PDP–11 C

digital

Run–Time Library Reference Manual

Order Number: AA–NA45B–TC

94-003|049|15

PDP–11 C
Run-Time Library Reference Manual

Order Number: AA–NA45B–TC

November 1990

This manual describes the functions and macros in the PDP–11 C Run-Time Library.

Revision/Update Information:	This is a revised manual.
Operating System and Version:	Micro/RSX Version 4.3 or a higher version RSTS/E Version 10.0 or a higher version RSX–11M (mapped) Version 4.6 or a higher version RSX–11M–PLUS Version 4.3 or a higher version RT–11 Version 5.5 or a higher version VMS Version 5.3 or a higher version
Software Version:	PDP–11 C Version 1.1

digital equipment corporation
maynard, massachusetts

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

This document is available on CDROM.

CID #76788

Contents

Chapter 2 PDP–11 C Standard Input and Output

Chapter 3 Character-Handling Functions and Macros

Chapter 4 Localization Functions and Macros

Chapter 5 General Utility Functions

Chapter 6 Math Functions

Chapter 7 Using PDP–11 C with Record Management Services

Chapter 8 Using PDP–11 C with File Control Services

Reference Section

Examples

Figures

Tables

Preface

This manual provides reference information on the PDP–11 C Run-Time Library functions and macros that provide input/output, character and string manipulation, mathematical functionalities, error detection, file creation, and system access. PDP–11 C was developed in compliance with the Draft Proposed American National Standard for Information Systems—Programming Language C.

Intended Audience

This manual is intended for both experienced and novice programmers who need reference information on the functions and macros found in the PDP–11 C Run-Time Library.

Document Structure

This manual describes the PDP–11 C Run-Time Library. It provides information about portability concerns between operating systems and categorical descriptions of the functions and macros. This manual has ten chapters, a reference section, and two appendixes. They are as follows:

- Chapter 1 provides an overview of the PDP–11 C Standard Libraries.
- Chapter 2 describes the PDP–11 C Standard I/O functions and macros.
- Chapter 3 describes the character-handling functions and macros.
- Chapter 4 describes the localization functions and macros.
- Chapter 5 describes string conversion, pseudorandom sequence generation, memory management, environmental communication, search and sort, integer arithmetic, and multibyte character and string functions.

- Chapter 6 describes the math functions.
- Chapter 7 describes how to use PDP–11 C programs with Record Management Services (RMS).
- Chapter 8 describes how to use PDP–11 C with File Control Services (FCS).
- Chapter 9 describes operating systems services and system directives.
- Chapter 10 describes how to use PDP–11 C with other PDP–11 languages.
- The Reference Section describes alphabetically the functions and macros contained in the PDP–11 C Run-Time Library.
- Appendix A describes compatibility issues between the PDP–11 C and VAX C languages.
- Appendix B provides a description of the PDP–11 C modules and the PDP run-time modules used in this implementation.

Associated Documents

You may find the following documents useful when programming in the PDP–11 C language:

- *Guide to PDP–11 C*—For programmers who need additional information on using the PDP–11 C language.
- *PDP–11 C Installation Guide*—For system programmers who install the PDP–11 C software.
- *The C Programming Language*[1]—For those who need a more intensive tutorial than that provided in the *Guide to PDP–11 C*.

  PDP–11 C contains features and enhancements to the C language as it is defined in *The C Programming Language*. Therefore, the *Guide to PDP–11 C* should be used for a full description of PDP–11 C.

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, second edition (Englewood Cliffs, New Jersey: Prentice-Hall, 1988).

Conventions

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.
Color	Color is used to show user input. For online versions, user input is shown in bold.
. . .	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.
. . .	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.
[ ]	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 file specification or when used to delimit the dimensions of a multidimensional array in PDP–11 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.
Δ	A delta symbol is used in some contexts to indicate a single ASCII space character.
boldface	Boldface type identifies language keywords and the names of PDP–11 C Run-Time Library functions.
italic	Italics are used to identify variable names.

Chapter 1

PDP–11 C Standard Libraries

This chapter describes the PDP–11 C Standard Library functions, which includes those functions specified by the ANSI Standard, as well as some extensions to the PDP–11 C language.

To use a library function, the PDP–11 C source program should use a **#include** statement to include the appropriate **header file** that defines the function. A header file contains a set of definitions or declarations of related functions, types, and macros. To include a header file, use the **#include** preprocessor directive, which generally appears at the beginning of the program in the following format:

```
#include <file-name.h>
```

See the *Guide to PDP–11 C* for more information on the **#include** directive.

The name of a header file is file-name.h. Table 1–1 lists and briefly describes the PDP–11 C Standard Library header files.

NOTE

All PDP–11 C header files are source files.

Table 1–1: Standard Library Header Files

Header File	Purpose
assert.h	Defines the **assert** macro that is used for diagnostics.
ctype.h	Defines the functions used for testing and mapping characters.

(continued on next page)

Table 1–1 (Cont.): Standard Library Header Files

Header File	Purpose
errno.h	Defines the error-reporting macros.
float.h	Defines the macros that expand to various limits and parameters.
limits.h	Defines the macros that expand to various limits and parameters.
locale.h	Defines the functions, macros, and one type used for setting locale-dependent formatting and collating items.
math.h	Declares the functions and macros used for mathematical computations.
setjmp.h	Defines the macro and declares the function for bypassing the normal function call mechanism.
signal.h	Declares a type and the functions and defines the macros that report conditions during program execution.
stdarg.h	Declares a type and defines the macros used by a called function while going through a list of arguments whose numbers and types are not known.
stddef.h	Declares the types and defines macros for common definitions.
stdio.h	Declares the types, macros, and functions for standard input and output.
stdlib.h	Declares the types and functions used by the general utility functions.
string.h	Declares the type and the functions and defines the macro used for manipulating arrays of characters.
time.h	Defines the macros and declares the functions used for time manipulation.

1.1 The <assert.h> Header File

The <assert.h> header file defines the macro **assert**. The macro **NDEBUG** may be defined as a macro name in the source file before the <assert.h> file is included.

The **assert** macro puts diagnostics into programs. If the argument given to **assert** evaluates to false (0), the error status of the failed call is written, using the implementation-defined format, on the standard error file. Then, the **abort** function is called.

The format for the message output by the **assert** macro is:

```
assert error:  expression = <exp>, in file
<file>, at line <line>
```

In this message, <exp> is the text of the argument to **assert**, <file> is the value of the __FILE__ preprocessing macro, and <line> is the value of the __LINE__ preprocessing macro.

1.2 The <ctype.h> Header File

The <ctype.h> header file declares the functions and macros used for testing and mapping characters. These functions and macros are divided into two classes: character-testing and character case-mapping. See Table 3–1 for a list of the functions and macros declared in the <ctype.h> header file.

Character-Testing Functions and Macros

Character-testing functions take an argument of type **int**. The input value of the character-testing macro must be either the value defined as EOF or a value between 0 and 255. If the value is outside that range, the value returned by the character-testing macro is undefined.

Character-testing macros are defined by including **#include** <ctype.h> in a source file. When the <ctype.h> header file is included, the macro form of character-testing and mapping is used. To call the function form of the character-testing functions, include the header file and use the **#undef** directive to undefine the macro form.

Although character-testing macros are available as functions, it is recommended that the macro versions be used because they execute much faster. However, for the locale functions to work properly, the function form must be used.

Character Case-Mapping Functions and Macros

Character case-mapping functions are defined by putting **#include** <ctype.h> in a source file. The character mapping functions take an argument of type **int**. The input value must be either the value defined as EOF or a value between 0 and 255. If the value is outside that range, the value returned for either the character-mapping function or macro is undefined.

1.3 The <errno.h> Header File

The <errno.h> header file declares the modifiable lvalue, *errno*. At program start-up, *errno* is initialized to zero.

Many Standard Library functions deposit a nonzero value in *errno* when an error occurs during the execution of the function. If a program deposits a zero in *errno* before calling a Standard Library function, *errno* can be checked after the function completes for a zero value to determine if the function completed correctly. The lvalue *errno* contains a zero value if the function has completed correctly; otherwise, it contains a nonzero value indicating that an error has occurred.

The <errno.h> header file also defines a number of macros which define values that may be placed into *errno* by Standard Library functions.

1.4 The <float.h> and <limits.h> Header Files

The <float.h> and <limits.h> header files define a number of macros that expand to various limits and parameters. The size and a brief description of each macro defined by <limits.h> are listed in Table 1–2.

Table 1–2: Sizes of Integral Types

Macro	Size	Purpose
CHAR_BIT	8	Number of bits for smallest object that is not a bit-field.
CHAR_MAX	+127	Maximum value of an object of type **char**.
CHAR_MIN	–128	Minimum value of an object of type **char**.
INT_MAX	+32767	Maximum value of an object of type **int**.
INT_MIN	–32768	Minimum value of an object of type **int**.
LONG_MAX	+2147483647	Maximum value of an object of type **long int**.

(continued on next page)

Table 1–2 (Cont.): Sizes of Integral Types

Macro	Size	Purpose
LONG_MIN	–2147483648	Minimum value of an object of type **long int**.
MB_LEN_MAX	1	Maximum number of bytes in a multibyte character.
SCHAR_MAX	+127	Maximum value of an object of type **signed char**.
SCHAR_MIN	–128	Minimum value of an object of type **signed char**.
SHRT_MAX	+32767	Maximum value of an object of type **short int**.
SHRT_MIN	–32768	Minimum value of an object of type **short int**.
UCHAR_MAX	255U	Maximum value of an object of type **unsigned char**.
UINT_MAX	65535U	Maximum value of an object of type **unsigned int**.
ULONG_MAX	4294967295U	Maximum value of an object of type **unsigned long int**.
USHRT_MAX	65535U	Maximum value of an object of type **unsigned short int**.

The characteristics of floating types describe a representation of floating-point numbers and values that provide information about floating-point arithmetic. Table 1–3 lists the macros defined by the <float.h> header file, as well as a brief description of each macro.

Table 1–3: Characteristics of Floating Types

Macro	Characteristic	Purpose
DBL_DIG	16	Number of decimal digits.
FLT_DIG	6	
LDBL_DIG	16	

(continued on next page)

Table 1–3 (Cont.): Characteristics of Floating Types

Macro	Characteristic	Purpose
DBL_EPSILON FLT_EPSILON LDBL_EPSILON	1.39E–17 6E–8 1.39E–17	The difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type.†
DBL_MANT_DIG FLT_MANT_DIG LDBL_MANT_DIG	56 24 56	Number of decimal digits.
DBL_MAX FLT_MAX LDBL_MAX	1.7E38 1.7E38 1.7E38	The maximum representable finite floating-point number.†
DBL_MAX_EXP FLT_MAX_EXP LDBL_MAX_EXP	127 127 127	The maximum integer such that **FLT_RADIX** raised to that power minus 1 is a representable finite floating-point number.
DBL_MAX_10_EXP FLT_MAX_10_EXP LDBL_MAX_10_EXP	38 38 38	The maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.
DBL_MIN FLT_MIN LDBL_MIN	2.94E–39 2.94E–39 2.94E–39	The minimum normalized positive floating-point number.†

†Rounded to three significant digits.

(continued on next page)

Table 1–3 (Cont.): Characteristics of Floating Types

Macro	Characteristic	Purpose
DBL_MIN_EXP FLT_MIN_EXP LDBL_MIN_EXP	–127 –127 –127	The minimum negative integer such that **FLT_RADIX** raised to that power minus 1 is a normalized floating-point number.
DBL_MIN_10_EXP FLT_MIN_10_EXP LDBL_MIN_10_EXP	–38 –38 –38	The minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.
FLT_RADIX FLT_ROUNDS	2 –1	Radix of exponent. The rounding mode of floating-point addition is indeterminable.

1.5 The <locale.h> Header File

The <locale.h> header file declares two functions, **setlocale** and **localeconv**, and one type, **struct lconv**, and defines several macros used for setting the character set, collating sequence, monetary format, decimal-point character, and date and time formats. For more information, refer to Chapter 4.

1.6 The <math.h> Header File

The <math.h> header file declares the mathematical functions and the macro **HUGE_VAL** which is the largest representable double precision value.

For each function, a domain error occurs if the input argument is outside the domain of the mathematical function. The function returns a value of 0 and places the value of the macro **EDOM** in *errno*.

The value assigned to **HUGE_VAL** is equal to the value assigned to the macro **DBL_MAX**.

A range error occurs if the result of the function cannot be represented as a double value. The value of the macro **HUGE_VAL** is returned and the value of *errno* is set to the value of the macro **ERANGE**.

If there is an underflow error, the function returns zero and *errno* is set to the value of the macro **ERANGE**.

See Table 6–1 for a listing of the functions declared by the <math.h> header file.

1.7 The <setjmp.h> Header File

The <setjmp.h> header file declares the type **jmp_buf**, the **longjmp** function, and the **setjmp** macro which are used to bypass normal function returns and allow an immediate return from a nested function call.

The type **jmp_buf** is declared as an array of **int**.

The **setjmp** macro saves the current context of the function in a data area of type **jmp_buf** and returns a value of zero. A call to **setjmp** can only occur in the context of a test of *if, switch,* and loops, and then only in simple relational expressions.

The **longjmp** function restores the context saved by the **setjmp** macro. Control appears to transfer from the macro **setjmp** and returns a nonzero value.

1.8 The <signal.h> Header File

The <signal.h> header file declares the functions **raise**, **signal**, and **__sleep**, as well as the type and macros that handle various conditions that may be reported during the execution of a program; these conditions are referred to as signals. Table 1–4 lists the signal-handling macros and the conditions associated with them.

Table 1–4: Signal-Handling Conditions

Condition	Description
SIGABRT	Abnormal termination, such as is initiated by the **abort** function.

(continued on next page)

Table 1–4 (Cont.): Signal-Handling Conditions

Condition	Description
SIGFPE	An erroneous arithmetic operation, such as zero divide or an operation resulting in overflow.
SIGILL	Detection of an invalid function image, such as an illegal instruction.
SIGINT	Receipt of an interactive attention signal.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request sent to the program.
Action	**Description**
SIG_DFL	Default action to be taken.
SIG_IGN	Ignore the signal.
Return Value	**Description**
SIG_ERR	Indicates signal value cannot be honored.

1.9 The <stdarg.h> Header File

The <stdarg.h> header file declares the type, functions, and macros that are used for advancing through a list of arguments whose number and types are not known at compile time. Table 1–5 lists and briefly describes these macros. For more information, refer to the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

Table 1–5: Variable Argument Macros

Macro	Description
va_arg	Returns the next item in the argument list.
va_end	Finishes a function call using a variable argument list.
va_start	Initializes a variable to the beginning of the argument list.

Chapter 2 of the *Guide to PDP–11 C* provides an example on including the <stdarg.h> header file in a parameter list.

1.10 The <stddef.h> Header File

The <stddef.h> header file contains a number of type and macro definitions, many of which are implementation-defined. Table 1–6 lists the types and macros that are implementation-defined and the definitions assigned to them by PDP–11 C.

Table 1–6: Implementation-Defined Types and Macros

Type or Macro	Definition
NULL	((void *)0)
offsetof(type, member)	((size_t) (& (((type *) NULL)->member)))
ptrdiff_t	Type int
size_t	Type unsigned int
wchar_t	Type unsigned char

1.11 The <stdio.h> Header File

The <stdio.h> header file declares three types and several macros and functions that perform input and output. This includes writing to files, reading from files, opening and closing files, and maneuvering in files. For more information and a list of these types, functions, and macros, refer to Chapter 2.

1.12 The <stdlib.h> Header File

The <stdlib.h> header file defines several macros and declares four general utility types and several functions, including string conversion, memory management, environmental communication, string and sorting utility, and multibyte character and string functions. For a list of these macros, types, and functions, as well as more information, refer to Chapter 5.

1.13 The <string.h> Header File

The <string.h> header file declares several functions and one type, **size_t**. It also defines one macro, NULL, for use as a null pointer constant. Table 1–7 lists and briefly describes the copy, comparison, search, concatenation, and miscellaneous functions.

Table 1–7: String Functions

Copy	Description
memcpy, memmove	Copies a specified number of bytes from one object to another.
strcpy,strncpy	Copies all or part of one string into another.

Comparison	Description
memcmp	Compares two objects, byte by byte.
strcmp, strncmp	Compares two character strings and returns a negative, zero, or positive integer indicating that the values of the individual characters in the first string are less than, equal to, or greater than the values in the second string.
strcoll	Compares two character strings using the collating sequence of the current setting of the **LC_COLLATE** portion of the locale.
strxfrm	Transforms one string into another string according to the collating sequence established by the **setlocale** function.

Search	Description
memchr	Locates the first occurrence of the specified byte within the initial length of the object to be searched.
strchr, strrchr	Returns, respectively, the address of the first or last occurrence of a given character in a null-terminated string.

(continued on next page)

Table 1–7 (Cont.): String Functions

Search	Description
strcspn	Searches a string for a character in a specified set of characters.
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.
strstr	Locates the first occurrence of a sequence of characters in one string that matches the sequence of characters in another string.
strtok	Locates text tokens in a given string.

Concatenation	Description
strcat, strncat	Concatenates one string to the end of another string.

Miscellaneous	Description
memset	Sets a specified number of bytes in a given object to a given value.
strerror	Maps an error number to an error message string.
strlen	Returns the length of a string. The returned length does not include the terminating NUL character (\0).

For further information on the functions, refer to the Reference Section.

1.14 The <time.h> Header File

The <time.h> header file defines two macros, **CLOCKS_PER_SEC**, the value returned by the **clock** function, and **NULL**. It also declares four types and several functions for time manipulation.

The types are:

- **clock_t** and **time_t**, arithmetic types representing time
- **struct_tm**, which holds the components of calendar time referred to as *broken-down time*

- **size_t**, the unsigned **int** result of the operator **sizeof**.

The functions manipulate calendar time, which represents the current date according to the Gregorian calendar; local time, which represents calendar time expressed for a specific time zone; and Daylight Saving Time, which represents a temporary change for determining local time. Local time and Daylight Saving Time are implementation-defined.

Table 1–8 lists and briefly describes the date and time functions. For more information, refer to the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

Table 1–8: Date and Time Functions

Function	Description
asctime	Converts a broken-down time into a 26-character string.
clock	Determines the CPU time used since program execution.
ctime	Converts a time in seconds to an ASCII string.
difftime	Computes the difference in seconds between two specified times.
gmtime	Converts a given calendar time into time expressed as Coordinated Universal Time (UTC).
localtime	Converts a time expressed as numbers of seconds into hours, minutes, and seconds.
mktime	Converts time into a calendar time value.
strftime	Gives the time for the current locale.
time	Returns the elapsed time since 00:00:00, January 1, 1970, in seconds.

Chapter 2

PDP–11 C Standard Input and Output

This chapter describes the I/O capabilities of the PDP–11 C Standard Libraries. Table 2–1 lists all the I/O functions and macros found in the PDP–11 C Run-Time Library. These functions and macros are defined in the <stdio.h> header file. For more detailed information, see the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

Table 2–1: I/O Macros and Functions

Macro or Function	Purpose
Macros	
BUFSIZ	Size of the buffer used by **setbuf** function.
EOF	A value indicating end-of-file.
_IOFBF, _IOLBF, _IONBF	Buffer mode used as third argument to **setvbuf** function.
L_tmpnam	Size of an array large enough to hold a temporary file-name string generated by the **tmpnam** function.
FOPEN_MAX	Maximum number of files that can be opened simultaneously.
FILENAME_MAX	Maximum length for a file name.

