure. See Section D.4.1 for a description of the **hostent** structure.

gethostbyname

Searches the host database sequentially from the beginning of the database for a host record with a given name or alias.

Format

#include *netdb*

struct hostent *gethostbyname (**char** **name);*

Arguments

name
Is a pointer to a NULL-terminated character string containing the name or an alias of the host sought.

Description

The **gethostbyname** routine finds the first host in the host database with the given name or alias.

The **gethostent**, **gethostbyaddr**, and **gethostbyname** routines all use a common static area for their return values. This means that subsequent calls to any of these routines will overwrite any existing host entry. You must make a copy of the host entry if you wish to save it.

Return Values

NULL	Indicates an error.
x	A pointer to an object with the hostent structure. See Section D.4.1 for a description of the **hostent** structure.

gethostent

Reads the next record in the host database, opening the database if necessary. This routine is not currently supported by the VMS/ULTRIX Connection product on a system running the VMS operating system, but it is supported by the ULTRIX operating system.

Format

#include *netdb*

struct hostent *gethostent *();*

Description

The **gethostent** routine allows the records in the host database to be read sequentially in the order in which they appear in the database.

The **gethostent**, **gethostbyaddr**, and **gethostbyname** routines all use a common static area for their return values. This means that subsequent calls to any of these routines will overwrite any existing host entry. You must make a copy of the host entry if you wish to save it.

Return Values

NULL	Indicates an error.
x	A pointer to an object with the **hostent** structure. See Section D.4.1 for a description of the **hostent** structure.

gethostname

Returns the name currently associated to the host.

Format

#include *types*

#include *socket*

gethostname (**char** **name,* **int** *namelen);*

Arguments

name
Specifies the address of a buffer into which the name should be written. The returned name is NULL-terminated unless sufficient space is not provided.

namelen
Specifies the size of the buffer pointed to by name.

Description

The **gethostname** routine returns the translation of the logical UCX$INET_HOST when used with the VMS/ULTRIX Connection on a VMS system.

Return Values

0	Indicates success.
-1	Indicates that an error has occurred and is further specified in the global **errno**.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

getnetbyaddr

Searches the network database sequentially from the beginning of the database for a network record with a given address.

Format

#include *netdb*

struct netent *getnetbyaddr (**long** *net,* **int** *type);*

Arguments

net
Specifies the network number of the network database entry required. It should be specified in host byte order.

type
Specifies the type of network sought. Currently, only AF_INET is supported.

Description

The **getnetbyaddr** routine finds the first network record in the network database with the given address.

The **getnetent**, **getnetbyaddr**, and **getnetbyname** routines all use a common static area for their return values. This means that subsequent calls to any of these routines will overwrite any existing network entry. You must make a copy of the network entry if you wish to save it.

Return Values

NULL	Indicates EOF or an error.
x	A pointer to an object with the **netent** structure. See Section D.4.6 for a description of the **netent** structure.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

getnetbyname

Searches the network database sequentially from the beginning of the database for a network record with a given name or alias.

Format

#include *netdb*

struct netent *getnetbyname *(***char** **name);*

Argument

name
Is a pointer to a NULL-terminated character string of the name or an alias of the network sought.

Description

The **getnetbyname** routine finds the first host in the network database with the given name or alias.

The **getnetent**, **getnetbyaddr**, and **getnetbyname** routines all use a common static area for their return values. This means that subsequent calls to any of these routines will overwrite any existing network entry. You must make a copy of the network entry if you wish to save it.

Return Values

NULL	Indicates EOF or an error.
x	A pointer to an object with the **netent** structure. See Section D.4.6 for a description of the **netent** structure.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

getnetent

Reads the next record in the network database, opening the database if necessary. This routine is not currently supported by the VMS/ULTRIX Connection product on a VMS operating system, but it is supported by the ULTRIX operating system.

Format

#include *netdb*

struct netent *getnetent *();*

Description

The **getnetent** routine allows the records in the network database to be read sequentially in the order in which they appear in the database.

The **getnetent**, **getnetbyaddr**, and **getnetbyname** routines all use a common static area for their return values. This means that subsequent calls to any of these routines will overwrite any existing network entry. You must make a copy of the network entry if you wish to save it.

Return Values

NULL	Indicates EOF or an error.
x	a pointer to an object with the **netent** structure. See Section D.4.6 for a description of the **netent** structure.

htonl

Converts longwords from host to network byte order.

Format

#include *in*

unsigned long int htonl *(***unsigned long int** *hostlong);*

Argument

hostlong
Is a longword in host (VAX) byte order. All integers on the VAX system are in host byte order unless otherwise specified.

Description

This routine converts 32-bit unsigned integers from host byte order to network byte order.

The network byte order is the format in which data bytes are supposed to be transmitted through a network. All hosts on a network must send data in network byte order. Not all hosts have an internal data representation format that is identical to the network byte order. The host byte order is the format in which bytes are ordered internally on a specific host.

The host byte order on VAX systems differs from the network order.

This routine is most often used with Internet addresses and ports as returned by **gethostent** and **getservent**, and when manipulating values in the structures. **Network byte order** places the byte with the most significant bits at lower addresses, whereas the VAX system places the most significant bits at the highest address.

Return Values

x	A longword in network byte order.

htons

Converts short integers from host to network byte order.

Format

#include *in*

unsigned short int htons *(***unsigned short int** *hostshort);*

Argument

hostshort
Is a short integer in host (VAX) byte order. All short integers on the VAX system are in host byte order unless otherwise specified.

Description

This routine converts 16-bit unsigned integers from host byte order to network byte order.

The network byte order is the format in which data bytes are suppose to be transmitted through a network. All hosts on a network must send data in network byte order. Not all hosts have an internal data representation format that is identical to the network byte order. The host byte order is the format in which bytes are ordered internally on a specific host.

The host byte order on VAX systems differs from the network order.

This routine is most often used with Internet addresses and ports as returned by **gethostent** and **getservent**, and when manipulating values in the structures. Network byte order places the byte with the most significant bits at lower addresses, whereas the VAX system places the most significant bits at the highest address.

Return Values

x	A short integer in network byte order. Integers in network byte order cannot be used for arithmetic computation on the VAX system.

inet_addr

Converts Internet addresses in text form into numeric (binary) Internet addresses.

Format

#include *in*

#include *inet*

int inet_addr *(***char** **cp);*

Argument

cp
Is a pointer to a NULL-terminated character string containing an Internet address in the standard Internet "." format.

Description

This routine returns an Internet address in network byte order when given as its argument an ASCIZ (NULL-terminated) string representing the address in the Internet standard "." notation.

Internet addresses specified using the "." notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on the VAX system, the bytes referred to above appear in binary as “d.c.b.a”. That is, VAX bytes are ordered from least significant to most significant.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a "." address expression may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; a leading 0 implies octal, otherwise, the number is interpreted as decimal).

Return Values

-1	Indicates that cp does not point to a proper Internet address.
x	Is an Internet address in network byte order.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

inet_lnaof

Returns the local network address portion of an Internet address.

Format

#include *in*

#include *inet*

int inet_lnaof *(***struct in_addr** *in);*

Argument

in
Is an Internet address.

Description

This routine returns the local network address (lna) portion of a full Internet address.

Return Values

x	The local network portion of an Internet address in byte order host.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

inet_makeaddr

Returns an Internet address given a network address and a local address on that network.

Format

#include *in*

#include *inet*

struct in_addr inet_makeaddr *(***int** *net,* **int** *lna)* **;**

Arguments

net
Is an Internet network address in host byte order.

lna
Is a local network address on network **net** in host byte order.

Description

This routine combines the **net** and **lna** arguments into a single Internet address.

Return Values

x	An Internet address in network byte order.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

inet_netof

Returns the Internet network address portion of an Internet address.

Format

#include *in*

#include *inet*

int inet_netof (**struct in_addr** *in);*

Argument

in
Is an Internet address.

Description

This routine returns the Internet network address (net) portion of a full Internet address.

Return Values

x	The Internet network portion of an Internet address in host byte order.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

inet_network

Converts a text string representing an Internet network address in the standard Internet "." notation into an Internet network address as machine-format integer values.

Format

#include *in*

#include *inet*

int inet_network *(***char** **cp);*

Argument

cp
Is a pointer to an ASCIZ (NULL-terminated) character string containing a network address in the standard Internet "." format.

Description

This routine returns an Internet network address as machine-format integer values when given as its argument an ASCIZ string representing the address in the Internet standard "." notation.

Return Values

-1	Indicates that cp does not point to a proper Internet network address.
x	Is an Internet network address as machine-format integer values.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

inet_ntoa

Converts an internet address into a text string representing the address in the standard Internet "." notation.

Format

#include *in*

#include *inet*

char *inet_ntoa *(***struct in_addr** *in);*

Argument

in
Is an Internet address in network byte order.

Description

This routine is used to convert an Internet address into an ASCIZ (NULL-terminated) string representing that address in the standard Internet "." notation.