(continued on next page)

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

Macro or Function	Purpose
Macros	
SEEK_SET, SEEK_CUR, SEEK_END	Third argument to the **fseek** function.
TMP_MAX	Minimum number of unique file names generated by the **tmpnam** function.
Standard I/O—Opening and Closing Files	
fclose	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.
fopen	Opens a file and returns a pointer to the file structure.
freopen	Substitutes the file, named by a file specification, for the open file addressed by a file pointer.
Standard I/O—Reading from Files	
fgetc	Returns a character from a specified file.
fgets	Reads a line from a specified file and stores the characters in a string pointed to by an argument.
fread	Reads a specified number of items from a file.
getc	Returns characters from a specified file.
sscanf	Performs formatted input from a string.
ungetc	Pushes back a character into an input stream and leaves the stream positioned before the character.
Standard I/O—Writing to Files	
fprintf	Performs formatted output to a specified file.
fputc	Writes a character to a specified file.

(continued on next page)

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

Macro or Function	Purpose
Standard I/O—Writing to Files	
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.
putc	Writes a character to a specified file.
sprintf	Performs formatted output to a string.
Standard I/O—Maneuvering in Files	
fflush	Writes any buffered information to the specified file.
fgetpos	Finds the current file position indicator for a stream.
fseek	Positions the file to the specified offset in the file.
fsetpos	Sets the current file position indicator of a stream.
ftell	Returns the current offset to the specified stream file.
rewind	Sets the current file position to the beginning of the file.
Standard I/O—Formatted Output	
vfprintf	Prints formatted output to a file based on an argument list.
vprintf	Prints formatted output to *stdout* based on an argument list.
vsprintf	Prints formatted output to a string based on an argument list.

(continued on next page)

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

Macro or Function	Purpose
Standard I/O—Additional Standard I/O Functions and Macros	
clearerr	Resets the error and end-of-file indicators 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.
perror	Prints a line to the standard error stream which consists of a user-passed string, colon, or space and the error message text that corresponds to the current value of the *errno* expression.
remove	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 unique 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 (**stdin**).

(continued on next page)

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

Macro or Function	Purpose
Terminal I/O—Writing to Files	
printf	Performs formatted output from the standard output (*stdout*) of a stream.
putchar	Writes a single character to the standard output (*stdout*) and returns the character.
puts	Writes a character string to the standard output (*stdout*) followed by a newline.
Accessing File Information	
__fbuf	Returns current buffer length associated with a file pointer.
__fger	Returns low-level error code that is associated with a previously called file operation.
__fgnm, fgetname	Returns a pointer to a file specification associated with a file variable.
__flun	Returns the logical unit number associated with a file pointer.
__frec	Returns the current record length associated with a file pointer.

2.1 Streams and Files

The PDP–11 C language refers to the logical data path upon which standard input/output occurs as a stream. A stream is a path from the program to and from the data stored in a file. Two types of streams are used in PDP–11 C: text and binary.

2.1.1 Text and Binary Streams

The choice between text and binary streams is made when the user program opens the file. Certain functions operate differently, depending on whether they are used with text or binary streams.

A text stream is an ordered sequence of characters composed into lines that allow a C program to create text files that are readable by other programs, especially text editors. Each line consists of zero or more characters plus a terminating newline character. A one-to-one correspondence between the characters in the text stream and those in the file is not necessary.

A binary stream maps data one-to-one with the data in the file. Although the newline character has meaning for binary streams, it must map to one character in the file.

2.1.2 Compatibility with VAX C

VAX C does not distinguish between text and binary streams; however:

- All files created by PDP–11 C are read by VAX C with no conversion.
- Files created by VAX C are read as binary stream files by PDP–11 C with no conversion. However, text files created by VAX C must be converted before they are read as text files on PDP–11 C.

For more information on PDP–11 C and VAX C compatibility, refer to Appendix A.

2.2 Streams and Operating Systems

Mapping from a PDP–11 C stream to a file system is dependent on the following:

- The operating system
- If the stream is text or binary
- If the file exists or is being created
- If the target of the stream is a physical device or a file on a supported file system (FCS or RMS)

The following sections describe how PDP–11 C maps text and binary streams to the file types on each operating system.

2.2.1 RSX Operating System and Text Files

On RSX operating systems, when a text stream is mapped to a file and the file is being created, PDP–11 C creates a sequential file with variable-length record format, implied carriage control record attributes, and no defined maximum record length.

PDP–11 C scans the output data for a newline character when placing the data to the output text file. All data up to but not including the newline character is put in the file as a file record. All data after the newline character becomes part of the next record. The newline character is never part of the file, but it is represented implicitly by the end of each record in the file.

When data is read from an external file, a record is read from the file and a newline character is appended to that data. PDP–11 C has a default maximum line length of 512 characters including one for the newline character. The PDP–11 C Standard Library places the first 511 characters in the file buffer. Additional characters are placed at the beginning of the file buffer where they form the characters of the next line in the text file.

The size of the internal buffer, and therefore line size, can be modified with the **setvbuf** function. The modified buffer size determines the maximum line length of the PDP–11 C program. If the buffer size is not modified by **setvbuf**, an error occurs when the program attempts to read a record larger than 511 bytes from a file.

Before an existing text file can be opened as a text stream, the file must be sequential and the record format must be variable length. It is not possible to open a relative or indexed file or a file with fixed-length records as a text stream using the PDP–11 C Standard I/O library.

If the defined maximum record size or longest record length is greater than 511 bytes, PDP–11 C allocates an internal buffer size equal to the defined size plus 1 byte for the newline character. If the defined maximum record size is less than 511 bytes and is opened for reading, PDP–11 C allocates storage space for the actual length of the record.

Table 2–2 shows the internal line size allocated when an existing file is opened as previously discussed.

Table 2–2: File Sizes

External Record Size[1]	New File, Read Only	Existing File, Write Only
<511 bytes	Actual record size	512 bytes
>=511 bytes	Actual record size	Actual record size
Unknown	512 bytes	512 bytes

[1]Defined by maximum record size and largest record length.

2.2.2 RSX File Attributes

Although PDP–11 C Standard I/O allows a program to create a sequential file with implicit carriage control, other record formats can be read and written when an existing file is opened using standard input/output.

Table 2–3 shows how PDP–11 C interprets different RSX record types on existing text streams.

Table 2–3: RSX Attributes and Behavior

Attribute	Behavior
Explicit carriage control	Input: Check for the <CR> <LF> sequence. If found, remove from input string and replace with newline character.
	Output: Replace newline character with <CR><LF> before output is performed.
FORTRAN input	If the control character is NUL, the record is not modified further.
	If the control character is 0, two newline characters are placed at the beginning of the record and a <CR> is placed after it.
	If the control character is 1, a <FF> is placed before the record and a <CR> is placed after it.
	If the control character is +, a <CR> is placed after the record.

(continued on next page)

Table 2–3 (Cont.): RSX Attributes and Behavior

Attribute	Behavior
	If the control character is $, a newline character is placed at the start of the record. For all other characters, a newline character is placed at the front of the record, and a <CR> is placed after the record.
FORTRAN output	Inverse to input mapping takes place.
Variable record format with fixed control area	Concatenate the fixed area to the front of the record. This is not supported by RMS or FCS.
Stream	Input: If the record does not end in <LF>, <FF>, or <VT>, a newline character is appended to the record. Output: Change the newline character to <LF>.
Mapped to a device (must be record-oriented device)	Input: Append the terminator to the input data. If the terminator was a <CR> or CTRL/Z, a newline character is appended. Termination characters are device-dependent. Output: Change newline characters to a <CR><LF> sequence.

2.2.3 RSX Operating System and Binary Files

When creating a new binary stream file on the RSX operating system, PDP–11 C creates a sequential file with a fixed record size of 512 bytes. The file has no record attributes.

All data is moved to and from the file in 512-byte increments unless the function **setvbuf** is used to change the internal buffer size or a device is being opened. The buffer size for an open device is the record size of the device.

The newline character is represented by a <LF> during output to all binary files. During input all <LF> characters are interpreted as newline characters.

PDP–11 C can open any file as a binary stream.

Any user-accessible device may be opened as a binary file.

2.2.4 RSTS/E Operating System and Stream Files

PDP–11 C uses a RSTS/E-native stream file as the system file to map to C streams.

2.2.5 RSTS/E Operating System and Text Files

PDP–11 C creates a RSTS/E-native stream file when it creates a text stream file. Newline characters are converted to <CR><LF> during output, and <CR><LF> is converted to a newline character during input.

Additional terminator characters are:

- <LF><CR><NUL> Translated to a newline character
- <LF> Passed unmodified
- <FF> Passed unmodified
- <ESC> Passed unmodified
- <VT> Passed unmodified

All null characters read from a RSTS/E-native file are ignored.

PDP–11 C opens a RSTS/E-native file when it opens an existing file as a text stream or when it opens a text stream on a device. There is no restriction on nonrecord-oriented devices. Additionally, all RMS files that can be read on RSX systems as text files can be opened and read as text files on the RSTS/E system.

2.2.6 RSTS/E Operating System and Binary Files

PDP–11 C creates a RSTS/E-native file when it creates a binary stream file. The newline character is represented by a <LF> character on output, and the <LF> character is converted to a newline character on input.

Besides supporting RSTS/E-native files as binary input files, PDP–11 C allows all RMS files with sequential organization and fixed-length records to be opened as a binary stream. Refer to Section 2.2.3 for further information on file behavior.

2.2.7 RT–11 Operating System and Stream Files

The RT–11 operating system supports only one file format. Although RT–11 has object, stream, save image, and other file *types*, there is no way of determining what the file type is by looking at the file or the data in it.

2.2.8 RT–11 Operating System and Text Files

On the RT–11 operating system any file can be opened as a text stream. PDP–11 C converts a <CR><LF> sequence to a newline character. All other characters, except NULL, pass unmodified. CTRL/Z denotes the end of a text file. All null characters are ignored.

2.2.9 RT–11 Operating System and Binary Files

All files can be opened as binary streams. PDP–11 C represents the newline character as a <LF> in the file. The <LF> is interpreted as a newline character during input. The end of the binary file is the physical end of the file.

2.3 The <stdio.h> Header

Table 2–1 lists the functions and macros which the <stdio.h> header file declares. For detailed descriptions of the Standard I/O functions, refer to the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

The <stdio.h> header file declares two types:

FILE
fpos_t

The PDP–11 C **FILE** type is a type capable of recording the information needed to control a stream. It is declared as an incomplete structure. Because only pointers to the object of type **FILE** are used by Standard Library I/O Functions, it is not necessary to declare the full contents of the **FILE** object. Access to key elements of this structure may be obtained by the **__fgnm**, **fger**, **__flun**, **__fbuf**, and **__frec** functions.

The PDP–11 C **fpos_t** type consists of four 16-bit words capable of recording the information needed to uniquely specify positions within a file.

The <stdio.h> header defines *stderr*, *stdin*, and *stdout*, which point to the FILE objects associated with the standard error stream, the standard input stream, and the standard output stream, respectively.

2.4 Conversion Specifications

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

```
int      x = 5;
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.

2.4.1 Converting Input Information

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

- 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. Conversion specifications must begin with the percent sign (%).

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 2–4 describes the conversion specifiers for formatted input.

Table 2–4: Conversion Specifiers for Formatted Input

Character	Meaning
d	Matches an optionally signed decimal integer. The corresponding argument is a pointer to an **int**.
i	Matches an optionally signed integer whose format is that of an integer constant. The corresponding argument is a pointer to an **int**.
o	Matches an optionally signed octal integer. The corresponding argument points to an unsigned **int**.
u	Matches an optionally signed decimal integer. The corresponding argument points to an unsigned **int**.
x, X	Matches an optionally signed hexadecimal integer. The corresponding argument points to an unsigned **int**.
e, E, f, g, G	Matches an optionally signed floating-point number. The corresponding argument points to a **float**.
s	Matches a sequence of non-white space characters. The corresponding argument points to an array of type **char** large enough to hold the input and a terminating NUL character.
[]	Matches a sequence of characters (scanlist) from a set of characters (scanset). The corresponding argument points to the initial **char** of an array large enough to hold the sequence of characters. The characters inside the brackets (scanlist) make up the scanset. However, if the left bracket is followed by a circumflex (^), then the scanset is all the characters that are not in the scanlist.
c	Matches a sequence of characters specified by the field width. If a field width is not given, then the width is 1. The corresponding argument points to an array of type **char** large enough to hold the input and a terminating NUL character.
p	Matches a sequence of characters representing a pointer. The corresponding argument points to a pointer to **void**.
n	No conversion. The corresponding argument is a pointer to an **int** into which is put the number of characters read from the input stream.
%	Matches a percent sign.

Refer to Table 2–5 for optional conversion modifiers for formatted input.

Table 2–5: Optional Conversion Modifiers

Modifier	Meaning
h	**Short int** for d, i, n.
	Unsigned **short int** for o, u, x.
l	**Long int** for d, i, n.
	Unsigned **long int** for o, u, x.
	Double for e, f, g.
L	**Long double** for e, f, g.
*	Suppress assignment.
number	A number used as the maximum field width.
[. . .]	Expects a string that is not delimited by white-space characters. The brackets enclose a set of characters (not a string). Ordinarily, this set (or "character class") is made up of the characters that comprise the string field. Any character not in the set will terminate the field. However, if the first (leftmost) character is a circumflex (^), then the set shows the characters that terminate the field. The corresponding argument must point to an array of characters.

Remarks

- The modifiers precede the conversion specification characters. For example, when the modification character l is added to the conversion specification character x, a **long** integer of the specified radix (lx) is expected.
- The delimiters of the input field can be changed with the bracket ([ ]) conversion specification. Otherwise, an input field is defined as a string of nonwhite-space characters. It extends either to the next white-space character or until the field width, if specified, is exhausted. The function reads across line 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 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)
  ```

  not

  ```
  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
  ```

  The format specification does not match the following:

  ```
  file d=5218
  ```

2.4.2 Converting Output Information

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

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

Table 2–6 describes the conversion specifiers for formatted output.

Table 2–6: Conversion Specifiers for Formatted Output

Character	Meaning
d, i	Converts to signed decimal in the format *[–]dddd*. The precision indicates the minimum number of digits to appear, with the default being 1 digit. Converting a zero value with a precision zero yields no characters.
o	Converts to unsigned octal in the format *dddd*.
u	Converts to unsigned decimal in the format *dddd* (giving a number in the range 0 to 65,535).
x, X	Converts to unsigned hexadecimal in the format *dddd* (without a leading 0x). An uppercase X causes the hexadecimal digits A–F to be printed in uppercase. A lowercase x causes those digits to be printed in lowercase.
f	Converts **float** or **double** to the format *[–]ddd.ddd*. The number of digits is specified by the precision (the default is 6). The precision does not determine the number of significant digits printed. If the precision is 0 and the # flag is not given, no decimal point characters appear.
e, E	Converts **float** or **double** to the format *[–]d.ddde±dd*. If no precision is given, the default is 6. If the precision is 0 and the # flag is not given, no decimal point characters 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.
g, G	Converts **float** or **double** to f or e format. The format depends on the value that is converted. If the exponent from the conversion is less than –4 or greater than or equal to the precision, then the e format is used. The fractional portion of the result has trailing zeros removed. A decimal-point does not appear if it is not followed by a digit.
c	Outputs an unsigned **char**.
s	Writes characters from an array of characters until a NUL character 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.
p	The argument is a pointer to **void**. The pointer is printed as an octal number of 7 digits, including a leading 0 character.
n	The argument points to an **int** where the number of output characters is placed. No conversion is performed.
%	Writes out the percent symbol. No conversion is performed.

You can use the characters listed in Table 2–7 between the percent sign (%) and the conversion character. These characters are optional; if specified, they must occur between the percent sign (%) and the conversion specifier.

Table 2–7: Optional Conversion Modifiers for Formatted Output

Modifier	Meaning
h	Indicates that a following d, i, o, u, x, or X specification corresponds to a **short int** or **unsigned short int** as appropriate.
l	Indicates that a following d, i, o, x, or X specification corresponds to a **long int** or **unsigned long int** as appropriate. In PDP–11 C, all **int** values are **short** by default.
L	Indicates that a following e, E, g, or G specification corresponds to a **long double**.
* (asterisk)	Is used to indicate the field width specification, the precision specification, or both. The field width or precision is given by an **int** argument. The arguments must appear in the following order preceding the argument to be converted: field width, precision, or both. A negative field width argument is interpreted as a "–" flag preceded by a positive field width. A negative precision argument is interpreted as no argument given.

Refer to Table 2–8 for descriptions of optional flag characters.

Table 2–8: Optional Conversion Flag Characters

Flag	Meaning
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 0s 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; 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.
- (hyphen)	Left-justify the converted output source in its field. If no hyphen is specified, the field is right-justified.
+	Indicates that the number prints with a sign.

(continued on next page)

Table 2–8 (Cont.): Optional Conversion Flag Characters

Flag	Meaning
space	A space is inserted following the first character of a signed conversion if there is no sign or if the conversion results in no characters. If there is a space and "+" sign, the space is ignored.
#	Alternate form of conversion of the result. For o conversion, it forces the first digit of the result to zero. For x and X conversion, it places 0x or 0X before a nonzero result. For e, E, f, g, and G conversions, the result contains a decimal point even when there are no digits following it. Normally, the only time a decimal point appears is when a digit follows it. For g and G conversions, any trailing zeros are not removed.
0	Leading 0s are used to pad the field width for d, i, o, u, x, X, e, E, f, g, and G conversions. Space padding is not normally performed. The 0 flag is ignored if the 0 and hyphen (-) appear. When a precision is given for d, i, o, u, x, and X conversions, the 0 flag is ignored.

2.5 The /CP Taskbuilder Switch

On the RSX operating system, programs that use Standard I/O functions must use the /CP taskbuilder switch when taskbuilding. This is because the memory management functions used by the Standard I/O run-time support routines require that the task must be built using the /CP taskbuilder switch. For further information, refer to the taskbuilder manual for the appropriate operating system.

2.6 Input/Output Support Package

PDP–11 C provides support routines which use RMS, FCS, and Native I/O to access files when using PDP–11 C Standard I/O functions. All PDP–11 C tasks that include any Standard I/O routines that do input or output include support for Native I/O. No user action is required to include this support.

The following table shows the I/O support, the operations supported, and the operating system on which they are used:

Operating System	Native I/O	RMS	FCS
RT	All operations	N/A	N/A
RSX[1]	To devices	To files	To files
RSTS/E	All operations	To files	N/A

[1]File deletion and renaming are done using FCS or RMS on RSX.

The following two sections describe the use of RMS and FCS for file input and output.