WARNING

Arguments should not be passed as integers because of how the VAX C language handles **struct** arguments.

Because the string is returned in a static buffer that will be overwritten by successive calls to **inet_ntoa**, it is recommended to copy the string to a safe place.

Return Values

x	A pointer to a string containing the Internet address in "." notation.
[EFAULT]	The buffer described by **name** and **namelen** is not a valid, writeable part of the user address space.

ntohl

Converts longwords from network to host byte order.

Format

#include *in*

unsigned long int ntohl *(***unsigned long int** *netlong)* **;**

Argument

netlong
Is a longword in network byte order. Integers in network byte order cannot be used for arithmetic computation on the VAX system.

Description

This routine converts 32-bit unsigned integers from network byte order to host byte order.

The network byte order is the format in which data bytes are supposed to be transmitted through a network. All hosts on a network must send data in network byte order. Not all hosts have an internal data representation format that is identical to the network byte order. The host byte order is the format in which bytes are ordered internally on a specific host.

The host byte order on VAX systems differs from the network order.

This routine is most often used with Internet addresses and ports as returned by **gethostent** and **getservent**, and when manipulating values in the structures. **Network byte order** places the byte with the most significant bits at lower addresses, whereas the VAX system places the most significant bits at the highest address.

Return Values

x	A longword in host byte order.

ntohs

Converts short integers from network to host byte order.

Format

#include *in*

unsigned short int ntohs *(***unsigned short int** *netshort);*

Argument

netshort
Is a short integer in network byte order. Integers in network byte order cannot be used for arithmetic computation on the VAX system.

Description

This routine converts 16-bit unsigned integers from network byte order to host byte order.

The network byte order is the format in which data bytes are suppose to be transmitted through a network. All hosts on a network must send data in network byte order. Not all hosts have an internal data representation format that is identical to the network byte order. The host byte order is the format in which bytes are ordered internally on a specific host.

The host byte order on VAX systems differs from the network order.

This routine is most often used with Internet addresses and ports as returned by **gethostent** and **getservent**, and when manipulating values in the structures. **Network byte order** places the byte with the most significant bits at lower addresses, whereas the VAX system places the most significant bits at the highest address.

Return Values

x	A short integer in host (VAX) byte order.

vaxc$get_sdc

Returns the socket device channel associated with a socket descriptor for direct use with the VMS/ULTRIX Connection product.

Format

#include *socket*

short int vaxc$get_sdc **(int** *s);*

Argument

s
Is a socket descriptor.

Description

This routine returns the Socket Device Channel (SDC) associated with a socket. C socket descriptors are normally used either as file descriptors or with one of the routines that takes an explicit socket descriptor as its argument. C sockets are implemented using VMS/ULTRIX Connection Socket Device Channels. This routine returns the Socket Device Channel used by a given socket descriptor so that you can use the VMS/ULTRIX Connection's facilities directly by means of various I/O system services ($QIO).

Return Values

0	Indicates that **s** is not an open socket descriptor.
x	Is the Socket Device Channel number.

D.10 Programming Examples

This section provides VAX C socket communications programming examples.

Example D–1 shows a TCP/IP server using the IPC socket interface.

Example D–1: TCP/IP Server

```
/*====================================================================
*
*                          Copyright (C) 1989 by
*              Digital Equipment Corporation, Maynard, Mass.
*
* This software is furnished under a license and may be used and  copied
* only  in  accordance  with  the  terms  of  such  license and with the
* inclusion of the above copyright notice.  This software or  any  other
* copies  thereof may not be provided or otherwise made available to any
* other person.  No title to and ownership of  the  software  is  hereby
* transferred.
*
* The information in this software is subject to change  without  notice
* and  should  not  be  construed  as  a commitment by Digital Equipment
* Corporation.
*
* Digital assumes no responsibility for the use or  reliability  of  its
* software on equipment that is not supplied by Digital.
*
*
*
*  FACILITY:
*        INSTALL
*
*
*  ABSTRACT:
*        This is an example of a TCP/IP server using the IPC
*        socket interface.
*
*
*  ENVIRONMENT:
*        UCX V1.2 or higher, VMS V5.2 or higher
*
*        This example is portable to Ultrix. The include
*        files are conditionally defined for both systems, and
*        "perror" is used for error reporting.
*
*        To link in VAXC/VMS you must have the following
*        entries in your .opt file:
*           sys$library:ucx$ipc.olb/lib
*           sys$share:vaxcrtl.exe/share
```

(continued on next page)

```
*
*  AUTHORS:
*        UCX Developer
*
*  CREATION DATE: May 23, 1989
*
*  MODIFICATION HISTORY:
*
*/

/*
*
*  INCLUDE FILES
*
*/

#ifdef VAXC
#include  <errno.h>
#include  <types.h>
#include  <stdio.h>
#include  <socket.h>
#include  <in.h>
#include  <netdb.h>               /* change hostent to comply with BSD 4.3 */
#include  <inet.h>
#include  <ucx$inetdef.h>         /* INET symbol definitions */
#else
#include <errno.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/uio.h>
#endif

/*
* Functional Description
*
*        This example creates a socket of type SOCK_STREAM (TCP),
*        binds and listens on the socket, receives a message
*        and closes the connection.
*        Error messages are printed to the screen.
```

(continued on next page)

```
*
*       IPC calls used:
*       accept
*       bind
*       close
*       gethostbyname
*       listen
*       recv
*       shutdown
*       socket
*
*
* Formal Parameters
*       The server program expects one parameter:
*       portnumber ... port number where it will listen
*
*
* Routine Value
*
*       Status
*/

/*--------------------------------------------------------------------*/
main(argc,argv)
int         argc;
char        **argv;
{

        int     sock_2, sock_3;                 /* sockets */
static  char    message[BUFSIZ];
static struct   sockaddr_in sock2_name;         /* Address struct for socket2.*/
static struct   sockaddr_in retsock2_name;      /* Address struct for socket2.*/
        struct  hostent hostentstruct;          /* Storage for hostent data.  */
        struct  hostent *hostentptr;            /* Pointer to hostent data.   */
static  char    hostname[256];                  /* Name of local host.        */
        int     flag;
        int     retval;                         /* helpful for debugging */
        int     namelength;

        /*
         * Check input parameters.
         */
        if (argc != 2 )
                {
                printf("Usage: server portnumber.\n");
                exit();
                }
```

(continued on next page)

Example D–1 (Cont.): TCP/IP Server

```
/*
 * Open socket 2: AF_INET, SOCK_STREAM.
 */
if ((sock_2 = socket (AF_INET, SOCK_STREAM, 0)) == -1)
        {
        perror( "socket");
        exit();
        }

/*
 * Get the host local name.
 */
retval = gethostname(hostname,sizeof hostname);
if (retval)
        {
        perror ("gethostname");
        cleanup (1, sock_2, 0);
        }

/*
 * Get pointer to network data structure for socket 2.
 */
if ((hostentptr = gethostbyname (hostname)) == NULL)
        {
        perror( "gethostbyname");
        cleanup(1, sock_2, 0);
        }

/*
 * Copy hostent data to safe storage.
 */
hostentstruct = *hostentptr;

/*
 * Fill in the name & address structure for socket 2.
 */
sock2_name.sin_family = hostentstruct.h_addrtype;
sock2_name.sin_port = htons(atoi(argv[1]));
sock2_name.sin_addr = * ((struct in_addr *) hostentstruct.h_addr);

/*
 * Bind name to socket 2.
 */
retval = bind (sock_2, &sock2_name, sizeof sock2_name);
if (retval)
        {
        perror("bind");
        cleanup(1, sock_2, 0);
        }
```

(continued on next page)

```
        /*
         * Listen on socket 2 for connections.
         */
        retval = listen (sock_2, 5);
        if (retval)
                {
                perror("listen");
                cleanup(1, sock_2, 0);
                }

        /*
         * Accept connection from socket 2:
         * accepted connection will be on socket 3.
         */
        namelength = sizeof (sock2_name);
        sock_3 = accept (sock_2, &sock2_name, &namelength);
        if (sock_3 == -1)
                {
                perror ("accept");
                cleanup( 2, sock_2, sock_3);
                }

        /*
         * Receive message from socket 1.
         */
        flag = 0;        /* maybe 0 or MSG_OOB or MSG_PEEK */

        retval = recv(sock_3, message ,sizeof (message), flag);
        if (retval == -1)
                {
                perror ("receive");
                cleanup( 2, sock_2, sock_3);
                }
        else
                printf (" %s\n", message);

        /*
         * Call cleanup to shutdown and close sockets.
         */
        cleanup(2, sock_2, sock_3);

} /* end main */
```

(continued on next page)

Example D–1 (Cont.): TCP/IP Server

```
/*--------------------------------------------------------------*/

cleanup(how_many, sock1, sock2)
int         how_many;
int         sock1, sock2;

{
        int         retval;

        /*
         * Shutdown and close sock1 completely.
         */
        retval = shutdown(sock1,2);
        if (retval == -1)
                perror ("shutdown");

        retval = close (sock1);
        if (retval)
                perror ("close");


        /*
         * If given, shutdown and close sock2.
         */
        if (how_many == 2)
        {
                retval = shutdown(sock2,2);
                if (retval == -1)
                        perror ("shutdown");

                retval = close (sock2);
                if (retval)
                        perror ("close");
        }

        exit();

} /* end cleanup*/
```

Example D–2 shows a TCP/IP client using the IPC socket interface.