RMS for File Input/Output

The module $PRMXF must be explicitly included in the Task Builder .ODL file when tasks are built that use RMS to access files through Standard I/O functions. Also, an appropriate RMS .ODL must be referenced to include the proper RMS support. The following example shows how to build $PRMXF and RMS into a task:

```
          .ROOT USER
USER:     .FCTR SY:TSTREN-LB:[1,1]CFPURSX/LB:$PRMXF-RMSROT-LIBR,RMSALL
LIBR:     .FCTR LB:[1,1]:CFPURSX/LB
@LB:[1,1]RMS11S
                  .END
```

FCS for File Input/Output

The module $PFCXF must be explicitly included in the Task Builder .ODL file when tasks are built that use FCS to access files through standard I/O functions. The following example shows how to build $PFCXF into a task:

```
          .ROOT USER
USER:     .FCTR SY:TSTREN-LB:[1,1]CFPURSX/LB:$PFCXF-LIBR
LIBR:     .FCTR LB:[1,1]:CFPURSX/LB
          .END
```

When including the FCS support package on an RSX system, the $PFCXF module allocates enough FCS internal storage for three files requiring FCS support to be opened concurrently. Should a program require more than three FCS files to be opened, it will be necessary for the user to increase the size of the $$FSR1 psect. This can be done when the task is linked. For more information, see the *RSX-11M/M-PLUS and Micro/RSX I/O Operations Reference Manual*.

2.7 Reserving LUNs

When the PDP–11 C Run-Time Library opens a file, it allocates one of the LUNs available to it. By default, a maximum of eight files can be opened at once, as indicated by the FOPEN_MAX definition in the <stdio.h> header file.

It is possible for a task to open more than eight files at once by patching the symbol $NLUNS to the desired value. Note that stdin uses one LUN, while *stdout* and *stderr* share another. Therefore, if you wanted to have 11 user files at once, you need to patch the value 13 into $NLUNS. Do this by using the GBLPAT option of the RSX Task Builder or the SIPP utility on RT–11.

When a task is built, the task builder automatically assigns a number of LUNs to the task. One LUN is required for every file that the program has opened. The number of required LUNs is equal to the number of files opened at one time plus an additional LUN if standard output is redirected.

The format of the PDP–11 C Run-Time Library module that defines which LUNs are reserved is:

```
.TITLE $PRLUN
 $PRLUN::
.WORD 0                                    ;Number of "reserve words"
.END
```

The $PRLUN global symbol is the start of the LUN reservation table. The first word of the table is the number of words that follow in the table. No LUNs are reserved by default and the length of the table is zero.

Reserve words appearing in the table make up the bit vector. A bit position in the vector corresponds directly to a LUN number. For example, the first reserve word holds bits corresponding to LUNs 0 to 15, and the second reserve word holds bits corresponding to LUNs 16 to 31. Because no LUNs are reserved by default, there are no reserve words in the module.

PDP–11 C provides the user a way to reserve any LUN or LUNs by creating a MACRO file to replace the default MACRO file included in the task. LUNs are reserved at task build time. The following example shows how to reserve LUNs 4, 5, and 8 in a MACRO program:

```
.TITLE $PRLUN
 $PRLUN::
.WORD 1             ;Need only 1 reserve word
.WORD 460           ;Set bits 4, 5, and 8
.END
```

The following example shows how to reserve LUNs 4, 5, and 8 in a PDP–11 C program:

```
#pragma module "$PRLUN", "V01.01"

const short $PRLUN[2] ={
                        1,      /* Number of reserve words */
                        0460    /* Reserve LUNs 4,5, and 8 */
                       };
```

Reference can be made to this module in the task's .ODL file, through the Task Builder command line, or through the Linker command line.

2.8 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 <stdio.h>

int main ()

{
   double      val        = 123.3456e+3;
   char        c          = 'C';
   long 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:    <%9.4f>\n",     val);
   printf("%%9f:      <%9f>\n",       val);
   printf("%%9.0f:    <%9.0f>\n",     val);
   printf("%%-9.0f:   <%-9.0f>\n\n",  val);

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

(continued on next page)

Example 2–1 (Cont.): Output of the Conversion Specifications

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

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

   printf("%%ld:       <%ld>\n",    i);
   printf("%%lu:       <%lu>\n",    i);
   printf("%%lx:       <%lx>\n\n", i);

   printf("%%s:        <%s>\n",            s);
   printf("%%-9.6s:    <%-9.6s>\n",        s);
   printf("%%-*.*s:    <%-*.*s>\n", 9, 5, s);
   printf("%%6.0s:     <%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:         <     123345>
%9g:          <   123345>

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

%ld:          <-1500000000>
%lu:          <2794967296>
%lx:          <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.h>
#include <stdlib.h>

int main ()
{
   char   buffer[32];
   int    i, pos;
   FILE   *fptr;
                                  /*  Set file pointer         */
   fptr = fopen("data.dat", "w+");
   if (fptr == NULL)
      {
         perror("fopen");
         exit (EXIT_FAILURE); /*  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 (EXIT_FAILURE);
      }
                                  /*  Put record 2 in the buffer */
   if (fgets(buffer, 32, fptr) == NULL)
      {
         perror("fgets");         /*  Exit on fgets error      */
         exit(EXIT_FAILURE);
      }
                                  /*  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:

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

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

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

Chapter 3

Character-Handling Functions and Macros

This chapter describes character-handling functions and macros. Table 3–1 lists and briefly describes all the character-handling functions and macros contained in the PDP–11 C Run-Time Library. These functions and macros are defined in the <ctype.h> header file. For more detailed information, see the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

Character-handling functions are affected by the currently set locale. By default, the *C* locale is set. See Chapter 4 for information on locales.

Table 3–1: Character- and List-Handling Functions and Macros

Function or Macro	Purpose
Character-Testing	
isalnum	Returns a nonzero integer if its argument is an alphanumeric character.
isalpha	Returns a nonzero integer if its argument is an alphabetic character. In PDP–11 C, **isalpha** is true only for characters having **isupper** or **islower** true.
isascii[1]	Returns a nonzero integer if its argument is any ASCII character in the ASCII character set. This function is a Digital extension added for VAX C compatibility.
__ischar	Returns a nonzero integer if its argument is contained in the current character set.

[1]Not defined when compiling /STANDARD=ANSI.

(continued on next page)

Table 3–1 (Cont.): Character- and List-Handling Functions and Macros

Function or Macro	Purpose
Character-Testing	
iscntrl	Returns a nonzero integer if its argument is a delete character or any nonprinting character for each of the character sets supported by PDP–11 C.
isdigit	Returns a nonzero integer if its argument is a decimal digit character (0–9).
isgraph	Returns a nonzero integer if its argument is any printing character with the exception of the space character.
islower	Returns a nonzero integer if its argument is a lowercase alphabetic character.
isprint	Returns a nonzero integer if its argument is a printing character.
ispunct	Returns a nonzero integer if its argument is a punctuation character.
isspace	Returns a nonzero integer if its argument is white space; that is, if it is a space, tab (horizontal or vertical), carriage-return, form-feed, or newline character.
isupper	Returns a nonzero integer if its argument is an uppercase alphabetic character.
isxdigit	Returns a nonzero integer if its argument is a hexadecimal digit.
Character Case-Mapping	
toascii[1]	Converts an 8-bit ASCII character to a 7-bit ASCII character. This function is a Digital extension provided for VAX C compatibility.
tolower	Converts uppercase characters to lowercase characters.
_tolower[1]	Converts uppercase characters to lowercase characters for VAX C compatibility.
toupper	Converts lowercase characters to uppercase characters.
_toupper[1]	Converts lowercase characters to uppercase characters for VAX C compatibility.

[1]Not defined when compiling /STANDARD=ANSI.

3.1 Character-Testing Macros

In PDP–11 C, the macro version of a function is declared in the appropriate header file if a macro version exists. If no macro version exists, the function is used. The header also declares a prototype for the function and maps it to the Run-Time Library (RTL) routine that implements the function.

If the macro exists, using **#undef** followed by the name of the macro ensures that the function is used rather than the macro.

For all macros, a nonzero return value indicates true. A return value of 0 indicates false.

For each character-testing macro, Table 3–2 lists the decimal equivalents of the character values which return true for each of the PDP–11 C supported locales.

Table 3–2: Character Values

Function	Locale	Character Values
isalnum	C	48–57, 65–90, 97–122
	English	48–57, 65–90, 97–122
	Danish	48–57, 65–90, 97–122, 197–198, 201, 216, 220, 229–230, 233, 248, 252
	Digital Multinational	48–57, 65–90, 97–122, 192–207, 209–221, 224–239, 241–253
	Finnish	48–57, 65–90, 97–122, 196–197, 214, 220, 228–229, 233, 246, 252
	French	48–57, 65–90, 97–122, 192, 194, 198–203, 206–207, 212, 215, 217, 219–220, 224, 226, 230–235, 238–239, 244, 247, 249, 251–252
	German	48–57, 65–90, 97–122, 196, 214–215, 220, 228, 246–247, 252
	Italian	48–57, 65–90, 97–122 192, 199–201, 204, 210, 217, 224, 231–233, 236, 242, 249

(continued on next page)

Table 3–2 (Cont.): Character Values

Function	Locale	Character Values
	Norwegian	48–57, 65–90, 97–122, 197–198, 216, 229–230, 248
	Portuguese	48–57, 65–74, 76–86, 88, 90, 97–106, 108–118, 120, 122, 192–195, 199, 201–202, 205, 211, 213, 218, 224–227, 231, 233–234, 237, 243–245, 250
	Spanish	48–57, 65–90, 97–122, 193, 201, 205, 209, 211, 218, 220, 225, 233, 237, 241, 243, 250, 252
	Swedish	48–57, 65–90, 97–122, 196–197, 214, 228–229, 246
isalpha	C	65–90, 97–122
	English	65–90, 97–122
	Danish	65–90, 97–122, 197–198, 201, 216, 220, 229–230, 233, 248, 252
	Digital Multinational	65–90, 97–122, 192–207, 209–221, 224–239, 241–253
	Finnish	65–90, 97–122, 196–197, 214, 220, 228–229, 233, 246, 252
	French	65–90, 97–122, 192, 194, 198–203, 206–207, 212, 215, 217, 219–220, 224, 226, 230–235, 238–239, 244, 247, 249, 251–252
	German	65–90, 97–122, 196, 214–215, 220, 228, 246–247, 252
	Italian	65–90, 97–122, 192, 199–201, 204, 210, 217, 224, 231–233, 236, 242, 249
	Norwegian	65–90, 97–122, 197–198, 216, 229–230, 248

(continued on next page)

Table 3–2 (Cont.): Character Values

Function	Locale	Character Values
	Portuguese	65–74, 76–86, 88, 90, 97–106, 108–118, 120, 122, 192–195, 199, 201–202, 205, 211, 213, 218, 224–227, 231, 233–234, 237, 243–245, 250
	Spanish	65–90, 97–122, 193, 201, 205, 209, 211, 218, 220, 225, 233, 237, 241, 243, 250, 252
	Swedish	65–90, 97–122, 196–197, 214, 228–229, 246
isascii	For all locales	0–127
__ischar	C	0–127
	English	0–127
	Danish	0–127
	Digital Multinational	0–127. 132–151, 155–159, 161–163, 165, 167–171, 176–179, 181–183, 185–189, 191–207, 209–221, 223–239, 241–253
	Finnish	0–127
	French	0–127
	German	0–127
	Italian	0–127
	Norwegian	0–127
	Portuguese	0–127
	Spanish	0–127
	Swedish	0–127
iscntrl	C	0–31, 127
	English	0–31, 127
	Danish	0–31, 127, 132–151, 155–159
	Digital Multinational	0–31, 127, 132–151, 155–159
	Finnish	0–31, 127, 132–151, 155–159

(continued on next page)

Table 3–2 (Cont.): Character Values

Function	Locale	Character Values
	French	0–31, 127, 132–151, 155–159
	German	0–31, 127, 132–151, 155–159
	Italian	0–31, 127, 132–151, 155–159
	Norwegian	0–31, 127, 132–151, 155–159
	Portuguese	0–31, 127, 132–151, 155–159
	Spanish	0–31, 127, 132–151, 155–159
	Swedish	0–31, 127, 132–151, 155–159
isdigit	C	48–57
	English	48–57
	Danish	48–57
	Digital Multinational	48–57
	Finnish	48–57
	French	48–57
	German	48–57
	Italian	48–57
	Norwegian	48–57
	Portuguese	48–57
	Spanish	48–57
	Swedish	48–57
isgraph	C	33–126
	English	33–126
	Danish	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Digital Multinational	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253

(continued on next page)

Table 3–2 (Cont.): Character Values

Function	Locale	Character Values
	Finnish	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	French	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	German	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Italian	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Norwegian	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Portuguese	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Spanish	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Swedish	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
islower	C	97–122
	English	97–122
	Danish	97–122, 229–230, 233, 248, 252

(continued on next page)

Table 3–2 (Cont.): Character Values

Function	Locale	Character Values
	Digital Multinational	97–122, 224–239, 241–253
	Finnish	97–122, 228–229, 233, 246, 252
	French	97–122, 224, 226, 230–235, 238–239, 244, 247, 249, 251–252
	German	97–122, 228, 246–247, 252
	Italian	97–122, 224, 231–233, 236, 242, 249
	Norwegian	97–122, 229–230, 248
	Portuguese	97–106, 108–118, 120, 122, 224–227, 231, 233–234, 237, 243–245, 250
	Spanish	97–122, 225, 233, 237, 241, 243, 250, 252
	Swedish	97–122, 228–229, 246
isprint	C	33–126
	English	32–126
	Danish	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Digital Multinational	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Finnish	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	French	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253

(continued on next page)

Table 3–2 (Cont.): Character Values

Function	Locale	Character Values
	German	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Italian	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Norwegian	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Portuguese	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Spanish	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
	Swedish	33–126, 161–163, 165, 167–171, 176–179, 181–183, 185–187, 189–207, 209–221, 223–239, 241–253
ispunct	C	33–47, 58–64, 91–96, 123–126
	English	33–47, 58–64, 91–96, 123–126
	Danish	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
	Digital Multinational	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
	Finnish	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
	French	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191

(continued on next page)

Table 3–2 (Cont.): Character Values

Function	Locale	Character Values
	German	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
	Italian	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
	Norwegian	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
	Portuguese	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
	Spanish	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
	Swedish	33–34, 39–41, 44–46, 58–59, 63, 91, 93, 123, 125, 161, 183, 191
isspace	C	9–13, 32
	English	9–13, 32
	Danish	9–13, 32
	Digital Multinational	9–13, 32
	Finnish	9–13, 32
	French	9–13, 32
	German	9–13, 32
	Italian	9–13, 32
	Norwegian	9–13, 32
	Portuguese	9–13, 32
	Spanish	9–13, 32
	Swedish	9–13, 32
isupper	C	65–90
	English	65–90
	Danish	65–90, 197–198, 201, 216, 220
	Digital Multinational	65–90, 192–207, 209–221
	Finnish	65–90, 196–197, 214, 220
	French	65–90, 192, 194, 198–203, 206–207, 212, 215, 217, 219–220

(continued on next page)

Table 3–2 (Cont.): Character Values

Function	Locale	Character Values
	German	65–90, 196, 214–215, 220
	Italian	65–90, 192, 199–201, 204, 210, 217
	Norwegian	65–90, 197–198, 216
	Portuguese	65–74, 76-86, 88, 90, 192–195, 199, 201–202, 205, 211, 213, 218
	Spanish	65–90, 193, 201, 205, 209, 211, 218, 220
	Swedish	65–90, 196–197, 214
isxdigit	For all character sets	48–57, 65–70, 97–102

Example 3–1 shows how to use the character-testing macros.

Example 3–1: Character-testing 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.h>
#include <stdio.h>
#include <stdlib.h>

int main ()
{

   int c;
   short i = 0, j = 0, k = 0;
   while ((c = getchar()) != EOF)
      {
         if (isalpha(c))
            i++;
         if (isdigit(c))
            j++;
         if (isspace(c))
            k++;
      }
```

(continued on next page)

Example 3–1 (Cont.): Character-testing Macros

```
   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 Case-Mapping Functions and Macros

The character case-mapping functions and macros perform conversions on characters. These functions include **toascii**, **tolower**, **_tolower**, **toupper**, and **_toupper**. For more information on these functions, see the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

Example 3–2 shows how to use the **toupper** and **tolower** functions.

Example 3–2: 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.h>
#include <stdio.h>          /*  To use EOF identifier   */

int main()
{
   char c, ch;

   while ((c = getchar()) != EOF)
      {
         if (isupper(c))
            ch = tolower(c);
         else
            ch = toupper(c);
         putchar(ch);
      }
}
```

Sample input and output from Example 3–2 are as follows:

```
$ RUN  EXAMPLE2 RETURN
LET'S GO TO THE stonewall INN. CTRL/Z
let's go to the STONEWALL inn.
$
```

Chapter 4

Localization Functions and Macros

This chapter describes the localization functions and macros supported by PDP–11 C. Localization means providing support for displaying data in formats used by various countries, reflecting differences in language and convention.

The header file for the localization is <locale.h>. The <locale.h> header file declares one type and two functions. It also defines several macros used for setting the character set, collating sequence, monetary format, decimal-point character, and date and time formats.

PDP–11 C, through the appropriately formatted **strftime** function, supports the following date formats:

- ISO format: 1990–11–22
- Customary Central European and British format: 22.11.90
- Customary United States format: 11/22/90
- Julian date: 90359
- Airline format 22NOV90

The following code fragment shows how to place the current Julian date into a character array named date.

```
#include <time.h>
#define longest_date_length    11
#define julian_length   6
          .
          .
          .
          char date[longest_date_length];
          time_t t0;
```

```
t0 = time(NULL);
.
.
.
strftime( date, julian_length, "%y%j", localtime(&t0));
.
.
.
```

4.1 The lconv Type

The <locale.h> header file declares one type, **lconv**, which is defined as follows:

```
struct
{
    char        *decimal_point;         /* "." */
    char        *thousands_sep;         /* "" */
    char        *grouping;              /* "" */
    char        *int_curr_symbol;       /* "" */
    char        *currency_symbol;       /* "" */
    char        *mon_decimal_point;     /* "" */
    char        *mon_thousands_sep;     /* "" */
    char        *mon_grouping;          /* "" */
    char        *positive_sign;         /* "" */
    char        *negative_sign;         /* "" */
    char        int_frac_digits;        /* CHAR_MAX */
    char        frac_digits;            /* CHAR_MAX */
    char        p_cs_precedes;          /* CHAR_MAX */
    char        p_sep_by_space;         /* CHAR_MAX */
    char        n_cs_precedes;          /* CHAR_MAX */
    char        n_sep_by_space;         /* CHAR_MAX */
    char        p_sign_posn;            /* CHAR_MAX */
    char        n_sign_posn;            /* CHAR_MAX */
}
            lconv;
```

4.2 The setlocale Function

The setlocale function specifies the indicated character set, collating sequence, monetary format, decimal-point character, and time and date format in the run-time environment.

The setlocale function takes two arguments. The first argument specifies the category. There are six possible values for this argument:

LC_ALL	Indicates all portions of the locale are affected.
LC_COLLATE	Indicates only the collation sequence is affected.

LC_CTYPE	Indicates only the character set is affected.
LC_MONETARY	Indicates only the monetary formations are affected.
LC_NUMERIC	Indicates only the numeric formations are affected.
LC_TIME	Indicates only the time is affected.

The second argument is a character string that specifies the character set for the first argument.

If fewer locale names are supplied than called for by the first argument to **setlocale**, or if a locale is not supported, the default locale for the class is used. If more than five character set names are supplied, the additional names are ignored. If none of the requested locales are supported by the running task, the **setlocale** function will return NULL.

The following example uses the German collating sequence and the Digital Multinational character set:

```
setlocale(LC_ALL, "german,dec_mcs")
```

To inquire about a locale, you can pass a null pointer as the second argument to the **setlocale** function. The name of the current locale for the class indicated by the first argument is returned. For example, if the first argument is LC_ALL, the name of each locale is returned in the following order:

- Collating sequence
- Character set
- Numeric format
- Monetary format
- Time

The following tables indicate the locales and locale types supported by PDP–11 C.

- Table 4–1 lists the character-set and collating sequence locales.
- Table 4–2 lists the monetary and numeric format locales.
- Table 4–3 lists the time locales.

Table 4–1: PDP–11 C Character-Set and Collating Sequence Locales

Character Set	String[1]	Support Module Name/RT-11 Global[2]
C[3]	C	[4]
Danish	danish	c$daty
Digital Multinational	dec_mcs	c$dmty
English	english	c$enty
Finnish	finnish	c$fity
French	french	c$frty
German	german	c$gety
Italian	italian	c$itty
Norwegian	norwegian	c$noty
Portuguese	portuguese	c$poty
Spanish	spanish	c$spty
Swedish	swedish	c$swty

[1]The string must be typed exactly as indicated.
[2]The support module name to be included for taskbuilder/ Global symbol for RT–11 Linker; required to incorporate locale support in the task.
[3]C locale is the ASCII locale.
[4]No user action required for default C support.

Table 4–2: PDP–11 C Monetary and Numeric Locales

Economic Locale	String[1]	Support Module Name/RT-11 Global[2]
C	C	[3]
Austrian	austrian	c$aumf
Belgian Flemish	belgian-flemish	c$bemf
Belgian French	belgian-french	c$bemf
Danish	danish	c$damf
Finnish	finnish	c$fimf

[1]The string must be typed exactly as indicated.
[2]The support module title to be included in ODL file to incorporate locale support in the task.
[3]No user action required for default C support.

(continued on next page)

Table 4–2 (Cont.): PDP–11 C Monetary and Numeric Locales

Economic Locale	String[1]	Support Module Name/RT–11 Global[2]
French	french	c$frmf
German	german	c$gemf
Iceland	icelandic	c$icmf
Ireland	irish	c$irmf
Italian	italian	c$itmf
Netherlands	netherlands	c$nemf
Norwegian	norwegian	c$nomf
Portuguese	portuguese	c$pomf
Spanish	spanish	c$spmf
Swedish	swedish	c$swmf
Swiss German	swiss-german	c$sumf
Swiss French	swiss-french	c$sumf
United Kingdom	united kingdom	c$ukmf
USA	usa	c$usmf

[1]The string must be typed exactly as indicated.
[2]The support module title to be included in ODL file to incorporate locale support in the task.

Table 4–3: PDP–11 C Time Locales

Time Locale	String[1]	Support Module Name/RT–11 Global[2]
C	C	[3]
Austrian	austrian	c$autm
Belgian Flemish	belgian-flemish	c$betm
Belgian French	belgian-french	c$betm
Danish	danish	c$datm
Finnish	finnish	c$fitm

[1]The string must be typed exactly as indicated.
[2]The support module title to be included in ODL file to incorporate locale support in the task.
[3]No user action required for default C support.

(continued on next page)

Table 4-3 (Cont.): PDP-11 C Time Locales

Time Locale	String[1]	Support Module Name/RT-11 Global[2]
French	french	c$frtm
German	german	c$getm
Iceland	icelandic	c$ictm
Italian	italian	c$ittm
Netherlands	netherlands	c$netm
Norwegian	norwegian	c$notm
Portuguese	portuguese	c$potm
Spanish	spanish	c$sptm
Swedish	swedish	c$swtm
Swiss German	swiss-german	c$sutm
Swiss French	swiss-french	c$sutm
United Kingdom	united kingdom	c$uktm

[1]The string must be typed exactly as indicated.

[2]The support module title to be included in ODL file to incorporate locale support in the task.

4.3 The localeconv Function

The **localeconv** function sets the appropriate values for formatting monetary quantities as controlled by the current locale.

For a more detailed description of the **localeconv** function, refer to the PDP-11 C Standard Library Macros and Functions subsection in the Reference Section.

4.4 Including Run-time Support for setlocale Function

Support for the various locales is not automatically included in the user task. In order to include this support, the user must, at taskbuild or link time, name the modules required by the running task.

An example is a task that requires support for the German character types and support for the French monetary and time locales. At taskbuild time, you must refer directly to the three modules providing this support. The module names are C$GETY for German character types, C$FRTM for the

French time locale, and C$FRMF for the French monetary locale. On RSX or RSTS systems, you can reference these names in a taskbuild in the following way:

```
> TKB RETURN
TKB> usrtsk/cp=usrtsk RETURN
TKB> LB:[1,1]CFPURSX/LB:C$GETY:C$FRMF:C$FRTM RETURN
TKB> LB:[1,1]CFPURSX/LB RETURN
TKB> // RETURN
```

Under RT–11, the global symbols C$GETY, C$FRTM, and C$FRMF will be found in the previously named modules allowing the following LINK command to include the needed locale support:

```
.LINK/include/stack:1100/bot:1100 usrtsk,cfpurtl
Library search? C$GETY
Library search? C$FRTM
Library search? C$FRMF
Library search?
.
```

RT–11 LINK

> Please observe that the stack and bottom settings given in the RT–11 LINK example are the minimum required by a PDP–11 C task which includes standard I/O.

In this way, you can specify the particular **setlocale** support required by a task without including any locales that are not required (except perhaps the default C locale).

A complete list of supported locales, and the module names associated with those locales, may be found in Table 4–1, Table 4–2, and Table 4–3.

Chapter 5

General Utility Functions

This chapter lists and briefly describes string conversion, memory management, environment communication, search and sort, integer arithmetic, pseudorandom sequence generation, and multibyte character and string functions. Table 5–1 lists and describes the general utility functions supported by PDP–11 C. These functions are defined in the <stdlib.h> header file. For more detailed information, see the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

Table 5–1: Summary of General Utility Functions

Function	Purpose
String Conversion	
atof	Converts a string of ASCII characters to a number of type **double**.
atoi	Converts a string of ASCII characters to the appropriate **int** numeric value.
atol	Converts a string of ASCII characters to the appropriate **long int** numeric value.
strtod	Converts a string of ASCII characters to a number of type **double**.
strtol	Converts a string of ASCII characters to the appropriate **long int** numeric value.
strtoul	Converts a string of ASCII characters to an unsigned **long int**.

(continued on next page)

Table 5–1 (Cont.): Summary of General Utility Functions

Function	Purpose
Pseudorandom Sequence Generation	
rand	Returns pseudorandom numbers in the range 0 to RAND_MAX.
srand	Provides a seed value for subsequent calls to **rand**.
Memory Management Functions	
calloc	Allocates an area of memory and initializes each element to all bits zero.
free	Makes available for reallocation an area allocated by a previous **calloc, malloc,** or **realloc** call.
malloc	Allocates an area of memory.
realloc	Changes the size of the area pointed to by the first argument to the number of bytes given by the second argument.
Environmental Communication	
abort	Causes the signal, SIGABRT, to be raised and terminates the program if the signal is not handled.
atexit	Registers a function that will be called at program termination.
exit	Terminates the process from which it is called.
getenv	Searches the environment array for the current process and returns the value associated with a specified environment.
system	Passes a given string to the host environment to be executed by a command processor (useful on RSX systems only.)
Search and Sort	
bsearch	Performs a search for a specified object on an array of sorted objects.
qsort	Sorts an array of objects in place.

(continued on next page)

Table 5–1 (Cont.): Summary of General Utility Functions

Function	Purpose
Integer Arithmetic	
abs	Returns the absolute value of an **int**.
div, ldiv	Returns the quotient and remainder after the division of its arguments.
labs	Returns the absolute value of an integer as **long int**.
Multibyte Character and String	
mblen, mbtowc	Determines the number of bytes in a multibyte character pointed to by its character pointer argument.
mbstowcs	Converts a sequence of multibyte characters using the **mbtowc** function.
wcstombs	Converts a sequence of codes that correspond to multibyte characters into a sequence of multibyte characters and stores them in the array pointed to by the character pointer argument.
wctomb	Determines the number of bytes needed to represent a multibyte character.
Converting Between ASCII and RAD50	
__alr50	Converts the first six characters of the input string to an unsigned 32-bit integer corresponding to the radix-50 translation.
__asr50	Converts the first three characters of the input string to an unsigned 16-bit integer corresponding to the radix-50 translation.
__lr50a	Converts an unsigned 32-bit radix-50 string to the corresponding 6-character ASCII character string.
__sr50a	Converts an unsigned 16-bit radix-50 string to the corresponding 3-character ASCII character string.

5.1 String Conversion Functions

The string conversion functions convert strings to numeric values. PDP–11 C supports the following string conversion functions: **atof**, **atoi**, **atol**, **strtod**, **strtol**, and **strtoul**.

5.2 Pseudorandom Sequence Generation

The pseudorandom sequence generation functions generate numbers in a sequence which appears random. PDP–11 C supports the following pseudorandom sequence generation functions: **rand** and **srand**.

5.3 Memory Management Functions

The PDP–11 C memory management functions allocate memory space, free previously allocated memory space, and change the size of a previously allocated memory area. The following memory allocation functions are supported by PDP–11 C: **calloc**, **malloc**, **realloc**, and **free**.

The order and contiguity of storage allocation is unspecified when successive calls to the **calloc**, **malloc**, and **realloc** functions are made. If space can be allocated, the pointer points to the lowest byte address of the allocated space. If space cannot be allocated, a NULL pointer is returned. Each pointer is aligned on an **int** boundary. PDP–11 C returns a NULL pointer when a request is made for an allocation of memory space of 0 bytes.

The memory management functions that allocate memory space round the requested memory size to a size that is divisible by 4 bytes. The function call **malloc** (6) will actually return a pointer to an area of memory that is 8 bytes long.

On the RSX and RSTS/E operating systems and their derivatives, programs must be linked using the /CP taskbuilder switch. For general information on the taskbuilder switch, refer to the taskbuilder manual for the appropriate operating system.

5.3.1 The calloc Function

The **calloc** function obtains blocks of memory space to satisfy the space requirement of an array of n objects each the specified size of each item. If the request cannot be satisfied, NULL is returned. If the memory can be allocated, **calloc** initializes the memory to all bits zero.

5.3.2 The malloc Function

The **malloc** function allocates memory space for an object whose size is specified. If the request cannot be satisfied, NULL is returned. The memory allocated is not initialized.

5.3.3 The realloc Function

The **realloc** function changes the size of an object.

If the first argument to **realloc** is not a pointer returned by the previous call to the **calloc**, **malloc**, or **realloc** functions, or if it points to memory previously freed by the **free** function, a NULL pointer is returned. In the latter case, **realloc** behaves the same as **malloc**.

If the request cannot be satisfied, NULL is returned. If the size of requested memory is greater than the size of the original object, the object may be moved, and the original object is no longer valid.

5.3.4 The free Function

In PDP–11 C the **free** function frees space previously allocated by the **calloc**, **malloc**, or **realloc** functions.

If the argument to **free** is a NULL pointer or if it does not point to space previously allocated by the **calloc**, **malloc**, or **realloc** functions, no action is taken.

5.3.5 Program Example

Example 5–1 shows the use of the **malloc**, **free**, and **calloc** functions.

Example 5–1: 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  <stdlib.h>
#include  <stdio.h>
#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     */
    };

int main ()
{
char   *buffer;                     /* Define pointers to        */
                                    /*  structure (input lines) */
struct line_rec  *first_line, *next_line, *last_line = NULL;

buffer = malloc(MAX_LINE_LENGTH);  /* buffer points to memory   */

if (buffer == 0)                    /* If error ...              */
    {
    perror("malloc");
    exit(EXIT_FAILURE);
    }

puts("Type text - terminate with CTRL/Z");
while (gets(buffer) != NULL)        /* While not CTRL/Z ...      */
    {
                                    /* Allocate for input line   */
    next_line = calloc(1, sizeof (struct line_rec));

    if (next_line == NULL)
        {
        perror("calloc");
        exit(EXIT_FAILURE);
        }

    next_line->data = buffer;       /* Put line in data area     */

    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);
```

(continued on next page)

Example 5–1 (Cont.): Allocating and Deallocating Memory for Structures

```
    if (buffer == 0)
        {
        perror("malloc");
        exit(EXIT_FAILURE);
        }
    }
free(buffer);                      /* Last buffer always unused */
next_line = first_line;            /* Pointer to beginning      */

do
    {
    puts(next_line->data);         /* Write line to screen      */
    free(next_line->data);         /* Deallocate a line         */
    last_line = next_line;
    next_line = next_line->next;
    free(last_line);
    }
while (next_line != NULL);
}
```

The sample input and output for Example 5–1 is as follows:

```
$ RUN  EXAMPLE RETURN
Type text - terminate with CTRL/Z
line one RETURN
line two RETURN
CTRL/Z
EXIT
line one
line two
$
```

5.4 Environmental Communication Functions

The environmental communication functions communicate with the host environment to terminate a process, register a function to be called at program termination, search the environment array for the current process information, and pass a given string to the host environment to be executed by the host environment's command interpreter.

5.4.1 The abort and exit Functions

The **abort** function causes abnormal termination of the program. It returns the value EXIT_FAILURE to the operating system unless the signal SIGABRT is caught and the signal handler does not return. PDP–11 C attempts to flush any buffers and closes any open standard input/output files. Note that **abort** will never return to the function that called it.

The implementation-defined forms of successful and unsuccessful termination for the **exit** function are the values EXIT_FAILURE and EXIT_SUCCESS. The **exit** function calls all functions registered by the **atexit** function in the reverse order of their registration.

The **exit** function causes normal termination of the program and returns a value to the operating system. PDP–11 C flushes any buffers and closes any open standard input/output files.

5.4.2 The getenv Function

The **getenv** function searches an implementation-defined environment list for a string that matches a string pointed to by the argument **name**. The PDP–11 C environment list is provided by the host environment. The PDP–11 C environment list for the **getenv** function is shown in Table 5–2.

Table 5–2: Environment List

Name	Purpose
HOME	The user's login directory.
TERM	The type of terminal being used.
PATH	The default device and directory.
USER	The name of the user who initiated the process.
OPSYS	The operating system the program is using.

The Example 5–2 shows how to use the **getenv** function.

Example 5–2: Searching the Environment for a String

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

int main ()
{
char *buff;

buff = getenv("HOME");
printf ("getenv (\"HOME\") is %s\n",buff);

buff = getenv("TERM");
printf ("getenv (\"TERM\") is %s\n",buff);

buff = getenv("PATH");
printf ("getenv (\"PATH\") is %s\n",buff);

buff = getenv("USER");
printf ("getenv (\"USER\") is %s\n",buff);

buff = getenv("OPSYS");
printf ("getenv (\"OPSYS\") is %s\n",buff);
}
```

The sample input and output for Example 5–2 is as follows:

```
$ run getenv RETURN
getenv ("HOME") is [30,41]
getenv ("TERM") is VT2XX)
getenv ("PATH") is [30,41]
getenv ("USER") is [30,41]
getenv ("OPSYS") is RSX-11M PLUS
$
```

5.4.3 The system Function

The **system** function returns 1 when called with a NULL argument in the RSX execution environment, which indicates that the function is supported on the RSX operating system. When the **system** function is called with a nonnull argument, it passes the specified string to the current command line interpreter, waits for the command to be executed, and returns the value returned by the command.

Passing a command to a command line interpreter is not available on RSTS/E and RT–11 operating systems. If the execution environment is RSTS/E or RT–11, the **system** function always returns 0, indicating that passing a command to a command line interpreter is available on these operating systems.

5.5 Search and Sort Functions

The search and sort functions and macros search an array for a specified object and sort an array of objects. PDP–11 C supports the following search and sort functions: **bsearch** and **qsort**.

5.6 Integer Arithmetic Functions

The integer arithmetic functions and macros return the absolute value of an integer or long integer, and return the quotient and remainder of a division. PDP–11 C supports the following integer arithmetic functions: **abs**, **div**, **ldiv**, and **labs**.

5.7 Multibyte Character and String Functions

The multibyte character and string functions and macros determine the number of bytes in a multibyte character or the number of bytes needed to represent the multibyte character. They also convert a sequence of multibyte characters to a sequence of corresponding code or convert a sequence of code to corresponding multibyte characters. PDP–11 C supports the following multibyte character and string functions: **mblen**, **mbtowc**, **mbstowcs**, **wcstombs**, and **wctomb**. PDP–11 C also contains a set of functions that allows you to copy buffers containing binary data. Note that PDP–11 C multibyte characters are one byte long. For more detailed information on the functions that access binary data, refer to the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

Chapter 6

Math Functions

This chapter summarizes all the math functions contained in the PDP–11 C Run-Time Library. These functions, which are defined in the <math.h> header file, are listed in Table 6–1. For more detailed information, refer to the PDP–11 C Standard Library Macros and Functions subsection in the Reference Section.

Table 6–1: Summary of Math Functions

Function	Purpose
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.
ceil	Returns 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 6–1 (Cont.): Summary of Math Functions

Function	Purpose
floor	Returns the largest integer that is less than or equal to its argument.
fmod	Computes the floating-point remainder of the first argument divided by the second argument.
frexp	Breaks the argument into normalized fraction and to integral powers of 2.
ldexp	Returns a value that is the first argument multiplied by 2 raised to the power of the second argument.
log	Returns the natural logarithm of the double argument.
log10	Returns the base10 logarithm of its argument.
modf	Returns the signed fractional part of the first **modf** argument and assigns the integral part, expressed as a **double**, to the object whose address is specified by the second argument.
pow	Returns a value that is the first argument raised to the power of the second argument.
sin	Returns the sine of its radian argument.
sinh	Returns the hyperbolic sine of its argument.
sqrt	Returns the positive square root of its argument.
tan	Returns the tangent of its radian argument.
tanh	Returns the hyperbolic tangent of its argument.

To help you detect run-time errors, the <errno.h> header file defines the following two symbolic values that are returned by many (but not all) of the math functions:

- **EDOM** indicates that an argument is inappropriate; that is, the argument is not within the function's domain. The return value is 0.
- **ERANGE** indicates that a result is out of range; that is, the argument is too large or too small to be represented by the machine. The return value for overflow is the value of the macro **HUGE_VAL**. An underflow returns a value of 0. PDP–11 C sets the value of the expression *errno* to the value of the macro **ERANGE**.

The <errno.h> header file also defines the variable *errno*. When using the math functions, check the external variable *errno* for either or both of these values, and take the appropriate action if an error occurs.

In Example 6–1, the program example checks the variable *errno* for the value EDOM, which indicates that a negative number was specified as input to the function **sqrt**.

Example 6–1: Checking the Variable *errno*

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

int 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);
}
```

Because the **sqrt** function returns a 0 when a negative number is passed, always check the value of *errno* against the symbolic value of EDOM to ensure that you do not get any unpredictable results.

To test for errors, set *errno* to zero before several operators and then test it at the end to see if any operations failed. The variable *errno* is unchanged if there are no errors.

Example 6–2 shows the functionality of the **tan**, **sin**, and **cos** functions.