Example D–2: TCP/IP Client

```
/*==================================================================
 *
 *                            Copyright (C) 1989 by
 *               Digital Equipment Corporation, Maynard, Mass.
 *
 * This software is furnished under a license and may be used and  copied
 * only  in  accordance  with  the  terms  of  such  license and with the
 * inclusion of the above copyright notice.  This software or  any  other
 * copies  thereof may not be provided or otherwise made available to any
 * other person.  No title to and ownership of  the  software  is  hereby
 * transferred.
 *
 * The information in this software is subject to change  without  notice
 * and  should  not  be  construed  as  a commitment by Digital Equipment
 * Corporation.
 *
 * Digital assumes no responsibility for the use or  reliability  of  its
 * software on equipment that is not supplied by Digital.
 *
 *
 *
 *  FACILITY:
 *        INSTALL
 *
 *
 *  ABSTRACT:
 *        This is an example of a TCP/IP client using the IPC
 *        socket interface.
 *
 *
 *  ENVIRONMENT:
 *        UCX V1.2 or higher, VMS V5.2 or higher
 *
 *        This example is portable to Ultrix. The include
 *        files are conditionally defined for both systems, and
 *        "perror" is used for error reporting.
 *
 *        To link in VAXC/VMS you must have the following
 *        entries in your .opt file:
 *           sys$library:ucx$ipc.olb/lib
 *           sys$share:vaxcrtl.exe/share
```

(continued on next page)

Example D–2 (Cont.): TCP/IP Client

```
*
*  AUTHORS:
*        UCX Developer
*
*  CREATION DATE: May 23, 1989
*
*  MODIFICATION HISTORY:
*
*/

/*
*
*  INCLUDE FILES
*
*/

#ifdef VAXC
#include  <errno.h>
#include  <types.h>
#include  <stdio.h>
#include  <socket.h>
#include  <in.h>
#include  <netdb.h>                /* change hostent to comply with BSD 4.3*/
#include  <inet.h>
#include  <ucx$inetdef.h>        /* INET symbol definitions */
#else
#include <errno.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/uio.h>
#endif

/*
*
*   MACRO DEFINITIONS
*
*/

#ifndef vms
#define TRUE 1
#define FALSE 0
#endif
```

(continued on next page)

Example D–2 (Cont.): TCP/IP Client

```
/*
* Functional Description
*
*       This example creates a socket of type SOCK_STREAM (TCP),
*       initiates a connection to the remote host, sends
*       a message to the remote host, and closes the connection.
*       Error messages are printed to the screen.
*
*       IPC calls used:
*       close
*       connect
*       gethostbyname
*       send
*       shutdown
*       socket
*
*
* Formal Parameters
*       The client program expects two parameters:
*       hostname ... name of remote host
*       portnumber ... port where remote host(server) is listening
*
*
* Routine Value
*
*       Status
*/

/*--------------------------------------------------------------------*/
main(argc,argv)
int         argc;
char        **argv;
{

        int     sock_1;                         /* socket */
static  char    message[] = "Hi there.";
static  struct  sockaddr_in sock2_name;         /* Address struct for socket2.*/
        struct  hostent         hostentstruct;  /* Storage for hostent data.  */
        struct  hostent         *hostentptr;    /* Pointer to hostent data.   */
        static  char            hostname[256];  /* Name of local host.        */
        int     flag;
        int     retval;                          /* helpful for debugging */
        int     shut = FALSE;                 /* flag to cleanup */
```

(continued on next page)

```
/*
 * Check input parameters.
 */
if (argc != 3 )
        {
        printf("Usage: client hostname portnumber.\n");
        exit();
        }

/*
 * Open socket 1: AF_INET, SOCK_STREAM.
 */
if ((sock_1 = socket (AF_INET, SOCK_STREAM, 0)) == -1)
        {
        perror( "socket");
        exit();
        }

/*
 *Get pointer to network data structure for socket 2 (remote host).
 */
if ((hostentptr = gethostbyname (argv[1])) == NULL)
        {
        perror( "gethostbyname");
        cleanup(shut, sock_1);
        }


/*
 * Copy hostent data to safe storage.
 */
hostentstruct = *hostentptr;

/*
 * Fill in the name & address structure for socket 2.
 */
sock2_name.sin_family = hostentstruct.h_addrtype;
sock2_name.sin_port = htons(atoi(argv[2]));
sock2_name.sin_addr = * ((struct in_addr *) hostentstruct.h_addr);

/*
 * Connect socket 1 to sock2_name.
 */
retval = connect(sock_1, &sock2_name, sizeof (sock2_name));
if (retval)
        {
        perror("connect");
        cleanup(shut, sock_1);
        }
```

(continued on next page)

```
        /*
         * Send message to socket 2.
         */
        flag = 0;           /* maybe 0 or MSG_OOB */
        retval = send(sock_1, message ,sizeof (message), flag);
        if (retval < 0)
                {
                perror ("send");
                shut = TRUE;
                }

        /*
         * Call cleanup to shutdown and close socket.
         */
        cleanup(shut, sock_1);

 } /* end main */

/*-----------------------------------------------------------*/

cleanup(shut, socket)
int         shut;
int         socket;

{
        int         retval;

        /*
         * Shutdown socket completely -- only if it was connected
         */
        if (shut) {
                retval = shutdown(socket,2);
                if (retval == -1)
                        perror ("shutdown");
        }

        /*
         * Close socket.
         */
        retval = close (socket);
        if (retval)
                perror ("close");

        exit();

 } /* end main */
```

Example D–3 shows a UDP/IP server using the IPC socket interface.

Example D–3: UDP Server

```
/*=====================================================================
 *
 *                          Copyright (C) 1989 by
 *              Digital Equipment Corporation, Maynard, Mass.
 *
 * This software is furnished under a license and may be used and  copied
 * only  in  accordance  with  the  terms  of  such  license and with the
 * inclusion of the above copyright notice.  This software or  any  other
 * copies  thereof may not be provided or otherwise made available to any
 * other person.  No title to and ownership of  the  software  is  hereby
 * transferred.
 *
 * The information in this software is subject to change  without  notice
 * and  should  not  be  construed  as  a commitment by Digital Equipment
 * Corporation.
 *
 * Digital assumes no responsibility for the use or  reliability  of  its
 * software on equipment that is not supplied by Digital.
 *
 *
 *
 *  FACILITY:
 *        INSTALL
 *
 *
 *  ABSTRACT:
 *        This is an example of a UDP/IP server using the IPC
 *        socket interface.
 *
 *
 *  ENVIRONMENT:
 *        UCX V1.2 or higher, VMS V5.2 or higher
 *
 *        This example is portable to Ultrix. The include
 *        files are conditionally defined for both systems, and
 *        "perror" is used for error reporting.
 *
 *        To link in VAXC/VMS you must have the following
 *        entries in your .opt file:
 *           sys$library:ucx$ipc.olb/lib
 *           sys$share:vaxcrtl.exe/share
```

(continued on next page)

Example D–3 (Cont.): UDP Server

```
*
*   AUTHORS:
*         UCX Developer
*
*   CREATION DATE: May 23, 1989
*
*   MODIFICATION HISTORY:
*
*/


/*
*
*   INCLUDE FILES
*
*/

#ifdef VAXC
#include  <errno.h>
#include  <types.h>
#include  <stdio.h>
#include  <socket.h>                /* timeval declared here */
#include  <in.h>
#include  <netdb.h>                 /* change hostent to comply with BSD 4.3 */
#include  <inet.h>
#include  <ucx$inetdef.h>           /* INET symbol definitions */
#else
#include <errno.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/uio.h>
#include <time.h>                  /* timeval declared here */
#endif
```

(continued on next page)