Example 6–2: 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.h>                       /*  Include modules    */
#include <stdio.h>

/*  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;
}
int 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 6–2 is:

```
$ 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 7

Using PDP–11 C with Record Management Services

This chapter describes how to use Record Management Services (RMS) from PDP–11 C programs. Table 7–1 lists and briefly describes the PDP–11 C RMS operation macros. Each of these macros are described in the RMS Extension Library Macros subsection in the Reference Section of this manual. Knowledge of Macro–11 and RMS–11 is assumed. For more information refer to the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide*. Note that RMS is not supported on the RT–11 operating system.

Table 7–1: PDP–11 C RMS Macros

Macros	Purpose
RMS$CLOSE	Closes an open file.
RMS$CONNECT	Connects a record stream to an open file and initializes the stream context.
RMS$CREATE	Creates a new file and opens it for processing.
RMS$DELETE	Removes a record from a relative or indexed file.
RMS$DISCONNECT	Terminates a stream and disconnects the internal resources it was using.
RMS$DISPLAY	Writes values into control block fields.
RMS$ENTER	Inserts a file name into a directory file. This macro is not supported on RSTS/E.

(continued on next page)

Table 7–1 (Cont.): PDP–11 C RMS Macros

Macros	Purpose
RMS$ERASE	Erases a file and deletes its directory entry.
RMS$EXTEND	Extends the allocation for an open file.
RMS$FIND:	
Sequential Access	Transfers a record or part of a record from a file to an I/O buffer.
Key Access	Transfers a record or part of a record from a sequential disk file, a relative file, or an indexed file to an I/O buffer.
Record File Access (RFA)	Transfers a record or part of a record from a file to an I/O buffer.
RMS$FLUSH	Writes any unwritten buffers for a stream.
RMS$FREE	Frees a locked bucket for a stream.
RMS$GET:	
Sequential Access	Transfers a record from a file to an I/O buffer and to a user buffer.
Key Access	Transfers a record from a sequential disk file, a relative file, or an indexed file to an I/O buffer and a user buffer.
Record File Access (RFA)	Transfers a record from a file to an I/O buffer and to a user buffer.
RMS$NXTVOL	Advances the context for a stream to the beginning of the next magnetic tape volume. This macro is not supported on RSTS/E.
RMS$OPEN	Opens a file for processing by the calling task.
RMS$PARSE	Analyzes a file specification.
RMS$PUT:	
Sequential Access	Transfers a record from a user buffer to an I/O buffer and to a file.

(continued on next page)

Table 7–1 (Cont.): PDP–11 C RMS Macros

Macros	Purpose
Key Access	Transfers a record from a user buffer to an I/O buffer and to a sequential disk file, a relative file, or an indexed file.
RMS$READ: *Sequential and VBN Access*[1]	Transfers blocks to an I/O buffer.
RMS$RELEASE	This macro is supplied for VMS compatibility only.
RMS$REMOVE	Removes the directory entry for a file. This macro is not supported on RSTS/E.
RMS$RENAME	Changes the directory entry for a file.
RMS$REWIND	Resets the context for a stream to the beginning-of-file. This macro is not supported on RSTS/E.
RMS$SEARCH	Scans a directory and returns a file specification and identifiers in NAM block fields.
RMS$SPACE	Moves a magnetic tape backwards or forwards. This macro is not supported on RSTS/E.
RMS$TRUNCATE	Removes records from the latter part of a sequential file.
RMS$UPDATE	Transfers a record from a user buffer to a disk file, overwriting the existing record.
RMS$WAIT	Suspends processing until an outstanding asynchronous operation on the stream is completed. This macro is not supported on RSTS/E.
RMS$WRITE: *Sequential and VBN Access* [1]	Writes blocks to a file.

[1]Virtual Block Number

Introduction to RMS–11

PDP–11 C provides a set of Run-Time Library functions to perform I/O. Some of these functions perform in the same manner as I/O functions found on C implementations running on UNIX systems.

The PDP–11 C Run-Time Library routines use RMS or File Control Services (FCS) to perform I/O; however, RMS–11 may be accessed directly. This chapter introduces the following RMS topics:

- RMS functions
- PDP–11 C RMS header files
- PDP–11 C and RMS
- RMS example program

This chapter briefly reviews the basic concepts and facilities of RMS and shows examples of their application in PDP–11 C programming. Because this is an overview, the chapter does not explain all RMS concepts and features. For language-independent information concerning RMS, refer to the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide*.

7.1 RMS Functions

RMS provides a number of functions that create and manipulate files. These functions use RMS data structures to define the characteristics of a file and its records. The data structures thus are used as indirect arguments to the function call.

The RMS data structures are grouped into four main categories, as follows:

- File access block (FAB)

  Defines the file's characteristics, such as file organization and record format.

- Record access block (RAB)

  Defines the way in which records are processed, such as the record access mode.

- Extended attribute block (XAB)

  Various kinds of extended attribute blocks contain additional file characteristics, such as the definition of keys in an indexed file. Extended attribute blocks are optional.

- Name block (NAM)

  Defines all or part of a file specification to be used when an incomplete file specification is given in an OPEN or CREATE operation. Name blocks are optional.

RMS uses these data structures to perform file and record operations. Table 7–2 lists some of the common functions.

Table 7–2: Common RMS Run-Time Processing Functions

Category	Function	Description
File Processing	RMS$CREATE	Creates and opens a new file of any organization.
	RMS$OPEN	Opens an existing file and initiates file processing.
	RMS$CLOSE	Terminates file processing and closes the file.
	RMS$ERASE	Deletes a file.
Record Processing	RMS$CONNECT	Associates a file access block with a record access block to establish a record access stream; a call to this function is required before any other record processing function can be used.
	RMS$GET	Retrieves a record from a file.
	RMS$PUT	Writes a new record to a file.
	RMS$UPDATE	Rewrites an existing record to a file.
	RMS$DELETE	Deletes a record from a file.
	RMS$REWIND	Positions the record pointer to the first record in the file.
	RMS$DISCONNECT	Disconnects a record access stream.

All RMS functions are directly accessible from PDP–11 C programs by the FORTRAN calling mechanism. The syntax for any RMS function is:

```
RMS$<operation>
```

or

```
sys$<operation>   (This format is supplied for VAX C compatibility)
```

These two symbols are defined in the <rmsops.h> header file.

In this syntax, <operation> corresponds to the name of the RMS function (such as OPEN or CREATE).

The operations require arguments as described in the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide*. In general, the address of a FAB is required, but there may be additional or optional arguments. The following is a syntax example:

```
RMS$CREATE (fab);
```

Note that these syntax descriptions do not show all the options available when you invoke an RMS function. For a complete description of the RMS calling sequence, refer to the *RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide*.

All RMS functions are declared as type **void**. They do not return a value.

7.2 PDP-11 C and RMS Header Files

The following section describes the nine header files supported by the PDP-11 C RMS Extension Library. The PDP-11 C RMS Extension Library header files functionally replace the RMS-11 macros used by MACRO-11 programmers. Before one of the PDP-11 C macros is used, the appropriate header file must be included by using the **#include** preprocessing directive. It is also possible to declare and initialize RMS data structures by using the **static** or **extern** storage class explicitly at compile time.

7.2.1 The <rms.h> Header

The <rms.h> header file includes all of the PDP-11 C RMS header files supplied by the PDP-11 C RMS Extension Library except the <rmsorg.h> and <rmspoo.h> files.

7.2.2 The <rmsops.h> Header

The <rmsops.h> provides functional prototyping of each RMS operation routine. Additionally it defines the sys$<operation> names used by VAX C to the RMS operation names used by PDP-11 C.

7.2.3 The <fab.h>, <nam.h>, <rab.h>, and <xab.h> Headers

Control blocks are defined as structures in the header files. Including the header files <fab.h>, <rab.h>, <nam.h>, and <xab.h> defines the control blocks.

The <fab.h>, <nam.h>, <rab.h>, and <xab.h> header files define RMS data structures and struct definitions including bit mask and offsets. The following examples define an offset and a bit mask:

```
#define FAB$B_BID (00) / * 0$BID */
```

The offset into the FAB data structure of the BID field is defined as 0.

```
#define FAB$C_BLN (0120) /* FB$BLN FAB Length (bytes) */
```

The BLN bit mask of the FAB data structure is defined to have a constant data field size of 0120.

Declaring and initializing control blocks with a combination of default values and selected values can be done at compile time or at run time.

7.2.3.1 Declaring and Initializing Control Blocks at Compile Time

At compile time, space for the control blocks can be allocated, and they can be initialized and declared at this time as well.

The following example shows how to allocate space for the control blocks The second example shows how to declare and initialize the control blocks manually. In both examples, <class> may be **extern** or **static**.

Example 1:

```
<class> struct FAB fab;          /* declare a FAB */
<class> struct S_RAB s_rab;      /* declare a synchronous RAB */
<class> struct A-RAB a_rab;      /* declare an asynchronous RAB */
<class> struct NAM nam;          /* declare an NAM XAB */
<class> struct XABALL all;       /* declare an ALL XAB */
<class> struct XABDAT dat;       /* declare a  DAT XAB */
<class> struct XEBEC key;       /* declare a  KEY XAB */
<class> struct XABPRO pro;       /* declare a  PRO XAB */
```

Example 2:

```
<class> struct XABPRO proxab = {
        XAB$C_PRO,            /* O$COD field */
        XAB$C_PROLEN,         /* O$BLN field */
        &sumxab,              /* O$NXT field */
        20,                   /* O$PRG field */
        30,                   /* O$PRJ field */
        255,                  /* O$PRO field */
        };
```

7.2.3.2 Declaring and Initializing Control Blocks at Compile Time with Default Values

To declare and initialize a control block at compile time with default values, define the symbol RMSxxx$PROTOTYPE and include the appropriate header file, where xxx describes the block type. The control block is initialized to default values and included in the task. The block is accessed by using cc$rms_xxx, where _xxx is the structure to be defined.

The following example shows how to declare and initialize the FAB with default values and selected values prior to including the appropriate header file:

```
#define RMS_FAB$PROTOTYPE                          /* Declares cc$rms_fab */
#include <fab.h>                       /* Declares cc$rms_fab as default FAB */
#include <string.h>

main ()
{

struct FAB myfab;                                  /* Declares storage for FAB */

memcpy (&myfab,&cc$rms_fab,sizeof(myfab)); /* Copies default values */
myfab.fab$b_org = FAB$C_REL;                       /* Sets to relative org */
myfab.fab$b_lch = 2;                               /* Uses channel 2 */
}
```

Table 7–3 lists and describes the control block types, which may be defined in this manner.

Table 7–3: Control Block Types

Structure	Description
FAB	File access block
NAM	Name block
RAB	Record access block
Extended Attribute Blocks	
XABALL	Area allocation
XABDAT	Date and time
XABPRO	File protection
XABSUM	File summary block

7.2.3.3 Setting Control Block Fields

Data fields may be accessed directly and their contents may be changed by using PDP–11 C language constructs. The following example shows how to set the control block fields:

```
#include <rms.h>

main ()
{

struct FAB      fabblk;             /* Declares a FAB */
struct S_RAB    rabblk;             /* Declares a synchronous RAB */
struct NAM      namblk;             /* Declares a NAM */
long alqval;
short rszsav;

fabblk.fab$b_bid = FAB$C_BID;      /* Copy value from specified field */
fabblk.fab$l_nam = &namblk;        /* Copy value from specified field */
fabblk.fab$l_alq = alqval;         /* Copy value from specified field */
fabblk.fab$w_fop |= FAB$M_RWC;     /* Set bits in 1-byte or 1-word field */
fabblk.fab$w_fop &= FAB$M_RWC;     /* Clear bits in 1-byte or 1-word field */
alqval = fabblk.fab$l_alq;         /* Copy from field to specified location */
if (rabblk.rab$w_rsz == rszsav) /* Compare field value to specified value */
    {                              /* Code is executed if true
    .
    .
    .                                                                      */
    }
else
    {                              /* Code is executed if false
    .
    .
    .                                                                      */
    }
if (fabblk.fab$b_dev & FAB$M_TRM)/* Are specified bits in field set? */
    {                               /* Code is executed if true
    .
    .
    .                                                                      */
    }
else
    {                               /* Code is executed if false
    .
    .
    .                                                                      */
    }
}
```

7.2.4 The <rmsdef.h> Header

The <rmsdef.h> header file defines and declares the values defined by the RMS–11 macro, $RMSTAT. This macro defines RMS–11 success and error values. The following examples show how the bit masks for error codes and success codes are defined:

Error

```
#define RMS$_FLD          (0xx)          /* Comment */

#define RMS$_CCR          (0177340)      /* Can't connect RAB */
```

The value is octal and enclosed in parentheses.

Success

```
#define RMS$SU_FLD        (0xx)          /* Comment */
#define RMS$_SUC          (01)           /* Operation succeeded */
```

7.3 Declaring RMS–11 Facilities

The <rmsorg.h> header file contains the C language statements for including support of the various operations on file organizations within the proper PSECTs. The following example shows how to define the organization and operation:

```
#define RMS$ORG$<org>$<operation>
```

In this syntax, $<org> is one of the following:

IDX	Indexed file organization
DIR	Direct file organization
REL	Relative file organization
SEQ	Sequential file organization

The $<operation> is one of the following:

CRE	CREATE operation
DEL	DELETE operation
FIN	FIND operation
GET	GET operation
PUT	PUT operation
UPD	UPDATE operation

The file organization and the operation must be defined *before* including the <rmsorg.h> header file. The code for defining the RMS facilities is supported by the RMSORG.C file. If you include the source code from this file in the C program, the file organizations and operations you do not use can be deleted or commented out.

The following example shows how to define a DELETE operation for an indexed file, a GET operation for a relative file, and a FIND operation for a sequential file:

```
#define RMS$ORG$IDX$DEL /* Index file organization, DELETE operation */
#define RMS$ORG$REL$GET /* Relative file organization, GET operation */
#define RMS$ORG$SEQ$FIND /* Sequential file organization, FIND operation */
#include <rmsorg.h>
```

7.4 Defining Pool Space

The <rmspoo.h> header file contains the C language statements for allocating space for the various pools within the proper PSECTs. The code for defining pool space is supported by the RMSPOO.C file.

Table 7–4 list the PDP–11 C equivalents of the RMS–11 macros for defining pool space.

Table 7–4: PDP–11 C Symbols for Defining Pool Space

Symbol	Purpose
RMSPBDB	Defines space for BDBs in BDB pool.
RMSPBUF	Defines space for I/O buffers in I/O buffer pool.
RMSPFAB	Defines space for FAB pool.
RMSPIDX	Defines space for IDX pool.
RMSPRAB	Defines space for RABs, for sequential and relative files, and for block-accessed indexed files in RAB pool.
RMSPRABC, **RMSPRABK**, and **RMSPRABX**	Define space for key buffers in key buffer pool.

Pool space must be defined before including the <rmspoo.h> header file. The following is an example of defining pool space:

```
#define RMS$P$FAB        <fabcount>
#define RMS$P$IDX        <indexcount>
#define RMS$P$RAB        <rabcount>
#define RMS$P$RABK       <keysize>
#define RMS$P$RABC       <keychanges>
#define RMS$P$BUF        <bufcount>
#define RMS$P$BDB        <bdbcount>
#include <RMSPOO.H>
```

For further information, refer to the *RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide*.

7.5 Calling Operation Macros

Each RMS operation macro has two equivalent macros in the PDP-11 C RMS Extension Library. They are RMS$NAME and sys$name, where NAME (or name) is the name of the operation macro called.

With the exception of RMS$RENAME and RMS$WAIT, all operation macros take three arguments:

- The address of a FAB or RAB
- The address of an error handler for the operation
- The address of a success handler for the operation

The error and success handlers are optional. If the handlers are not desired, simply omit them or pass –1 to indicate that no handler is used.

The RMS$RENAME macro takes a fourth argument: the address of a FAB for the new file specification. The first argument is the address of a FAB for the old file specification.

The RMS$WAIT macro takes only one argument: the address of the RAB for the operation.

The following example shows how to call operation macros:

```
#include <fab.h>
#include <rab.h>
#include <rmsops.h>

        struct FAB      onefab;
        struct FAB      anotherfab;
        struct S_RAB    arab;
               short    bdbcount;
               void     errh();
               void     succh();

        RMS$CREATE (&onefab);
        RMS$OPEN   (&anotherfab,errh,succh);
        RMS$RENAME (&onefab,(void (*)())-1, void (*)())-1,&anotherfab);
        RMS$WAIT   (&arab);
```

7.6 Writing Completion Handlers

Completion handlers are routines that may be called at the completion of an RMS operation. They may be specified to be invoked upon successful completion of the operation, unsuccessful completion of the operation, or both. The completion handlers may be written in either Macro–11 or C. If the routine is written in C, the fortran calling sequence must be specified in the function declaration of the completion routine. When the completion handler is called, the four arguments to the function are:

1. The address of the RAB or FAB
2. The address of the error handler
3. The address of the success handler
4. The address of the new FAB if RMS$RENAME is called

The following example shows how to write a completion routine:

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

#pragma linkage fortran rmscmp

void rmscmp     (struct FAB *fab,
                 void (*perrh)(),      /* Error address */
                 void (*psuch)(),      /* Success address */
                 struct FAB *newfab)
{
printf ("The RMS STV field is %d\n", fab->fab$w_stv);
}
```

7.7 Using Get-Space Routines

The following sections explain how to use the get-space routines. The PDP–11 C jacket routine, C$RHLP, calls the specified user-provided get-space routine.

The first section describes the **RMS$GETGSA$** routine, which returns the address of the **getspace** function. The second section describes the **RMS$SETGSA$** function, which places the address of the argument's function into the PDP–11 C OTS work area. The third section describes the parameter passing, which would normally be passed by R0, R1, and R2 in a standard RMS call to a user-defined get-space routine.

For additional information, refer to the *RSX–11M/M–PLUS RMS–11 Macro Programmer's Guide*.

7.7.1 The RMS$GETGSA$ Routine

The RMS$GETGSA$ routine returns the address of the **getspace** function that is placed in the PDP–11 C OTS work area by RMS$SETGSA. Consider the following example:

```
#include <rmsops.h>

short (*getspace) ();

getspace = RMS$GETGSA$;
```

The difference between RMS$GETGSA$ and a direct call to the MACRO–11 $GETGSA macro is that $GETGSA returns the address of the jacket routine C$RHLP; RMS$GETGSA$ returns the address of the **getspace** function placed in the OTS work area by the RMS$SETGSA$ macro.

7.7.2 The RMS$SETGSA$ Macro

The RMS$SETGSA$ macro places the address of the argument's function into the PDP–11 C OTS work area, making that routine the one used by RMS–11 to get additional space. The following example shows how to use the RMS$SETGSA$ macro:

```
#include <rmsops.h>

short getspace();
short (*pGetspace) ();

pGetspace = getspace;
RMS$SETGSA$(pGetspace)
```

7.7.3 Receiving Parameters Passed by R0, R1, and R2 During an RMSGSA or RMS$SETGSA$ Macro

The PDP–11 C jacket routine, C$RHLP, calls the get-space routine specified by either an RMSGSA or RMS$SETGSA$ call. When the routine is called, it passes the three parameters, which are normally passed by R0, R1, and R2 during the RMS$GETGSA$ call, to a user-defined get-space routine. The get-space routine must return a pointer to a short. If the space allocation is successful, the address of the first allocated word should be returned. If the space allocation fails, a zero should be returned. Example 7–1 shows how to receive the parameters passed by R0, R1, and R2 and how to use a get-space routine which allows RMS to use the PDP–11 C **malloc** and **free** functions to get and release space.