Example D–3 (Cont.): UDP Server

```
/*
* Functional Description
*
*       This example creates a socket of type SOCK_DGRAM (UDP), binds
*       it, and selects to receive a message on the socket.
*       Error messages are printed to the screen.
*
*       IPC calls used:
*       bind
*       close
*       gethostbyname
*       recvfrom
*       select
*       shutdown
*       socket
*
*
* Formal Parameters
*       The server program expects one parameter:
*       portnumber ... port where it is listening
*
*
* Routine Value
*
*       Status
*/

/*--------------------------------------------------------------------*/
main(argc, argv)
int         argc;
char        **argv;
{

        unsigned long rmask,wmask,emask;
        int     sock_2;                 /* Socket 2  descriptor.      */
        int     buflen,fromlen;
        char    recvbuf[BUFSIZ];
static  struct  sockaddr_in sock1_name; /* Address struct for socket1.*/
static  struct  sockaddr_in sock2_name; /* Address struct for socket2.*/
        int     namelength;
        struct  hostent hostentstruct;  /* Storage for hostent data.  */
        struct  hostent *hostentptr;    /* Pointer to hostent data.   */
        static  char hostname[256];     /* Name of local host.        */
        int     retval;
        int     flag;
        struct  timeval timeout;
```

(continued on next page)

```
        /*
         * Check input parameters
         */
        if (argc != 2 )
                {
                printf("Usage: server portnumber.\n");
                exit();
                }

        /*
         * Open socket 2: AF_INET, SOCK_DGRAM.
         */
        if ((sock_2 = socket (AF_INET, SOCK_DGRAM, 0)) == -1)
                {
                perror( "socket");
                exit();
                }

        /*
         * Get the local host  name.
         */
        retval = gethostname(hostname,sizeof hostname);
        if (retval)
                {
                perror ("gethostname");
                cleanup(sock_2);
                 }

        /*
         * Get pointer to network data structure for local host.
         */
        if ((hostentptr = gethostbyname (hostname)) == NULL)
                {
                perror( "gethostbyname");
                cleanup(sock_2);
                 }

        /*
         * Copy hostent data to safe storage.
         */
        hostentstruct = *hostentptr;

        /*
         * Fill in the address structure for socket 2.
         */
        sock2_name.sin_family = hostentstruct.h_addrtype;
        sock2_name.sin_port = htons(atoi(argv[1]));
        sock2_name.sin_addr = * ((struct in_addr *) hostentstruct.h_addr);
```

(continued on next page)

Example D–3 (Cont.): UDP Server

```
/*
 * Bind name to socket 2.
 */
retval = bind (sock_2, &sock2_name, sizeof sock2_name);
if (retval)
        {
        perror("bind");
        cleanup(sock_2);
         }


/*
 * Select socket to receive message.
 */
emask = wmask = 0;
rmask = (1<<sock_2);  /* set read mask */
timeout.tv_sec = 30;
timeout.tv_usec = 0;

retval = select(32,&rmask,&wmask,&emask, &timeout);
switch (retval)
{
  case -1:
        {
        perror("select");
        cleanup(sock_2);
        }
  case 0:
        {
        printf("Select timed out with status 0.\n");
        cleanup(sock_2);
        }
  default:
        if ((rmask & (1<<sock_2)) == 0)
                {
                printf("Select not reading on sock_2.\n");
                cleanup(sock_2);
                }
} /*switch*/
```

(continued on next page)

Example D–3 (Cont.): UDP Server

```
        /*
         * Recvfrom buffer - from sock1 on sock2.
         */
        buflen = sizeof(recvbuf);
        fromlen = sizeof(sock1_name);
        flag = 0;         /* flag may be MSG_OOB and/or MSG_PEEK */

        retval = recvfrom(sock_2, recvbuf, buflen, flag, &sock1_name, &fromlen);
        if (retval == -1)
                perror("recvfrom");
           else
                printf (" %s\n", recvbuf);

        /*
         * Call cleanup to shutdown and close socket.
         */
        cleanup(sock_2);

 } /* end main */

/*-------------------------------------------------------------*/

cleanup(socket)
int  socket;

{
        int   retval;

        /*
         * Shutdown socket  completely.
         */
        retval = shutdown(socket,2);
        if (retval == -1)
                perror ("shutdown");


        /*
         * Close socket.
         */
        retval = close (socket);
        if (retval)
                perror ("close");

        exit();

} /* end cleanup */
```

Example D–4 shows a UDP/IP client using the IPC socket interface.

Example D–4: UDP Client

```
/*=====================================================================
 *
 *                        Copyright (C) 1989 by
 *              Digital Equipment Corporation, Maynard, Mass.
 *
 * This software is furnished under a license and may be used and  copied
 * only  in  accordance  with  the  terms  of  such  license and with the
 * inclusion of the above copyright notice.  This software or  any  other
 * copies  thereof may not be provided or otherwise made available to any
 * other person.  No title to and ownership of  the  software  is  hereby
 * transferred.
 *
 * The information in this software is subject to change  without  notice
 * and  should  not  be  construed  as  a commitment by Digital Equipment
 * Corporation.
 *
 * Digital assumes no responsibility for the use or  reliability  of  its
 * software on equipment that is not supplied by Digital.
 *
 *
 *
 *  FACILITY:
 *        INSTALL
 *
 *
 *  ABSTRACT:
 *        This is an example of a UDP/IP client using the IPC
 *        socket interface.
 *
 *
 *  ENVIRONMENT:
 *        UCX V1.2 or higher, VMS V5.2 or higher
 *
 *        This example is portable to Ultrix. The include
 *        files are conditionally defined for both systems, and
 *        "perror" is used for error reporting.
 *
 *        To link in VAXC/VMS you must have the following
 *        entries in your .opt file:
 *           sys$library:ucx$ipc.olb/lib
 *           sys$share:vaxcrtl.exe/share
```

(continued on next page)

```
*
*  AUTHORS:
*        UCX Developer
*
*  CREATION DATE: May 23, 1989
*
*  MODIFICATION HISTORY:
*
*/

/*
*
*  INCLUDE FILES
*
*/

#ifdef VAXC
#include  <errno.h>
#include  <types.h>
#include  <stdio.h>
#include  <socket.h>
#include  <in.h>
#include  <netdb.h>              /* change hostent to comply with BSD 4.3 */
#include  <inet.h>
#include  <ucx$inetdef.h>        /* INET symbol definitions */
#else
#include <errno.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/uio.h>
#endif

/*
* Functional Description
*
*       This example creates a socket of type SOCK_DGRAM (UDP),
*       binds it, and sends a message to the given host and port number.
*       Error messages are printed to the screen.
*
*       IPC calls used:
*       bind
*       close
*       gethostbyname
*       sendto
*       shutdown
*       socket
```

(continued on next page)

```
*
* Formal Parameters
*       The client program expects two parameters:
*       hostname ... name of remote host
*       portnumber ... port where remote host(server) is listening
*
*
* Routine Value
*
*       Status
*/

/*--------------------------------------------------------------------*/
main(argc, argv)
int        argc;
char       **argv;
{

        int      sock_1;                          /* Socket 1 descriptor.       */
        int      sendlen, tolen;
static  char     sendbuf[] = "Hi there.";
static struct    sockaddr_in sock2_name;          /* Address struct for socket2.*/
        int namelength;
        struct   hostent hostentstruct;          /* Storage for hostent data.  */
        struct   hostent *hostentptr;            /* Pointer to hostent data.   */
        static   char hostname[256];             /* Name of local host.        */
        int         flag;
        int         retval;

        /*
         * Check input parameters.
         */
        if (argc != 3 )
                {
                printf("Usage: client hostname portnumber.\n");
                exit();
                }

        /*
         * Open socket 1: AF_INET, SOCK_DGRAM.
         */
        if ((sock_1 = socket (AF_INET, SOCK_DGRAM, 0)) == -1)
                 {
                perror( "socket");
                exit();
                }
```

(continued on next page)

```
        /*
         *Get pointer to network data structure for given host.
         */
        if ((hostentptr = gethostbyname (argv[1])) == NULL)
                {
                perror( "gethostbyname");
                cleanup(sock_1);
                }


        /*
         * Copy hostent data to safe storage.
         */
        hostentstruct = *hostentptr;

        /*
         * Fill in the address structure for socket 2 (to receive message).
         */
        sock2_name.sin_family = hostentstruct.h_addrtype;
        sock2_name.sin_port = htons(atoi(argv[2]));
        sock2_name.sin_addr = * ((struct in_addr *) hostentstruct.h_addr);

        /*
         * Initialize send block.
         */
        sendlen = sizeof sendbuf;
        tolen = sizeof sock2_name;
        flag = 0;                    /* flag may be MSG_OOB */

        /*
         * Send message from socket 1 to socket 2.
         */
        retval = sendto(sock_1, sendbuf, sendlen, flag, &sock2_name,tolen);
        if (retval == -1)
                {
                perror ( "sendto");
                cleanup(sock_1);
                }

        /*
         * Call cleanup to shutdown and close socket.
         */
        cleanup(sock_1);

 } /* end main */

/*-----------------------------------------------------------*/

cleanup(socket)
int         socket;
```

(continued on next page)

Example D–4 (Cont.): UDP Client

```
{
        int         retval;

        /*
         * Shutdown socket  completely.
         */
        retval = shutdown(socket,2);
        if (retval == -1)
                perror ("shutdown");


        /*
         * Close socket.
         */
        retval = close (socket);
        if (retval)
                perror ("close");

        exit();

} /* end cleanup */
```