Example 7–1: Receiving Parameters

```
        extern short getspace (
                int *pool_space,       /*Address of pool free space list*/
                int block_size,        /*Size of requested block*/
                int released)          /*Address of first word released*/

{
if (released)                          /*Releasing memory?*/
        {
        free((void *) released);       /*Yes, call free*/
        return (short *) TRUE;         /*Indicates success*/
        }
return (short *) malloc (block_size);/*No, call malloc*/
}
```

7.8 Using PDP–11 C to Write RMS Programs

PDP–11 C supplies a number of headers that describe the RMS data structures and status codes. Table 7–5 lists the structure tags, which are defined by the header files, the header files, and a description.

Table 7–5: PDP–11 C Data Structures and Headers

Structure Tag	Header File	Description
FAB	fab.h	Defines the file access block structure.
A_RAB(asynchronous) S_RAB(synchronous)	rab.h, rab1.h rab.h, rab1.h	Defines the record access block structure.
NAM	nam.h	Defines the name block structure.
XABALL XABDAT XEBEC XABPRO XABSUM	xab.h xab.h xab.h xab.h xab.h	Defines all the extended attribute block structures.

These header files define all the RMS data structures as structure tag names. However, they perform no allocation or initialization of the structures; these modules describe only a template for the structures. To use the structures, you must create storage for them and initialize all the structure members as required by RMS–11. Note that these header files are part of PDP–11 C RMS–11 RMS Extension Library.

RMS can be used in programs which use PDP–11 C Standard Library I/O functions; however, you must reserve the ones used in accessing RMS directly. Refer to Section 2.7 for information on reserving LUNs.

7.9 RMS Example Program

The example program in this section uses RMS functions to maintain a simple employee file. The file is an indexed file with two keys: social security number and last name. The fields in the record are character strings defined in a structure with the tag record.

The records have the carriage-return attribute. Individual fields in each record are padded with blanks for two reasons. First, key fields must be padded in some way; RMS does not understand PDP–11 C strings with the trailing NUL character. Second, the choice of blank padding as opposed to NUL padding allows the file to be printed or typed without conversion.

The program does not perform range or bounds checking. Only the error checking that shows the mapping of PDP–11 C to RMS is performed. Any other errors are considered to be fatal.

The program is divided into the following sections:

- External data declarations and definitions
- Main program section
- Function to initialize the RMS data structures
- Internal functions to open the file, display HELP information, pad the records, and process fatal errors
- Utility functions
  - ADD
  - DELETE
  - TYPE
  - PRINT
  - UPDATE

The complete (by section) example program follows. Notes on each section are keyed to the numbers at the left of the listing. Example 7–2 shows the external data declarations and definitions.

For information on linking and compiling a PDP–11 C program, refer to the *Guide to PDP–11 C*.

Example 7–2: External Data Declarations and Definitions

```
/* This segment of RMSEXP.C contains external data        *
 * definitions.                                          */

❶ #define RMS_FAB$PROTOTYPE
  #define RMS_RAB$PROTOTYPE
  #define RMS_KEY$PROTOTYPE

  /* Indicate use of Indexed file organization operations  */
  #define RMS$ORG$IDX$CRE
  #define RMS$ORG$IDX$DEL
  #define RMS$ORG$IDX$FIN
  #define RMS$ORG$IDX$GET
  #define RMS$ORG$IDX$PUT
  #define RMS$ORG$IDX$UPD

  #include <rmsdef.h>
❷ #include <rmsorg.h>
  #include <rms.h>
  #include <string.h>
  #include <stdio.h>
  #include <stdlib.h>

❸ #define  DEFAULT_FILE_NAME       ".dat"

  #define RECORD_SIZE             (sizeof record)
  #define SIZE_SSN                15
  #define SIZE_LNAME              25
  #define SIZE_FNAME              25
  #define SIZE_COMMENTS           15
  #define KEY_SIZE                \
  (SIZE_SSN > SIZE_LNAME ? SIZE_SSN: SIZE_LNAME)

❹ static struct  FAB fab;
  static struct  S_RAB rab;
  static struct  XEBEC primary_key,alternate_key;

❺ static struct
     {
        char     ssn[SIZE_SSN], last_name[SIZE_LNAME];
        char     first_name[SIZE_FNAME],
                 comments[SIZE_COMMENTS];
     }  record;
```

(continued on next page)

Example 7–2 (Cont.): External Data Declarations and Definitions

```
❻ static char  response[BUFSIZ],*filename;

❼ static int  rms_status;

❽ static void initialize          (char *);
  static void open_file           (void);
  static void add_employee        (void);
  static void delete_employee     (void);
  static void list_employees      (void);
  static void type_employees      (void);
  static void update_employee     (void);
  static void type_options        (void);
  static void error_exit          (char *);
```

Key to Example 7–2:

❶ The default FAB, RAB, and KEY data structures are brought into the task by defining them before including the <rms.h> header file. The RMSORGIDX$xxx symbols are defined before <rmsorg.h>.

❷ The <rms.h> header file defines the RMS data structures. The <rmsorg.h> header file defines the RMS support that is needed. <stdio.h>, <string.h>, and <stlib.h> header files contain the definitions for Standard I/O, string functions, and common use functions.

❸ Preprocessor variables and macros are defined. A default file RMS Extension .DAT is defined.

The sizes of the fields in the record are also defined. Some (such as the social security number field) are given a constant length. Others (such as the record size) are defined as macros; the size of the field is determined with the **sizeof** operator. PDP–11 C evaluates constant expressions, such as KEY_SIZE, at compile time. No special code is necessary to calculate this value.

❹ Static storage for the RMS data structures is declared. The file access block, record access block, and extended attribute block types are defined by the <rms.h> header file. One extended attribute block is defined for the primary key and one is defined for the alternate key.

❺ The records in the file are defined by using a structure with four fields of character arrays.

❻ The BUFSIZ constant defines the size of the array that will be used to buffer input from the terminal. The filename variable is defined as a pointer to type **char**.

❼ The variable rms_status is used to receive RMS return status information. After each RMS function call, the status of the operation is obtained from the STS field of the FAB or RAB. This status is used to check for specific errors, end-of-file, or successful program execution.

❽ The functional prototypes are defined for the functions used in the applications. After the prototypes are defined, PDP–11 C checks to ensure that the function calls are made with the correct type of parameters.

The main function, shown in Example 7–3, controls the general flow of the program.

Example 7–3: Main Program Section

```
/* This segment of RMSEXP.C contains the main function     *
 * and controls the flow of the program.                    */

❶ main(short argc,char **argv)
  {
❷       if (argc < 1 || argc > 2)
        printf("\nRMSEXP - incorrect number of arguments\n");
     else
        {
           printf("\nRMSEXP - Personnel Database \\ Manipulation Example\n");

❸          filename = (argc == 2 ? *++argv : "personnel.dat");
❹          initialize(filename);
❺          open_file();

           for(;;)
              {
❻                printf("\\Enter option (A,D,E,L,T,U)or \\ ? for help :\n");
                 gets(response);
                 if (response[0] == 'E')
                    break;
                 printf("\n\n");

❼                switch(response[0])
                    {
                       case 'A':  add_employee();
                                         break;

                       case 'D':  delete_employee();
                                         break;
```

(continued on next page)

Example 7–3 (Cont.): Main Program Section

```
                        case 'L':  list_employees();
                                         break;

                        case 'T':  type_employees();
                                         break;

                        case 'U':  update_employee();
                                         break;

                        default:                 printf("RMSEXP - \
Unknown Operation.\n");

                        case '?': case '\0':
                                         type_options();
                    }
                }
❽           sys$close(&fab);
            rms_status = fab.fab$w_sts;

❾           if (rms_status != RMS$SU_SUC)
                        error_exit("$CLOSE");
        }
}
```

Key to Example 7–3:

❶ The main function is entered with two parameters: the first is the number of arguments used to call the program; the second is a pointer to the argument list.

❷ This statement checks that you used the correct number of arguments when invoking the program.

❸ If a file name is included in the command line to execute the program, that file name is used. If no file name is specified, then the file name is PERSONNEL.DAT.

❹ The file access block, record access block, and extended attribute blocks are initialized by calling initialize.

❺ The file is opened by calling **open_file**.

❻ The program displays a menu.

❼ A **switch** statement and a set of **case** statements control the function to be called, determined by the response from the terminal.

❽ The program ends when "E" is entered in response to the menu. At that time, the RMS **sys$close** function closes the employee file.

❾ The rms_status variable is checked for a return status of RMS$SU_SUC. If the file is not closed successfully, then the error-handling function terminates the program.

Example 7–4 shows the function that initializes the RMS data structures. Refer to the RMS documentation for more information about the file access block, record access block, and extended attribute block structure members.

Example 7–4: Function to Initialize RMS Data Structures

```
/* This segment of RMSEXP.C contains the function that    *
 * initializes the RMS data structures.                   */

static void initialize(char *fn)
{

❶   fab =  cc$rms_fab;                /* Initialize FAB     */
    fab.fab$b_bks =  4;
    fab.fab$l_dna =  DEFAULT_FILE_NAME;
    fab.fab$b_dns =  sizeof DEFAULT_FILE_NAME -1;
    fab.fab$b_fac =  FAB$M_DEL | FAB$M_GET | FAB$M_PUT | FAB$M_UPD;
    fab.fab$l_fna =  fn;
    fab.fab$b_fns =  strlen(fn);
    fab.fab$w_mrs =  RECORD_SIZE;
    fab.fab$b_org =  FAB$C_IDX;
    fab.fab$b_rfm =  FAB$C_FIX;
    fab.fab$b_shr =  FAB$M_NIL;
    fab.fab$l_xab =  (char *) &primary_key;
    fab.fab$b_lch =  7;                        /* Use LUN 7 */

❷   memcpy(&rab, &cc$rms_rab, sizeof rab); /* Initialize RAB      */

    rab.rab$l_fab =  &fab;

❸   primary_key =  cc$rms_xabkey;       /* Initialize Primary  *
                                          *  key XAB            */
    primary_key.xab$b_dtp = XAB$C_STG;
    primary_key.xab$b_flg = 0;
❹   primary_key.xab$w_pos0 = record.ssn -  (char *) &record;
    primary_key.xab$b_ref =  0;
    primary_key.xab$b_siz0 = SIZE_SSN;
    primary_key.xab$l_nxt =  (char *) &alternate_key;
    primary_key.xab$l_knm =  "Employee Social Security Number      ";
```

(continued on next page)

Example 7–4 (Cont.): Function to Initialize RMS Data Structures

```
❺   alternate_key = cc$rms_xabkey; /* Initialize Alternate *
                                    * Key XAB             */
    alternate_key.xab$b_dtp = XAB$C_STG;
❻   alternate_key.xab$b_flg =  XAB$M_DUP | XAB$M_CHG;
    alternate_key.xab$w_pos0 =  record.last_name - (char *) &record;
    alternate_key.xab$b_ref =  1;
    alternate_key.xab$b_siz0 =  SIZE_LNAME;
❼   alternate_key.xab$l_knm =   "Employee Last Name            ";
}
```

Key to Example 7–4:

❶ The prototype cc$rms_fab initializes the file access block with default values. Some members have no default values; they must be initialized. Such members include the filename string address and size. Other members can be initialized to override the default values.

❷ The prototype cc$rms_rab initializes the record access block with the default values. In this case, the only member that must be initialized is the rab$l_fab member, which associates a file access block with a record access block.

❸ The prototype cc$rms_xabkey initializes an extended attribute block for one key of an indexed file.

❹ The position of the key is specified by subtracting the offset of the member from the base of the structure.

❺ A separate extended attribute block is initialized for the alternate key.

❻ This statement specifies that more than one alternate key can contain the same value (XAB$M_DUP), and that the value of the alternate key can be changed (XAB$M_CHG).

❼ The key-name member is padded with blanks because it is a fixed-length, 32-character field.

Example 7–5 shows the internal functions for the program.

Example 7–5: Internal Functions

```
/* This segment of RMSEXP.C contains the functions that   *
 * control the data manipulation of the program.          */

static void open_file()
{
   sys$open(&fab);
❶  rms_status =  fab.fab$w_sts;
   if (rms_status != RMS$SU_SUC)
      {
      if (rms_status == RMS$_FNF)
         {
         sys$create(&fab);
         rms_status =  fab.fab$w_sts;
         if (rms_status != RMS$SU_SUC)
            error_exit("$OPEN");

         printf("[Created new data file.]\n");
         }
      else
            error_exit("$OPEN");
      }
❷  sys$connect(&rab);
   rms_status = rab.rab$w_sts;
   if (rms_status != RMS$SU_SUC)
      error_exit("$CONNECT");
}

❸ static void type_options(void)
{
   printf("Enter one of the following:\n\n");
   printf("A     Add an employee.\n");
   printf("D     Delete an employee specified by SSN.\n");
   printf("E     Exit this program.\n");
   printf("L     List employee(s) by ascending SSN to a file.\n");
   printf("T     Type employee(s) by ascending last name on terminal.\n");
   printf("U     Update employee specified by SSN.\n\n");
   printf("?     Type this text.\n");
}
```

(continued on next page)

Example 7–5 (Cont.): Internal Functions

```
❹ static pad_record()
  {
     int       i;

     for(i = strlen(record.ssn); i < SIZE_SSN; i++)
        record.ssn[i] = ' ';
     for(i = strlen(record.last_name); i < SIZE_LNAME; i++)
        record.last_name[i] = ' ';
     for(i = strlen(record.first_name); i < SIZE_FNAME; i++)
        record.first_name[i] = ' ';
     for(i = strlen(record.comments);i < SIZE_COMMENTS; i++)
        record.comments[i] = ' ';
  }

  /* This subroutine is the fatal error handling routine.   */

❺ static void error_exit (char *operation)
  {
     printf("RMSEXP - file %s failed (%s)\n",
             operation, filename);
     exit(rms_status);
  }
```

Key to Example 7–5:

❶ The open_file function uses the RMS **sys$open** function to open the file. If the file is not found, the RMS **sys$create** function is used to create the file, giving the address of the file access block as an argument. The status information is obtained from the fab$w_sts field of the FAB.

❷ The RMS **sys$connect** function associates the record access block with the file access block.

❸ The type_options function, called from the main function, prints help information. Once the help information is displayed, control returns to the main function, which processes the response that is typed at the terminal.

❹ For each field in the record, the pad_record function fills the remaining bytes in the field with blanks.

❺ This function handles fatal errors. It prints the name of the function that caused the error, returns a PDP–11 error code (if appropriate), and exits the program.

Example 7–6 shows the function that adds a record to the file. This function is called when "a" or "A" is entered in response to the menu.

Example 7–6: Utility Function: Adding Records

```
/* This segment of RMSEXP.C contains the function that    *
 * adds a record to the file.                             */

static void add_employee(void)
{
❶   do
        {
            printf("(ADD)   Enter Social Security Number  ");

            gets(response);
        }
    while(strlen(response) == 0);

    strncpy(record.ssn,response,SIZE_SSN);

    do
        {
            printf("\n(ADD)    Enter Last Name ");

            gets(response);
        }
    while(strlen(response) == 0);

    strncpy(record.last_name,response,SIZE_LNAME);

    do
        {
            printf("\n(ADD)    Enter First Name ");

            gets(response);
        }
    while(strlen(response) == 0);

    strncpy(record.first_name,response,SIZE_FNAME);

    do
        {
            printf("n\\(ADD)    Enter Comments ");

            gets(response);
        }
    while(strlen(response) == 0);

    strncpy(record.comments,response,SIZE_COMMENTS);

❷   pad_record();

❸   rab.rab$b_rac =  RAB$C_KEY;
    rab.rab$l_rbf =  (char *) &record;
    rab.rab$w_rsz =  RECORD_SIZE;
```

(continued on next page)

Example 7–6 (Cont.): Utility Function: Adding Records

```
❹   sys$put(&rab);
    rms_status = rab.rab$w_sts;
❺   if (rms_status != RMS$SU_SUC && rms_status !=
                      RMS$_DUP)
       error_exit("$PUT");
    else
       if (rms_status == RMS$SU_SUC)
          printf("\n[Record added successfully.]\n");
       else
          printf("\nRMSEXP - Existing employee with same SSN, not added.\n");
}
```

Key to Example 7–6:

❶ A series of **do** loops controls the input of information. For each field in the record, a prompt is displayed. The response is buffered, and the field is copied to the structure.

❷ When all fields have been entered, the pad_record function pads each field with blanks.

❸ Three members in the record access block are initialized before the record is written. The record access member (rab$b_rac) is initialized for keyed access. The record buffer and size members (rab$l_rbf and rab$w_rsz) are initialized with the address and size of the record to be written.

❹ The RMS **sys$put** function writes the record to the file.

❺ The rms_status variable is checked. If the return status is normal, or if the record has a duplicate key value and duplicates are allowed, the function prints a message stating that the record was added to the file. Any other return value is treated as a fatal error, causing error_exit to be called.

Example 7–7 shows the function that deletes records. This function is called when "d" or "D" is entered in response to the menu.

Example 7–7: Utility Function: Deleting Records

```
  /* This segment of RMSEXP.C contains the function that    *
   * deletes a record from the file.                        */

  static void delete_employee(void)
  {
     int i;
❶    do
        {
           printf("\n(DELETE) Enter Social Security Number   ");
           gets(response);
           i =  strlen(response);
        }
     while(i == 0);

❷    while(i < SIZE_SSN)
        response[i++] = ' ';

❸    rab.rab$b_krf =  0;
     rab.rab$l_kbf =  response;
     rab.rab$b_ksz =  SIZE_SSN;
     rab.rab$b_rac =  RAB$C_KEY;

❹    sys$find(&rab);
     rms_status = rab.rab$w_sts;

❺    if (rms_status != RMS$SU_SUC && rms_status != PMS$_RNF)
        error_exit("$FIND");
     else
        if (rms_status == RMS$_RNF)
           printf("\nRMSEXP - specified employee does not exist.\n");

        else
           {
❻             sys$delete(&rab);
              rms_status = rab.rab$w_sts;
              if (rms_status != RMS$SU_SUC)
                 error_exit("$DELETE");
              printf("\n");
           }
  }
```

Key to Example 7–7:

❶ A do loop prompts the user to type a social security number at the terminal and places the response in the response buffer.

❷ The social security number is padded with blanks.

❸ Some members in the record access block must be initialized before the program can locate the record. Here, the key of reference (0 specifies the primary key), the location and size of the search string (this is the address of the response buffer and its size), and the type of record access (in this case, keyed access) are given.

4. The RMS **sys$find** function locates the record specified by the social security number entered from the terminal.
5. The program checks the rms_status variable for the values RMS$SU_SUC and RMS$_RNF (record not found). A message is displayed if the record cannot be found. Any other error is a fatal error.
6. The RMS **sys$delete** function deletes the record. The status returned in rab$w_sts is only checked for success.

The type_employees function in Example 7–8 displays the employee file at the terminal. This function is called from the main function when "t" or "T" is entered in response to the menu.