Index

A

abort function, REF–3
abs function, REF–4
accept socket routine, D–15
access function, REF–5
ACCVIO
  hardware error, 1–19
acos function, REF–7
addch macro, REF–8
addstr macro, REF–10
alarm function, REF–12, REF–230
Argument list-handling functions and macros, 3–11 to 3–15
Arguments
  variable-length lists, 3–11
ASCII
  table of values, 3–5
asctime function, REF–14
asin function, REF–16
asm call, 1–19
assert function, REF–17
assert macro, REF–17
atan function, REF–19
atan2 function, REF–20
atexit function, REF–21
atof function, REF–23
atoi function, REF–24
atol function, REF–24

B

bind socket routine, D–17
box function, REF–25
brk function, 8–2, REF–27
bsearch function, REF–29

C

C$LIBRARY logical name, 1–6
cabs function, REF–32
calloc function, 8–2, REF–33, REF–35
Carriage control
  FORTRAN, 1–15
  translation
    by VAX C, 1–15 to 1–17
ceil function, REF–34
cfree function, 8–2, REF–35
Character classification functions
  isalnum, REF–165
  isalpha, REF–166
  isascii, REF–168
  iscntrl, REF–170
  isdigit, REF–171
  isgraph, REF–172
  islower, REF–173
  isprint, REF–174
  ispunct, REF–175
  isspace, REF–176
  isupper, REF–177
  isxdigit, REF–178
  program examples, 3–9
Character classification macro return values, 3–5
Character classification macros, 3–4 to 3–9
Character conversion functions, 3–9 to 3–11
  ecvt, REF–71
  fcvt, REF–92
  gcvt, REF–134
  program examples, 3–9
  strtoul, REF–329
  toascii, REF–341
  tolower, REF–342
  _tolower, REF–342

Character conversion functions (Cont.)
  toupper, REF–344
  _toupper, REF–344
chdir function, REF–36
Child process
  creating with **vfork**, REF–365
  executing image
    with exec functions, 5–4
  implementation of, 5–2
  introduction to, 5–1
  program examples, 5–6
  sharing data with **pipe**, 5–6, REF–233
  synchronization with **wait**, 5–6
chmod function, REF–37
chown function, REF–39
C language
  comparison of run-time libraries, A–1 to A–16
  I/O background, 1–11
clear macro, REF–40
clearerr macro, REF–41
clear macro, REF–42
clearok macro, REF–42
clock function, REF–43
close function, REF–44
close socket routine, D–19
clrattr macro, 6–1, REF–46
clrtobot macro, REF–48
clrtoeol macro, REF–49
Command language interpreter
  UNIX, 1–8
connect socket routine, D–20
Conversion specifications
  for I/O functions, 2–7 to 2–13
  input
    table of characters, 2–9
  output
    table of characters, 2–11
cos function, REF–50
cosh function, REF–51
creat function, REF–52, REF–68, REF–94
crmode macro, REF–57
ctermid function, REF–59
ctime function, REF–14, REF–60
ctype
  definition module, 1–5
curscr window, 6–5
Curses, 6–1 to 6–19
  cursor movement, 6–14
  getting started, 6–9 to 6–12
  introduction to, 6–1
  program examples, 6–15
Curses (Cont.)
  terminology, 6–5 to 6–9
    curscr, 6–6
    stdscr, 6–5
    windows, 6–6
  using predefined variables and constants, 6–12
Curses functions
  box, REF–25
  clearok, REF–42
  delwin, REF–65
  endwin, REF–73
  getyx, REF–154
  initscr, REF–161
  leaveok, REF–184
  longname, REF–190
  mvcur, REF–207
  mvwin, REF–209
  mv[w]addch, REF–211
  mv[w]addstr, REF–212
  mv[w]delch, REF–213
  mv[w]getch, REF–214
  mv[w]getstr, REF–215
  mv[w]inch, REF–216
  mv[w]insch, REF–217
  mv[w]insstr, REF–218
  newwin, REF–219
  [no]crmode, REF–57
  [no]echo, REF–70
  [no]nl, REF–222
  [no]raw, REF–253
  overlay, REF–228
  overwrite, REF–229
  scroll, REF–267
  scrollok, REF–268
  subwin, REF–331
  touchwin, REF–343
  wrapok, REF–373
  [w]addch, REF–8
  [w]addstr, REF–10
  [w]clear, REF–40
  [w]clrattr, REF–46
  [w]clrtobot, REF–48
  [w]clrtoeol, REF–49
  [w]delch, REF–62
  [w]deleteln, REF–64
  [w]erase, REF–74
  [w]getch, REF–137
  [w]getstr, REF–151
  [w]inch, REF–160
  [w]insch, REF–162
  [w]insertln, REF–163

Curses functions (Cont.)
[w]insstr, REF–164
[w]move, REF–205
[w]printw, REF–241
[w]refresh, REF–259
[w]scanw, REF–265
[w]setattr, REF–269
[w]standend, REF–298
[w]standout, REF–299
cuserid function, REF–61

D

#define
preprocessor directive, 1–4
Definition modules
descriptions of, C–1 to C–8
Definitions
.H files, 1–6
See also, Standard I/O functions
See also, Substitution
modules, 1–6
delch macro, REF–62
delete function, 1–18, REF–63, REF–260
deleteln macro, REF–64
delwin function, REF–65
difftime function, REF–66
div function, REF–67
dup2 function, REF–68, REF–94, REF–236
dup function, REF–68, REF–94

E

echo macro, REF–70
ecvt function, 3–9, REF–71
edata global symbol, 1–18
end global symbol, 1–18
endwin function, REF–73
Entry points
to VAX C Run-Time Library, B–6 to B–19
erase macro, REF–74
errno
definition module, 4–3
external variable, 4–3
errno variable, 7–3, D–9
Error-handling functions
abort, REF–3
errno values, 4–3
exit, REF–88
_exit, REF–88
perror, REF–231
Error-handling functions (Cont.)
strerror, REF–310
ERR predefined variable, 6–13
etext global symbol, 1–18
exec function, REF–235
exec functions, 5–4
error conditions, 5–5
processing, 5–4
execl function, REF–75
execle function, REF–77
execlp function, REF–80
execv function, REF–82
execve function, REF–84
execvp function, REF–86
_exit function, 5–5, REF–88
exit function, 5–5, REF–88
exp function, REF–89

F

fabs function, REF–90
FALSE identifier, 1–5
fclose function, REF–91, REF–110
fcvt function, 3–9, REF–92
fdopen function, REF–94, REF–236
feof macro, REF–96
ferror macro, REF–97
fflush function, REF–98
fgetc function, REF–99
fgetname function, REF–100
fgetpos function, REF–102
fgets function, REF–104
FILE, 2–7
File descriptor, 2–5, 2–14
VAX C defaults
for VMS logical names, 1–10
fileno macro, REF–106
File pointer, 2–7, 2–14
File protection, REF–37, REF–346
File sharing, 2–6
floor function, REF–107
fmod function, REF–108
fopen function, 2–6, REF–109
fork function, REF–365
fprintf function, 2–6, REF–111
fputc function, REF–113
fputs function, REF–114
fread function, 2–6, REF–115
free function, 8–2, REF–35, REF–117, REF–139
freopen function, REF–118
frexp function, REF–120

fscanf function, REF–122
fseek function, 1–14, 2–6, REF–124, REF–130, REF–347
fsetpos function, REF–126
fstat function, REF–127
ftell function, 2–6, REF–130
ftime function, REF–131
Function prototype, 1–6
Functions
  argument list-handling, 3–11
  character classification, 3–4
  character conversion, 3–4, 3–9
  Curses, 6–1 to 6–4
  entry points of, B–6 to B–17
  error-handling, 4–3 to 4–5
  Standard I/O, 2–1
  string-handling, 3–11
  Terminal I/O, 2–13
  UNIX I/O, 2–5
  VAX C RTL compared to other RTLs, A–1 to A–16
fwrite function, 2–6, REF–132

G

gcvt function, 3–9, REF–134
getc function, REF–136
getch macro, REF–137
getchar macro, REF–138
getcwd function, REF–139
getegid function, REF–141
getenv function, REF–142
geteuid function, REF–144
getgid function, REF–145
gethostbyaddr socket routine, D–65
gethostbyname socket routine, D–67
gethostent socket routine, D–68
gethostname socket routine, D–69
getname function, REF–146, REF–235
getnetbyaddr socket routine, D–71
getnetbyname socket routine, D–73
getnetent socket routine, D–75
getpeername socket routine, D–54
getpid function, REF–148
getppid function, REF–149
gets function, REF–104, REF–150
getsockname socket routine, D–56
getsockopt socket routine, D–58
getstr macro, REF–151
getuid function, REF–152
getw function, REF–153
getyx macro, REF–154
gmtime function, REF–155
gsignal function, REF–156