Example 7–8: Utility Function: Typing the File

```
/* This segment of RMSEXP.C contains the function that    *
 * displays a single record at the terminal.              */

void type_employees(void)
{
❶   int  number_employees;

❷   rab.rab$b_krf =  1;

❸   sys$rewind(&rab);
    rms_status = rab.rab$w_sts;
    if (rms_status != RMS$SU_SUC)
       error_exit("$REWIND");

❹   printf("\n\nEmployees (Sorted by Last Name)\n\n");
    printf("Last Name       First Name          SSN       \
           Comments\n");

    printf("---------       ----------          ---------\
           --------\n\n");
❺   rab.rab$b_rac =  RAB$C_SEQ;
    rab.rab$l_ubf =  (char *) &record;
    rab.rab$w_usz =  RECORD_SIZE;

❻   for(number_employees = 0; ; number_employees++)
       {
          sys$get(&rab);
          rms_status = rab.rab$w_sts;
          if (rms_status != RMS$SU_SUC && rms_status != RMS$_EOF)
             error_exit("$GET");
          else
             if (rms_status == RMS$_EOF)
                break;
```

(continued on next page)

Example 7–8 (Cont.): Utility Function: Typing the File

```
        printf("%.*s%.*s%.*s%.*s\n",
                SIZE_LNAME, record.last_name,
                SIZE_FNAME, record.first_name,
                SIZE_SSN, record.ssn,
                SIZE_COMMENTS, record.comments);
    }

❼  if (number_employees)
      printf("\nTotal number of employees = %d.\n", number_employees);
   else
      printf("[Data file is empty.]\n");
}
```

Key to Example 7–8:

❶ A running total of the number of records in the file is kept in the number_employees variable.

❷ The key of reference is changed to the alternate key, so that the employees are displayed in alphabetical order by last name.

❸ The file is positioned to the beginning of the first record according to the new key of reference, and the status of the **sys$rewind** function is checked for success.

❹ A heading is displayed.

❺ Sequential record access is specified, and the location and size of the record is given.

❻ A **for** loop controls the following operations:

- Incrementing the number_employees counter
- Locating a record and placing it in the record structure, using the RMS **sys$get** function
- Checking the status of the RMS **sys$get** function
- Displaying the record at the terminal

❼ This **if** statement checks for records in the file. The result is a display of the number of records or a message indicating that the file is empty.

Example 7–9 shows the function that prints the file on the printer. This function is called by the main function when "p" or "P" is entered in response to the menu.

Example 7–9: Utility Function: Printing the File

```
/* This segment of RMSEXP.C contains the function that    *
 * outputs the file to a list file.                       */

static void list_employees(void)
{
   int  number_employees;
   FILE *fp;

1  fp = fopen("personnel.lis", "w");
   if (fp == NULL)
      {
         perror("RMSEXP - failed opening listing file");

         exit(EXIT_FAILURE);
      }

2  rab.rab$b_krf =  0;

3  sys$rewind(&rab);
   rms_status = rab.rab$w_sts;
   if (rms_status != RMS$SU_SUC)
      error_exit("$REWIND");

4  fprintf(fp,"\n\nEmployees (Sorted by SSN)\n\n");

   fprintf(fp,"Last Name      First Name       SSN      \
     Comments\n");

   fprintf(fp,"---------      ----------       ---------\
     --------\n\n");

5  rab.rab$b_rac =  RAB$C_SEQ;
   rab.rab$l_ubf =  (char *)&record;
   rab.rab$w_usz =  RECORD_SIZE;

6  for(number_employees = 0; ; number_employees++)
      {
         sys$get(&rab);
         rms_status = rab.rab$w_sts;
         if (rms_status != RMS$SU_SUC &&
             rms_status != RMS$_EOF)
            error_exit("$GET");
         else
            if (rms_status == RMS$_EOF)
               break;

         fprintf(fp, "%.*s%.*s%.*s%.*s",
                 SIZE_LNAME,record.last_name,
                 SIZE_FNAME,record.first_name,
                 SIZE_SSN,record.ssn,
                 SIZE_COMMENTS,record.comments);
      }
```

(continued on next page)

Example 7–9 (Cont.): Utility Function: Printing the File

```
❼   if (number_employees)
       fprintf(fp, "\nTotal number of employees = %d.\n", number_employees);
    else
       fprintf(fp,"\n[Data file is empty.]\n");
    fclose(fp);
       printf("[Listing file \\ ""personnel.lis\\"" created.]\n");
}
```

Key to Example 7–9:

❶ This function creates a sequential file and outputs it as a text file. The file is created by using the Standard I/O Run-Time Library function **fopen**, which associates the file with the file pointer, fp.

❷ The key of reference for the indexed file is the primary key.

❸ The **sys$rewind** function positions the file at the first record. The status is checked for success.

❹ A heading is written to the sequential file by using the Standard I/O function **fprintf**.

❺ The record access, user buffer address, and user buffer size members of the record access block are initialized for keyed access to the record located in the record structure.

❻ A **for** loop controls the following operations:

- Initializing the running total and then incrementing the total at each iteration of the loop
- Locating the records and placing them in the record structure with the RMS **sys$get** function, one record at a time
- Checking the rms_status information for success and end-of-file
- Writing the record to the sequential file

❼ The number_employees counter is checked. If it is 0, a message is printed indicating that the file is empty. If it is not 0, the total is printed at the bottom of the listing.

Example 7–10 shows the function that updates the file. This function is called by the main function when "u" or "U" is entered in response to the menu.

Example 7–10: Utility Function: Updating the File

```
/* This segment of RMSEXP.C contains the function that    *
 * updates the file.                                       */

static void update_employee(void)
{
   int i;
❶  do
      {
         printf("(UPDATE) Enter Social Security Number ");

         gets(response);
         i =  strlen(response);
      }
   while(i == 0);

❷  while(i < SIZE_SSN)
      response[i++] = ' ';

❸  rab.rab$b_krf =  0;
   rab.rab$l_kbf =  &response;
   rab.rab$b_ksz =  SIZE_SSN;
   rab.rab$b_rac =  RAB$C_KEY;
   rab.rab$l_ubf =  (char *) &record;
   rab.rab$w_usz =  RECORD_SIZE;

❹  sys$get(&rab);
   rms_status = rab.rab$w_sts;
   if (rms_status != RMS$SU_SUC && rms_status != RMS$_RNF)
      error_exit("$GET");
   else
      if (rms_status == RMS$_RNF)
         printf("\nRMSEXP - specified employee does not exist.\n");
❺     else
         {
            printf("\nEnter the new data or RET to leave \
data unmodified.\n\n");

            printf("\nLast Name:");
            gets(response);
            if (strlen(response))
               strncpy(record.last_name, response,
                       SIZE_LNAME);

            printf("First Name:");
            gets(response);
            if (strlen(response))
               strncpy(record.first_name, response,
                       SIZE_FNAME);

            printf("Comments:");
            gets(response);
            if (strlen(response))
               strncpy(record.comments, response, SIZE_COMMENTS);
```

(continued on next page)

Example 7–10 (Cont.): Utility Function: Updating the File

```
❻           pad_record();

❼           sys$update(&rab);
            rms_status = rab.rab$w_sts;
            if (rms_status != RMS$SU_SUC)
               error_exit("$UPDATE");

            printf("\n[Record has been successfully updated.]\n");
         }
}
```

Key to Example 7–10:

❶ A **do** loop prompts for the social security number and places the response in the response buffer.

❷ The response is padded with blanks, so that it will correspond to the field in the file.

❸ Some of the members in the record access block are initialized for the operation. The primary key is specified as the key of reference, the location and size of the key value are given, keyed access is specified, and the location and size of the record are given.

❹ The RMS **sys$get** function locates the record and places it in the record structure. The function checks the rms_status value for RMS$_NORMAL and RMS$_RNF (record not found). If the record is not found, a message is displayed. If the record is found, the program prints instructions for updating the record.

❺ For each field (except the social security number, which cannot be changed), the program displays the current value for that field. If you press the RETURN key, the record is placed in the record structure unchanged. If you make a change to the record, the new information is placed in the record structure.

❻ The fields in the record are padded with blanks.

❼ The RMS **sys$update** function rewrites the record. The program then checks that the update operation was successful. Any error causes the program to call the fatal error-handling routine.

Example 7–11 shows how to reserve a lun.

Example 7–11: Reserving a lun for Use by RMS

```
❶ const short $PRLUN[2] = {1,0200}; /* reserve lun 7 */
```

Key to Example 7–11:

❶ This code programs PDP–11 C Standard I/O to reserve lun 8 for use by RMS because RMS must use a lun to access the file.

Chapter 8

Using PDP–11 C with File Control Services

This chapter describes how to use File Control Services (FCS) with PDP–11 C programs. The reader is assumed to have a working knowledge of MACRO–11 and wishes to access FCS in a similar fashion through the PDP–11 C FCS Extension Library using PDP–11 C language constructs. Refer to the *RSX–11M–PLUS and Micro/RSX I/O Operations Reference Manual* for more detailed information. The following topics are described in this chapter:

- Compile-time initialization of the File Descriptor Block (FDB) and Default Filename Block (DFB)
- The FCS header files
- Run-time initialization of the FDB and file storage region (FSR)
- File processing
- File control routines
- Command-line processing

Table 8–1 lists the macros supported by the PDP–11 C FCS Extension Library. Each of these macros are described in the FCS Extension Library Macros subsection in the Reference Section of this manual.

Table 8–1: PDP–11 C FCS Macros

Macro	Purpose
Compile-Time FDB Declaration and Initialization	
FCS$FDBDF$	Allocates space in the program for the FDB.
Run-Time FDB Initialization	
FCS$FSRSZ$	Establishes the size of the FSR.
Run-Time FSR Initialization	
FCS$FINIT$	Initializes coding to set up the FSR.
File Processing	
FCS$CLOSE$	Terminates file processing.
FCS$DELET$	Removes a named file from the associated volume directory.
FCSGET	Reads logical data records from a file.
FCSGETR	Reads fixed-length records from a file in random mode.
FCSGETS	Reads records from a file in sequential mode.
FCS$OFID$x	Opens an existing file by using file identification information in the FNB.
FCS$OFNB$x	Opens a file by using file name information in the FNB.
FCS$OPEN$x	Opens and prepares a file for processing. The *x* is the alphabetic suffix indicating the type of operation to be performed on the file.
FCS$OPNS$x	Opens and prepares a file for processing and allows shared access to that file.
FCS$OPNT$D	Creates and opens a temporary file for processing.
FCS$OPNT$W	Creates and opens a temporary file for processing data.
FCSPUT	Writes logical data records to a file.

(continued on next page)

Table 8–1 (Cont.): PDP–11 C FCS Macros

Macro	Purpose
File Processing	
FCSPUTR	Writes fixed-length records to a file in random mode.
FCSPUTS	Writes records to a file in sequential mode.
FCS$READ$	Reads virtual data blocks from a file.
FCS$WAIT$	Suspends program execution until a requested block I/O operation is completed.
FCS$WRITE$	Writes virtual data blocks to a file.
File Control Routines	
FCS$ASCPP and FCS$PPASC	Converts a directory string from ASCII to binary or from binary to ASCII.
FCS$ASLUN	Assigns a logical unit number (LUN) to a specified device and unit and returns the device information to a specified FDB filename block.
FCS$CTRL	Performs device-specific control functions.
FCS$DLFNB	Deletes a file by FNB.
FCS$ENTER	Inserts an entry by file name into a directory.
FCS$EXPLG	Expands a logical name and returns a pointer to the task that points to the expanded string.
FCS$EXTND	Extends either contiguous or noncontiguous files.
FCS$FIND	Locates a directory entry by file name and lists it in the file identification field (N.FID) in both the MFD and UFD.
FCS$FLUSH	Writes the block buffer to the file being written in record mode.
FCS$GTDID and FCS$GTDIR	Inserts directory information in a specified file name block (FNB).
FCS$MARK	Points to a byte or record within a specified file.
FCS$MRKDL	Marks a temporary file for deletion.
FCS$PARSE	Performs any necessary logical expansion and parses the resultant string.

(continued on next page)

Table 8–1 (Cont.): PDP–11 C FCS Macros

Macro	Purpose
File Control Routines	
FCS$POINT, FCS$POSIT, and FCS$POSRC	Points to a byte or record within a specified file.
FCS$PRINT$	Queues a file for printing on a specified device.
FCS$PRSDI	Same as $PARSE but performs only those operations associated with requisite directory identification information.
FCS$PRSDV	Same as $PARSE but performs only those operations associated with requisite device and unit information.
FCS$PRSFN	Same as $PARSE but performs only operations associated with requisite file name, file type, and file version information.
FCS$REMOV	Deletes an entry from a directory by file name.
FCS$RENAM	Changes the name of a file in its associated directory.
FCS$RDFDR	Reads and writes directory string descriptors.
FCS$RDFFP	Reads and writes the default file protection word in a location in the program section of the FSR.
FCS$RDFUI	Reads and writes the default UIC maintained program section.
FCS$RFOWN	Reads the contents of the file owner word in the program section.
FCS$TRNCL	Truncates a file to the logical end of the file, deallocates any space beyond that point, and closes the file.
FCS$WDFDR	Reads and writes directory string descriptors.
FCS$WDFFP	Reads and writes the default file protection word in a location in the program section of the FSR.

(continued on next page)

Table 8–1 (Cont.): PDP–11 C FCS Macros

Macro	Purpose
File Control Routines	
FCS$WDFUI	Reads and writes the default UIC maintained program section.
FCS$WFOWN	Initializes the file owner word in the program section.
FCS$XQIO	Executes a specified QIO$ function and waits for its completion.

For more information about these macros, refer to the *RSX–11M–PLUS Operations Manual* and *Micro/RSX I/O Operations Manual*.

8.1 Introduction to the FCS Extension Library

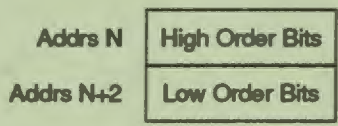
The PDP–11 C FCS Extension Library provides an access to FCS which is similar to accessing FCS from MACRO–11. The FCS extension library supports file control functions.

PDP–11 C provides three FCS header files:

- The <fcs.h> header file provides functional prototyping for each routine and declares a number of macros for accessing FCS with PDP–11 C.
- The <fcsfhb.h> header file defines the file header block.
- The <fcsiff.h> header file defines the index file format.

Two word quantities, such as the BKVB field of the FDB, are interpreted by FCS as shown Figure 8–1, which is opposite from how PDP–11 C stores integers of type **long**:

Figure 8–1: PDP–11 C Integer Storage



NU–2126A–RA

Two word fields are defined by the <fcs.h> header file as two, short, fields such as fcs$f$bkvb and fcs$f$bkvb2. When placing the values in these fields, the high-order bits must be placed in the first word, fcs$f$bkvb; the low-order bits must be placed in the second word, fcs$f$bkvb2.

However, the two MACROs that use long arguments, **FCS$READ$** and **FCS$WRITE$**, accept long integers as stored by PDP–11 C and convert them to the format expected by FCS before sending them to FCS.

For additional information on MACRO–11 and FCS, refer to the *RSX–11M–PLUS and Micro/RSX I/O Operations Reference Manual*.

FCS can be used in programs which use PDP–11 C Standard Library I/O functions; however, you must reserve the ones used in accessing FCS directly. Refer to Section 2.7 for information on reserving LUNs.

8.2 Declaring and Initializing the File Descriptor Block

Before you perform FCS I/O operations, you must declare and initialize an FDB for each file. To declare the FDB, use the FCS$FDBDF$ macro or explicitly declare an fcs$fdb object. To initialize an FDB, explicitly initialize an fcs$fdb object during its declaration, directly access and change the data structures through run-time FDB initialization, or use the file processing macros.

8.2.1 The <fcs.h> Header File

The <fcs.h> header file includes a compile-time FDB declaration macro but does not include a compile-time initialization macro. However, the FDB can be declared manually by using the *static* or *extern* storage class and initialized at compile time, as shown in Section 8.2.2.

The <fcs.h> header file defines the following fcs$fdb structure:

- File attribute section of the FDB
- Record access section of the FDB
- Block access section of the FDB
- File-open section of the FDB
- Block buffer section of the FDB

You must use the **#include** <fcs.h> statement to use any of the functions defined by the <fcs.h> header file.

Values used by FCS are defined in the <fcs.h> header file in the following manner:

```
#define FCS$F$RTYP (00000) /* Equivalent to MACRO-11 definition of F.RTYPE */
```

8.2.2 Compile-Time Initialization of the FDB

Manual declaration and compile-time initialization of the FDB are done by defining the fcs$fdb structure. The fcs$fdb structure functionally replaces the FDAT$A, FDRC$A, FDBK$A, FDOP$A, and FDBF$A FCS macros. The following example shows how to define the fcs$fdb structure (<class> may be either static or extern):

```
<class> fcs$fdb myfdb = {

                    FCS$R$FIX         /* F.RTYP field */
                    FCS$RFD$CR,       /* F.RATT field */
                    133,              /* F.RSIZ field */
                    4,                /* F.HIBK field */
                    3,                /* F.EFBK field */
                    .
                    .
                    .
                    0,                /* F.FNB field */
                    };
```

For further information, refer to the *RSX-11M-PLUS and Micro/RSX I/O Operations Reference Manual*.

8.2.3 Compile-Time Initialization of the Default Filename Block

Compile-time initialization of the DFB is done by defining the fcs$fnb structure.

The following example shows how the DFB is initialized at compile time (<class> may be either static or extern):

```
<class> fcs$fnb myfnb = {
                        0,              /* N.FID field */
                        0,
                        0,
                        'MYF' __RAD50,  /* N.FNAM field */
                        'ILE' __RAD50,  /* N.FNAM field */
                        '   ' __RAD50,  /* N.FNAM field */
                        'TXT' __RAD50,  /* N.FTYP field */
                        3,              /* N.FVER field */
                        FCS$NB$VER,     /* N.STAT field */
                        0,              /* N.NEXT field */
                        0,              /* N.DID  field */
                        0,
                        0,
                        'SY',           /* N.DVNM field */
                        0,              /* N.UNIT field */
                        };
```

8.2.4 Run-Time FDB Initialization and the File Storage Region

Run-time initialization of the FDB and the FSR is done by using C language constructs directly to access and change the data structures. Run-time initialization functionally replaces the FDAT$R, FDRC$R, FDBK$R, FDOP$R, and FDBF$R FCS macros. Consider the following examples:

```
#include <fcs.h>
FCS$FDBDF$(auto, myfdb)

myfdb.fcs$f$rtyp = FCS$R$FIX;
myfdb.fcs$f$rsiz = 132;
myfdb.fcs$f$facc = FCS$FA$WRT | FCS$FA$SHR;
```

The FCS$FDBDF$ macro takes two arguments which correspond to the arguments of the MACRO–11 FDBDF$ macro: the C storage class used to define the FDB, and the name of the FDB.

```
#include <fcs.h>
FCS$FSRSZ$(2,1024)
```

The **FCS$FSRSZ$** macro takes two arguments which correspond to the arguments of the MACRO–11 FSRSZ macro. PDP–11 C generates the correct PSECT and control transfer; therefore, the PSECT of the FSRSZ macro argument is not necessary.

To initialize the file storage region, include the following statements:

```
#include <fcs.h>

FCS$FINIT$
```

The FCS$FINIT$ macro has no arguments.

8.3 File Processing

Each PDP–11 C FCS Extension Library routine takes the parameters passed to it and forwards them to the corresponding FCS routine. Each of them returns a value of 1 if the operation is successful and 0 if it is not, as defined in the <fcs.h> header file.