H

htonl socket routine, D–76
htons socket routine, D–78
HUGE_VAL constant, REF–182
hypot function, REF–159

I

inch macro, REF–160
#include
  preprocessor directive, 1–5
inet_addr socket routine, D–80
inet_lnaof socket routine, D–82
inet_makeaddr socket routine, D–83
inet_netof socket routine, D–84
inet_network socket routine, D–85
inet_ntoa socket routine, D–87
initscr function, REF–161
Input and output (I/O), 1–10 to 1–17
  conversion specifications, 2–7 to 2–13
  Record Management Services (RMS), 1–10
  Standard, 1–10
  stream access
    in VAX C, 1–15
  UNIX, 1–10
  VMS system services, 1–10
insch macro, REF–162
insertln macro, REF–163
insstr macro, 6–1, REF–164
Internet
  application programs, D–1
  protocols, D–8
Interprocess communication, 5–2
isalnum macro, REF–165
isalpha macro, REF–166
isapipe function, REF–167
isascii macro, REF–168
isatty function, REF–169
iscntrl macro, REF–170
isdigit macro, REF–171
isgraph macro, REF–172
islower macro, REF–173
isprint macro, REF–174
ispunct macro, REF–175
isspace macro, REF–176
isupper macro, REF–177
isxdigit macro, REF–178

K

kill function, REF–179, REF–230

L

labs function, REF–181
ldexp function, REF–182
ldiv function, REF–183
leaveok macro, REF–184
LIB$ESTABLISH routine, REF–352
Linker
  search libraries, 1–2
Linking Internet programs, D–3
listen socket routine, D–22
List-handling functions
  va_arg, REF–359
  va_count, REF–361
  va_end, REF–362
  va_start, REF–363
  va_start_1, REF–363
LNK$LIBRARY logical name, 1–2
localtime function, REF–185
log function, REF–187
log10 function, REF–187
longjmp function, REF–188, REF–352, REF–366
longname function, REF–190
lseek function, REF–191
lseek function, 1–14

M

Macro definitions, 1–4
Main function, 1–2
  main_program option, 1–2
malloc function, 8–2, REF–35, REF–193
Math functions, 7–1 to 7–4
  abs, REF–4
  acos, REF–7
  asin, REF–16
  atan, REF–19
  atan2, REF–20
  cabs, REF–32
  ceil, REF–34
  cos, REF–50
  cosh, REF–51
  div, REF–67
  errno values, 7–1
  exp, REF–89
  fabs, REF–90
  floor, REF–107
Math functions (Cont.)
  frexp, REF–120
  hypot, REF–159
  labs, REF–181
  ldexp, REF–182
  ldiv, REF–183
  log, REF–187
  log10, REF–187
  modf, REF–204
  pow, REF–238
  rand, REF–252
  sin, REF–287
  sinh, REF–288
  sqrt, REF–292
  srand, REF–293
  tan, REF–335
  tanh, REF–336
memchr function, REF–194
memcmp function, REF–195
memcpy function, REF–197
memmove function, REF–198
Memory allocation
  introduction to, 8–2
  program examples, 8–3
Memory allocation functions
  brk, REF–27
  calloc, REF–33
  cfree, REF–35
  free, REF–117
  malloc, REF–193
  realloc, REF–257
  sbrk, REF–263
  VAXC$CALLOC_OPT, REF–349
  VAXC$CFREE_OPT, REF–350
  VAXC$FREE_OPT, REF–354
  VAXC$MALLOC_OPT, REF–355
  VAXC$REALLOC_OPT, REF–357
Memory reallocation, REF–117
memset function, REF–200
mkdir function, REF–201
mktemp function, REF–203
modf function, REF–204
move macro, REF–205
mvaddch macro, REF–211
mvaddstr macro, REF–212
mvcur function, REF–207
mvdelch macro, REF–213
mvgetch macro, REF–214
mvgetstr macro, REF–215
mvinch macro, REF–216
mvinsch macro, REF–217

mvinsstr macro, 6–1, REF–218
mvwaddch macro, REF–211
mvwaddstr macro, REF–212
mvwdelch macro, REF–213
mvwgetch macro, REF–214
mvwgetstr macro, REF–215
mvwin function, REF–209
mvwinch macro, REF–216
mvwinsch macro, REF–217
mvwinsstr function, 6–1
mvwinsstr macro, REF–218

N

newwin function, REF–219
nice function, REF–221
nl macro, REF–222
nocrmode macro, REF–57
noecho macro, REF–70
nonl macro, 1–18, REF–222
noraw macro, REF–253
ntohl socket routine, D–89
ntohs socket routine, D–91

O

Occlusion, 6–5
open function, REF–68, REF–94, REF–223
overlay function, REF–25, REF–228
overwrite function, REF–25, REF–229

P

pause function, REF–230
perror function, 4–4, REF–231
pipe function, REF–68, REF–94, REF–233
Portability concerns, 1–12
  arguments to **mkdir**, REF–201
  _exit function, REF–88
  gsignal function, REF–157
  longname function, REF–190
  memory deallocation, REF–35, REF–117
  mvcur function, 6–14
  mv[w]insstr macros, REF–218
  [no]nl macros, REF–222
  radix conversion characters, 2–9
  raise function, REF–251
  setgid function, REF–272
  setuid function, REF–275
  socket routines, D–1
Portability concerns (Cont.)
  specific
    list of, 1–18 to 1–22
  ssignal function, REF–296
  ttyname function, REF–345
  UNIX file specifications, 1–8
    ambiguity of, 1–9
  variable-length argument lists, 3–11
  VAX C RTL compared to other RTLs, A–1 to A–16
  va_start_1 function, REF–363
  vfork versus **fork** function, REF–365
  [w]clrattr macro and function, REF–46
  [w]insstr macro and function, REF–164
  [w]setattr macro and function, REF–269
pow function, REF–238
Predefined variable
  ERR, 6–13
Predefined variables and constants, 6–12
printf function, REF–240
printw function, REF–241
Process permanent files, 2–13
Protocols
  Internet, D–8
putc function, REF–243
putchar function, REF–244
puts function, REF–245
putw function, REF–246

Q

qsort function, REF–247
Quotas
  affecting RTL, 5–1, 5–3, 5–5

R

raise function, REF–179, REF–249
rand function, REF–252
raw macro, REF–253
read function, REF–255
read socket routine, D–24
realloc function, 8–2, REF–257
Record attributes
  RMS
    VAX C handling of, 1–15
Record Management Services (RMS)
  file organization, 1–13
  in VAX C programs, 1–10
  overview of, 1–13 to 1–17
  record formats, 1–14

Record Management Services (RMS) (Cont.)
stream access
in VAX C, 1–15
recvfrom socket routine, D–28
recvmsg socket routine, D–31
recv socket routine, D–26
refresh macro, REF–42, REF–259
remove function, REF–63, REF–260
rename function, REF–261
rewind function, REF–262

S

sbrk function, 8–2, REF–263
scanf function, REF–264
scanw macro, REF–265
Screen management
Curses
See Curses
scroll function, REF–267
scrollok macro, REF–268
select socket routine, D–34
sendmsg socket routine, D–39
send socket routine, D–37
sendto socket routine, D–42
setattr macro, 6–1, REF–269
setbuf function, REF–271
setgid function, REF–272
setjmp function, REF–188, REF–273, REF–352, REF–366
setsockopt socket routine, D–60
setuid, REF–275
setvbuf function, REF–271, REF–276
Shared Image
VAX C RTL, 1–3
shutdown socket routine, D–45
sigblock function, REF–278, REF–281
Signal definition module, 4–5
signal function, REF–279, REF–296
Signal handling, 4–5
Signal-handling functions
alarm, REF–12
gsignal, REF–156
kill, REF–179
longjmp, REF–188
pause, REF–230
program examples, 4–7
raise, REF–249
setjmp, REF–273
sigblock, REF–278
signal, REF–279
Signal-handling functions (Cont.)
sigpause, REF–281
sigsetmask, REF–282
sigstack, REF–283
sigvec, REF–285
sleep, REF–289
ssignal, REF–296
VAXC$ESTABLISH, REF–352
Signals, 4–6
sigpause function, REF–281
sigsetmask function, REF–282
sigstack function, REF–283
sigvec function, 4–5, REF–285
sin function, REF–287
sinh function, REF–288
sleep function, REF–289
Socket routines
accept, D–15
auxiliary communication routines, D–53
basic communication routines, D–14
bind, D–17
close, D–19
communication support routines, D–63
connect, D–20
gethostbyaddr, D–65
gethostbyname, D–67
gethostent, D–68
gethostname, D–69
getnetbyaddr, D–71
getnetbyname, D–73
getnetent, D–75
getpeername, D–54
getsockname, D–56
getsockopt, D–58
htonl, D–76
htons, D–78
inet_addr, D–80
inet_lnaof, D–82
inet_makeaddr, D–83
inet_netof, D–84
inet_network, D–85
inet_ntoa, D–87
introduction, D–1
listen, D–22
ntohl, D–89
ntohs, D–91
porting considerations, D–1
programming examples, D–94
read, D–24
recv, D–26
recvfrom, D–28

Socket routines (Cont.)
recvmsg, D–31
select, D–34
send, D–37
sendmsg, D–39
sendto, D–42
setsockopt, D–60
shutdown, D–45
socket, D–47
vaxc$get_sdc, D–93
VAX C structures, D–3
write, D–51
socket socket routine, D–47
Specification delimiters
VMS and UNIX, 1–8
sprintf function, REF–290
sqrt function, REF–292
srand function, REF–293
sscanf function, REF–294
ssignal function, REF–156, REF–157, REF–249, REF–296
Standard I/O, 1–10
introduction to, 2–1
program example, 2–16
Standard I/O functions
clearerr, REF–41
delete, REF–63, REF–260
fclose, REF–91
fdopen, REF–94
feof, REF–96
ferror, REF–97
fflush, REF–98
fgetc, REF–99
fgetname, REF–100
fgets, REF–104
fopen, REF–109
fprintf, REF–111
fputc, REF–113
fputs, REF–114
fread, REF–115
freopen, REF–118
fscanf, REF–122
fseek, REF–124
ftell, REF–130
fwrite, REF–132
getc, REF–136
getw, REF–153
mktemp, REF–203
putc, REF–243
putw, REF–246
rewind, REF–262
Standard I/O functions (Cont.)
setbuf, REF–271
sprintf, REF–290
sscanf, REF–294
tmpfile, REF–339
tmpnam, REF–340
ungetc, REF–347
standend macro, REF–298
standout macro, REF–299
stat function, REF–300
stderr, 2–14, REF–98, REF–118, REF–231, REF–235
stdin, 2–14, REF–118, REF–235, REF–264
stdio
definition module, 1–5, 2–14
stdout, 2–14, REF–118, REF–235, REF–240, REF–244, REF–245
stdscr window, 6–5
strcat function, REF–303
strchr function, REF–194, REF–305
strcmp function, REF–195, REF–307
strcmpn function, 1–19
strcpy function, REF–197, REF–308
strcpyn function, 1–19
strcspn function, REF–309
Stream
access by VAX C, 1–15
files, 2–1
I/O
VAX C handling of, 1–16
Stream files
sharing, 2–6
strerror function, REF–310
String-handling functions, 3–11 to 3–12
atof, REF–23
atoi, REF–24
atol, REF–24
memchr, REF–194
memcmp, REF–195
memcpy, REF–197
memmove, REF–198
memset, REF–200
program examples, 3–12
strcat, REF–303
strchr, REF–305
strcmp, REF–307
strcpy, REF–308
strcspn, REF–309
strlen, REF–312
strncat, REF–313
strncmp, REF–314
strncpy, REF–316

String-handling functions (Cont.)
- **strpbrk**, REF–318
- **strrchr**, REF–319
- **strspn**, REF–320
- **strtok**, REF–325
- **strtol**, REF–327
- **strtoul**, REF–329

strlen function, REF–312
strncat function, REF–313
strncmp function, REF–314
strncpy function, REF–316
strpbrk function, REF–318
strrchr function, REF–319
strspn function, REF–320
strstr function, REF–321
strtod function, REF–23, REF–323
strtok function, REF–325
strtol function, REF–24, REF–327
strtoul function, REF–329
Structures
- use with Socket routines, D–3

Subprocess, 5–1 to 5–15
- executing image
  - with exec functions, 5–4
- implementation of, 5–2
- introduction to, 5–1
- program examples, 5–6 to 5–15
- sharing data with **pipe**, 5–6, REF–233
- synchronization with **wait**, 5–6

Subprocess functions
- **execl**, REF–75
- **execle**, REF–77
- **execlp**, REF–80
- **execv**, REF–82
- **execve**, REF–84
- **execvp**, REF–86
- **pipe**, REF–233
- **vfork**, REF–365
- **wait**, REF–372

Substitution
- macro, 1–4

subwin function, REF–331
Synchronizing processes, 5–6
Syntax
- of VAX C RTL functions, 1–6

SYS$ERROR, 2–14
SYS$INPUT, 2–14
SYS$OUTPUT, 2–14
SYS$WAKE, REF–12, REF–230
system function, REF–333
System functions, 9–1 to 9–7

System functions (Cont.)
- **asctime**, REF–14
- **assert**, REF–17
- **atexit**, REF–21
- **bsearch**, REF–29
- **chdir**, REF–36
- **chmod**, REF–37
- **chown**, REF–39
- **clock**, REF–43
- **ctermid**, REF–59
- **ctime**, REF–60
- **cuserid**, REF–61
- **difftime**, REF–66
- **fmod**, REF–108
- **ftime**, REF–131
- **getcwd**, REF–139
- **getegid**, REF–141
- **getenv**, REF–142
- **geteuid**, REF–144
- **getgid**, REF–145
- **getpid**, REF–148
- **getppid**, REF–149
- **getuid**, REF–152
- **gmtime**, REF–155
- introduction to, 9–3
- **localtime**, REF–185
- **memset**, REF–200
- **mkdir**, REF–201
- **nice**, REF–221
- program examples, 9–3
- **qsort**, REF–247
- **remove**, REF–63, REF–260
- **rename**, REF–261
- **setgid**, REF–272
- **setuid** function, REF–275
- **setvbuf**, REF–271, REF–276
- **strtod**, REF–323
- **strtok**, REF–325
- **system**, REF–333
- **time**, REF–337
- **times**, REF–338
- **umask**, REF–346
- **vfprintf**, REF–367
- **vprintf**, REF–369
- **vsprintf**, REF–371

T

tan function, REF–335
tanh function, REF–336

Terminal I/O
program examples, 2–14 to 2–19
Terminal I/O functions
gets, REF–150
printf, REF–240
putchar, REF–244
puts, REF–245
scanf, REF–264
Terminal I/O macros
getchar, REF–138
Text substitution, 1–4
See also, Substitution
time function, REF–337
times function, REF–338
tmpfile function, REF–339
tmpnam function, REF–340
toascii macro, 3–9, REF–341
_tolower macro, 3–9, REF–342
tolower function, 3–9, REF–342
touchwin function, REF–343
toupper function, 3–9, REF–344
_toupper macro, 3–9, REF–344
TRUE identifier, 1–5
ttyname function, REF–345

U

umask function, REF–346
umask value, 5–4
ungetc function, REF–347
UNIX
file specifications of, 1–8 to 1–10
compared to VMS, 1–8
Run-Time Library, 1–8
use with VAX C RTL, 1–8 to 1–10
UNIX I/O, 1–10
file descriptors, 2–5
functions
program example, 2–18
UNIX I/O functions
close, REF–44
creat, REF–52
dup, REF–68
dup2, REF–68
fileno, REF–106
fstat, REF–127
getname, REF–146
isapipe, REF–167
isatty, REF–169
lseek, REF–191
open, REF–223
UNIX I/O functions (Cont.)
read, REF–255
stat, REF–300
ttyname, REF–345
write, REF–374
unlink function, 1–18

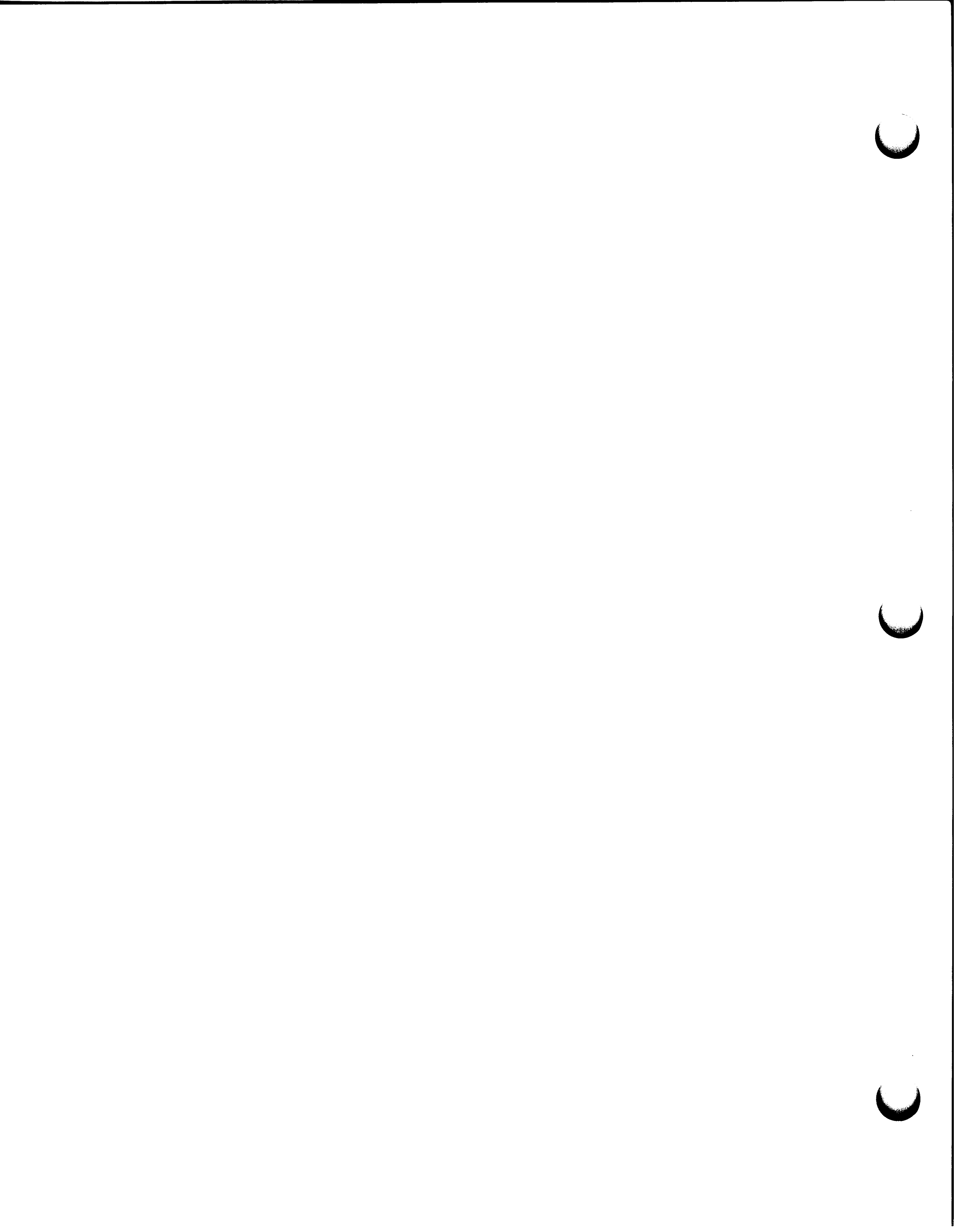
V

varargs
definition module, 3–11
Variable-length argument lists, 3–11
VAXC$CALLOC_OPT function, REF–349
VAXC$CFREE_OPT function, REF–350
VAXC$CRTL_INIT function, REF–346, REF–351
vaxc$errno variable, 4–5
VAXC$ESTABLISH function, REF–189, REF–274, REF–352
VAXC$EXECMBX, 5–4
VAXC$FREE_OPT function, REF–354
vaxc$get_sdc socket routine, D–93
VAXC$MALLOC_OPT function, REF–355
VAXC$REALLOC_OPT function, REF–357
VAXCDEF.TLB system library, 1–6
VAX C Run-Time Library (RTL)
as shared images, 1–3
compared to other RTLs, A–1 to A–16
Curses functions and macros, 6–1
definition modules, 1–7, C–1
entry points, B–6 to B–17
I/O, 1–10 to 1–17
VAX C handling of, 1–15 to 1–17
interpreting syntax, 1–6
introduction to, 1–1 to 1–22
main function, 1–2
portability concerns, 1–12
preprocessor directive, 1–7
procedures called by VAX C, B–17
run-time modules, B–1 to B–6
specific portability concerns, 1–18 to 1–22
stream I/O, 1–15
va_arg macro, REF–359
va_count macro, REF–361
va_end macro, REF–362
va_start function, REF–363
va_start_1 function, REF–363
vfork function, REF–235, REF–365
vfprintf function, REF–367
VMS system services
in VAX C programs, 1–10
VMS/ULTRIX Connection product, D–1

vprintf function, REF–369
vsprintf function, REF–371

W

waddch function, REF–8
waddstr function, REF–10
wait function, REF–372
wclear function, REF–40
wclrattr function, 6–1, REF–46
wclrtobot function, REF–48
wclrtoeol function, REF–49
wdelch function, REF–62
wdeleteln function, REF–64
werase function, REF–74
wgetch function, REF–70, REF–137
wgetstr function, REF–70, REF–151
winch function, REF–160
winsch function, REF–162
winsertln function, REF–163
winsstr function, 6–1, REF–164
wmove function, REF–205
wprintw function, REF–241
wrapok macro, REF–373
wrefresh function, REF–259
write function, REF–236, REF–374
write socket routine, D–51
wscanw function, REF–265
wsetattr function, 6–1, REF–269
wstandend function, REF–298
wstandout function, REF–299



How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

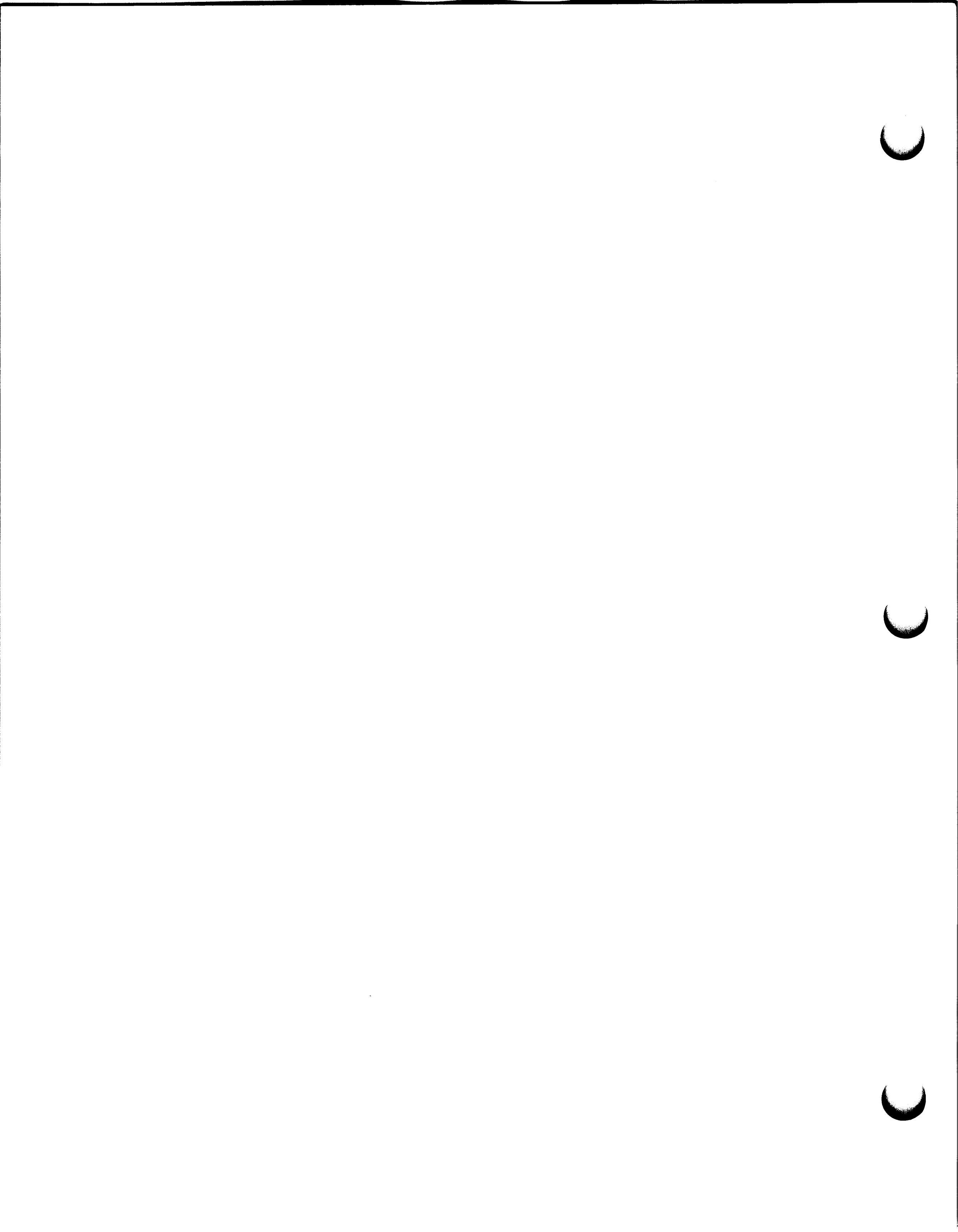
Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).



Reader's Comments

VAX C Run-Time Library Reference Manual
AA–JP84D–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:

Page Description

_____ ______________________________

_____ ______________________________

_____ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** _____ of the software this manual describes.

Name/Title ______________________________ Dept. ______________

Company ______________________________ Date ______________

Mailing Address ______________________________

______________________________ Phone ______________

-- Do Not Tear - Fold Here and Tape

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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

VAX C Run-Time Library Reference Manual
AA–JP84D–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:

Page Description

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ____________

Company ______________________________ Date ____________

Mailing Address ______________________________

______________________________ Phone ____________

— 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here

Cut Along Dotted Line