Some of these routines allow user-defined error routines to be specified. If user-defined error routines are specified, the user must ensure that the error routine does not alter the carry-bit of the Processor Status Word (PSW). If the carry-bit is changed, it must be changed back to its original status; otherwise, an improper return value may result.

Some FCS file control routines use the carry-bit to indicate that they completed successfully; others do not. For those routines that use the carry-bit to indicate success, the equivalent PDP–11 C routine returns the value TRUE (1) if the operation completed successfully and the value FALSE (0) if the operation did not complete successfully. For those routines that do not use the carry-bit to indicate success, the equivalent PDP–11 C routine is declared as a function returning void or no value. For further information on the FCS file control routines, see the *RSX–11M–PLUS and the Micro/RSX I/O Operations Reference Manual.*

8.4 FCS Example Program

The example program in this section uses FCS functions to copy a file. The program is divided into two sections:

- External data declarations and definitions
- Main program section

Example 8–1: External Data Declarations and Definitions

```
/* This segment of CRCOPB.C contains external data definitions. */

#pragma list title "CRCOPB"            /* Card reader copy routine    */

❶ #include "fcs.h"
  #include <stdio.h>
  #include <stdlib.h>

❷ const   short $PRLUN[2] = {1,030};     /* reserve luns 3 and 4 */

  #define INLUN   3
  #define OUTLUN  4

❸ FCS$FSRSZ$(2,1024)
❹ FCS$FDBDF$(static,fdbin)
  FCS$FDBDF$(static,fdbout)
```

Key to Example 8–1:

❶ The <fcs.h> header file defines the FCS data structures. The <stdio.h> header file defines functions used for Standard I/O and the <stdlib.h> header file defines the **exit** function.

❷ The LUNs which access FCS are reserved. This prevents PDP–11 C from trying to use them.

❸ FCS$FSRSZ$ defines the size of the FSR.

❹ This line and the next line define the input and output FBDs. They state the storage class where the FDB resides. The macros define the FDB's as structures which allows easy access to the various fields.

The main function, shown in Example 8–2, controls the general flow of the program.

Example 8–2: Main Program Section

```
/* This segment of CRCOPB.C contains the main function and. *
 * controls the flow of the program                          */

❶ int main ()
  {
  short rl;
  char  *r2;
  char  recbuf[80];
❷ struct desc  {
  short length;
  char  *pstring;
  };
❸ static struct desc  ofdspt[3] =
        { 0,    0,          /* Device descriptor     */
          0,    0,          /* Directory descriptor  */
          0,    0};         /* Filename descriptor   */
❹ static struct desc  ifdspt[3] =
        { 0,    0,          /* Device descriptor     */
          0,    0,          /* Directory descriptor  */
          0,    0};         /* Filename descriptor   */
❺ static char onam[] = "OUTPUT.DAT";
  static char inam[] = "INPUT.DAT";

❻ FCS$FINIT$;               /* Init file storage region */

❼ ifdspt[2].pstring = inam;
  ifdspt[2].length  = sizeof inam;
❽ FCS$OPEN$R (&fdbin, INLUN, (short *) ifdspt, (short) -1, recbuf,
            sizeof recbuf, (void (*)()) -1);
  if (fdbin.fcs$f$err == FCS$IS$SUC)
      {
❾     fdbout.fcs$f$rtyp = FCS$R$VAR;        /* Runtime initialization        */
      fdbout.fcs$f$ratt = FCS$FD$CR;
      ofdspt[2].pstring = onam;
      ofdspt[2].length  = sizeof onam;
❿     FCS$OPEN$W (&fdbout, OUTLUN, (short *) ofdspt, (short) -1, recbuf,
                sizeof recbuf, (void (*)()) -1);
      if (fdbout.fcs$f$err == FCS$IS$SUC)
          {
          for (;;)
              {                               /* Note - URBD is all set up     */
⓫             FCS$GET$(&fdbin, (char *) -1, -1, (void (*)()) -1);
              if (fdbin.fcs$f$err != FCS$IS$SUC)
                  break;
              rl = fdbin.fcs$f$nrbd;          /* rl = size of record read      */
⓬             r2 = recbuf + rl;               /* r2 = address of last byte + 1 */
              while (*(--r2) == ' ')          /* Strip trailing blanks         */
                  if (!(--rl))
                      break;

          /* At this point, rl contains the stripped size of the
           * record to be written. If the card is blank,
           * a zero length record is written.
           */
```

(continued on next page)

```
⓭                 FCS$PUT$(&fdbout, (char *) -1, rl, (void (*)()) -1);
                  if (fdbout.fcs$f$err != FCS$IS$SUC)
                      break;
                  }
⓮             if (fdbout.fcs$f$err != FCS$IS$SUC)
                  printf ("FCS error %d occurred during write\n", fdbout.fcs$f$err);
⓯             else if (fdbin.fcs$f$err != FCS$IE$EOF)
                  printf ("FCS error %d occurred during read\n", fdbin.fcs$f$err);

⓰              FCS$CLOSE$(&fdbout, (void (*)()) -1);
               if (fdbout.fcs$f$err != FCS$IS$SUC)
                   printf ("FCS error %d occurred during close of OUTPUT.DAT\n",
                       fdbout.fcs$f$err);
               }
           else
               printf ("FCS error %d occurred during open of OUTPUT.DAT\n",
                       fdbout.fcs$f$err);
           FCS$CLOSE$(&fdbin, (void (*)()) -1);
           if (fdbin.fcs$f$err != FCS$IS$SUC)
               printf ("FCS error %d occurred during close of INPUT.DAT\n",
                   fdbin.fcs$f$err);
           }
else
       printf ("FCS error %d occurred during open of INPUT.DAT\n",
           fdbin.fcs$f$err);
exit(EXIT_SUCCESS);
}
```

Key to Example 8–2:

❶ This begins the main function and the declarations of local storage. It uses automatic storage for the record buffer.

❷ Defines a structure type for a data-set descriptor.

❸ This is the output file data-set descriptor. It is defined as a structure and placed in static storage.

❹ This is the input file data-set descriptor. It is defined as a structure and placed in static storage.

❺ The output and input filenames are placed in static storage.

❻ A call to FCS$FINIT$ initializes the file storage region.

❼ This initializes the input file data-set descriptor.

❽ A call to FCS$OPEN$R opens the input file for read.

❾ These statements initialize the output FDB and the output file data descriptor.

❿ The FCS$OPEN$W macro is used to open the output file for write.

⓫ The main processing loop begins by obtaining a record using the FCSGET macro.

⓬ If a record was successfully obtained, the size of the record read is obtained from the NRBD field of the FDB. It scans backwards through the record which is in recbuf to determine the size of of the record without any trailing space characters.

⓭ With the size of the output record determined, the FCSPUT macro is used to output the record. It then loops to get the next record.

⓮ This looks into the ERR field of the FDB to see if there is an error. If there is an error, an appropriated message is displayed on the terminal.

⓯ This checks for the end-of-file.

⓰ The FCS$CLOSE$ macro closes the output file.

Chapter 9

Operating System Services and System Directives

This chapter describes operating system services and header files for the operating systems supported by PDP–11 C: RSX–11/M–PLUS, RT–11, and RSTS/E.

9.1 System Directives

The process that occurs when a task requests the Executive to perform an indicated operation is called a *system directive*. These directives control the execution and interaction of tasks and are issued as calls to subroutines contained in the system object module library.

System directives enable tasks to perform the following functions:

- Obtain task and system information
- Measure time intervals
- Perform I/O functions
- Spawn other tasks
- Communicate and synchronize with other tasks
- Manipulate a task's logical and virtual address space
- Suspend and resume execution
- Exit

For more detailed information, refer to the *RSX–11M/M–Plus Executive Reference Manual*, *RT–11 Programmer's Reference Manual*, and the *RSTS/E System Directives Manual*.

9.2 RSX System Services

The <rsxsys.h> header file defines the interface to RSX Executive Directives. PDP–11 C supports the directive names listed for FORTRAN in the *RSX-11M/M-Plus Executive Reference Manual*. Parameters are called by reference. To pass a null parameter, use –1 as the parameter.

The following example shows how to use the <rsxsys.h> header file:

```
#include <rsxsys.h>

extern void P (char *pfilename, short filename_length)

{
char      *pname;                       /* Pointer to name of file */
char      exp_name[48];                 /* Space for expanded name */
short     exp_size = sizeof exp_name;   /* Size of expanded name space */
short     exp_length;                   /* Space for returned size */
short     idsw;                         /* Directive status word */
short     rsx$fs$ndf = 010000;          /* Argument to PRSFCS */

                                        /* Expand the file name */

PRSFCS( (short *) -1, (short *) -1, (short *) -1,
        (short *) pfilename, &filename_length, (short *) exp_name,
        &exp_size, &exp_length, (short *) -1,
        (short *) -1, (short *) -1, (short *) -1,
        &rsx$fs$ndf,  &idsw);

if (idsw == 1)
        pname = exp_name;                /* Use expanded name */
else
        {                                /* Use what you have */
        pname = pfilename;
        exp_length = filename_length;
        }
}
```

9.3 RT–11 SYSLIB Routines

PDP–11 C supports the SYSLIB routines documented in the *RT–11 Programmer's Reference Manual*. The <rtsys.h> header file defines the PDP–11 C interface to the RT–11 SYSLIB functions and subroutines. These are available when PDP–11 C programs are linked with the RT–11 linker.

The interface used to call SYSLIB routines is the FORTRAN subroutine linkage. All parameters are passed by reference (see the example at the end of this section). To pass a NULL parameter via the FORTRAN subroutine linkage, use *(void *) –1* as the address of the parameter. For example:

```
some_function (a, b, (void *) -1, d);
```

Certain RT–11 library routines are unique to FORTRAN IV. They reside in FORLIB. Twelve of them are special cases since they once resided in SYSLIB until FORTRAN IV/RT–11 V2.8. The following twelve routines are documented in the *RT–11 Programmers's Reference Manual* although they are FORTRAN-dependent and are not supported by PDP–11 C.

- **GETSTR**—The <stdlib.h> function **fscanf** provides similar capabilities.
- **IASIGN**—Not supported.
- **ICDFN**—Not supported.
- **IFETCH**—The <rtsys.h> function RT$FETCH, described below, provides similar capabilities.
- **IFREEC**—Please refer to Chapter 2 for information on reserving LUNs.
- **IGETC**—Please refer to Chapter 2 for information on reserving LUNs.
- **IGETSP**—The <stdlib.h> functions **calloc** and **malloc** provide similar capabilities.
- **ILUN**—Not supported.
- **INTSET**—Not supported.
- **IQSET**—Not supported.
- **PUTSTR**—The <stdlib.h> function **printf** provides similar capabilities.
- **SECNDS**—Not supported.

PDP–11 C provides the function **RT$FETCH** to fetch device handlers. You can declare this function in the following way:

```
extern short RT$FETCH(short *__addr, short *__dnam);
```

This function simply issues a .FETCH directive. The parameters are described in the *RT–11 Programmer's Reference Manual*. The function returns a value of 1 for success, or a value of zero for failure.

The following example shows how to use the <rtsys.h> header file:

```
        /* Determine if the device is a random access device */

#include <rtsys.h>
#include <errno.h>

short afun (short *desc_block)

{
short device_block[4];                          /* Device status block */
short status;

status = IDSTAT (&desc_block[0], device_block); /* Get device info */
if (status)
    return -1;                                  /* Handler not found
                                                   in monitor tables */
```

```
if (device_block[0] & 91<(15))                /* Is it a random
                                                 access device? */
     return 1;                                /* Yes             */
else
     return 0;                                /* No              */
}
```

9.4 RSTS/E SYSLIB Routines

The <rstsys.h> header file defines the interface to the RSTS/E General Monitor Directives and supported RSX and RT–11 Emulator Directives. The first list shows the RSX Emulator Directives supported under RSTS/E; the second list shows the RT–11 Emulator Directives supported by RSTS/E.

RSX Emulator Directives

ASLUN—Assign LUN
ATRG—Attach region
CRAW—Create address window
CRRG—Create region
DTRG—Detach region
ELAW—Eliminate address window
EXIT—Task exit
EXST—Exit with status
EXTTSK—Extend task
GETLUN—Get LUN information
GETMCR—Get MCR command line
GETPAR—Get partition parameters
GETTIM—Get time parameters
GETTSK—Get task parameters
MAP—Map address window
QIO—Queue I/O request
WTQIO—Queue I/O request and wait
SUSPND—Suspend
UNMAP—Unmap address window
WFSNE—Wait for significant event
WAITFR—Wait for single event flag

RT–11 Emulator Directives

CHAIN—Chain to another program
CLOSEC—Terminate activity
GTIM—Return current time
GTJB—Return job information
GTLIN—Return line of input

LOOKUP—Lookup associate channel with device
PRINT—Print output string to console
PURGE—Deactivate channel
RCTRLO—Reset the console (CTRL/O)
SCCA—Provide CTRL/C intercept

Table 9–1 shows the functions, macro definitions, and structure definitions that assist in accessing the FIRQB and XRB data structures. The functions **RSTS$FIRQB** and **RSTS$XRB** take no arguments and return no values.

Table 9–1: FIRQB and XRB Data Structures

Use	FIRQB	XRB
Address definition macro	**RSTS$FIRQB**	RSTS$XRB
Structure definition	FIRWB	XRB
Clear structure function	void **RSTS$CLRFQB**(void)	void **RSTS$CLRXRB**(void)

Refer to the *RSTS/E System Directives Manual* for more information.

9.5 Qualifications on Using the TIME, EXIT, and ABORT Functions

When you reference the functions **time**, **exit**, or **abort**, you must take in consideration which system you are using and if there are conflicting symbols assigned to these functions. The following chart shows which symbols reference conflicting headers:

External Symbol	Conflicting Headers
time	<time.h> and <rtsys.h>
exit	<stdlib.h> and <rstsys.h>
exit	<stdlib.h> and <rsxsys.h>
abort	<stdlib.h> and <rsxsys.h>

To resolve these conflicts, simply include the appropriate system interface header file (<rtsys.h>, <rstsys.h>, or <rsxsys.h>) prior to including the conflicting standard header file <time.h> or <stdlib.h>). If you do not need access to the SYSLIB versions of these functions, no further action is necessary.

If you need access to the SYSLIB version of these functions, you must specify the SYSLIB symbol in upper case (TIME, EXIT, or ABORT), and you must explicitly include SYSLIB in your link before the PDP–11 C Run-Time Library. When you want to use both the PDP–11 C standard RTL symbol and the corresponding SYSLIB symbol, specify the PDP–11 C symbol in lower case (time, exit, abort). The following example illustrates this:

```
/* MYFILE.C */
#include <rtsys.h>
#include <time.h>
int main (void)
{
      char      time_string[8];
      time_t    since_1970;
      TIME      (time_string): /* Call the RT--11 SYSLIB TIME() function */
      time      (&since_1970); /* Call the PDP--11 C time() function */
}
```

As described previously, if you wish to use the TIME symbol in the RT–11 SYSLIB, you must explicitly include SYSLIB in a fashion similar to the following:

```
R LINK
FOO=FOO,SY:SYSLIB,CC:CEISRT/B:3000/M:3000
```

If you want to use the EXIT or ABORT symbols in the RSX SYSLIB or use the EXIT symbol in the RSTS/E SYSLIB, you must explicitly include SYSLIB in your link before PDP–11 C RTL in a fashion similar to the following:

```
             .ROOT USER
USER:        .FCTR FOO-LIBR
LIBR:        .FCTR LB:[1,1]SYSLIB/LB-LB:[1,1]CEISRSX/LB
             .END
```

Chapter 10

Linkages Supported by PDP–11 C

This chapter describes the linkages supported by PDP–11 C, as well as the register and stack usage during procedure calls.

The term linkage defines the exact internal calling mechanism used for function calls. A function may be assigned a linkage using the **#pragma** linkage directive. PDP–11 C supports the following linkages:

- PDP–11 C
- PDP–11 FORTRAN–77
- PDP–11 Pascal
- RSX AST
- RSX SST
- RSX CSM

For more information on the **#pragma** linkage directive, refer to the *Guide to PDP–11 C*.

The following sections show the details of the internal calling mechanisms including stack and register usage of the six linkages. Table 10–1 summarizes the register usage for the linkages supported by PDP–11 C.

The following sections describe the actions both the calling and the called function must take to use each linkage. This information is important if either the calling or called function is written in a language other than PDP–11 C. The PDP–11 C compiler will always take the correct action for each linkage.

Table 10–1: Register Usage for PDP–11 C-Supported Linkages

Linkage	Called-Function Actions	Calling-Function Actions
C	Saves registers used by the called function with the exception of R1 and F1.	Removes parameters after return.
FORTRAN	None.	Removes parameters after it returns. Saves registers before call. Restores registers after call.
Pascal	Saves registers. Removes parameters before return. Cannot be used with variable arguments.	None.
RSX SST	Saves and restores used registers. Removes trap-dependant parameters before returning. Returns by executing an RTI	Not callable.
RSX AST	Saves and restores used registers. Removes trap-dependant parameters before returning. Returns by executing an ASTX$ directive.	Not callable.
RSX CSM	Same as C linkage, but allows C function to be placed in a supervisor-mode library.	Removes parameters after return.

10.1 PDP–11 C Linkage

When a function is called by the C linkage, it receives the argument block shown in Figure 10–1. The values of all registers used by the function, with the exception of R1 and F1, must be saved before their use and restored before the function returns.

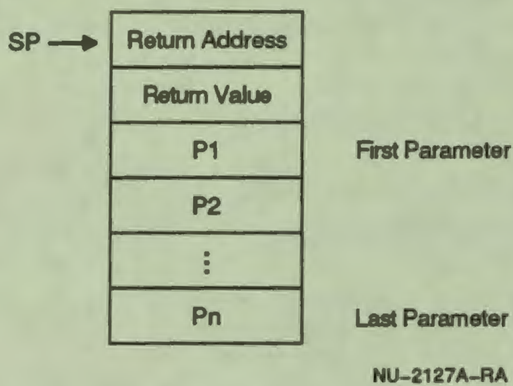
The calling function must create the argument block shown in Figure 10–1 and save the values of R1 and F1, but need not save the values of any other registers.

The return value is on the top of the stack when the call returns. For example, if a **short int** is being returned, the word at the top of the stack contains the return value. If a **struct** is being returned, the top of the stack will contain enough space to hold the structure being returned. The calling function should move the return value to an appropriate location and then remove the parameters from the stack.

Parameters are referenced by way of the Stack Pointer (SP); registers R0 through R5 can be used by the called function for other purposes. Functions that are declared with the C linkage can receive a variable number of parameters because the function's first parameter is the one closest to the top of the stack.

Functions that use the PDP–11 C Standard Library variable arguments (<stdarg.h>), and functions whose address is used, must be declared with C linkage.

Figure 10–1: Stack Usage Using C Linkage



10.2 FORTRAN Linkage

The FORTRAN linkage uses general register R5 to identify the parameters passed to a function. See Figure 10–2 for the detail of this mechanism.

It is unnecessary for a function that is called by FORTRAN linkage to save any registers that it uses. Return values are located as follows:

- R0, 1-word value
- R0, R1, 2-word values
- R0, R1, R2, R3, 4-word values

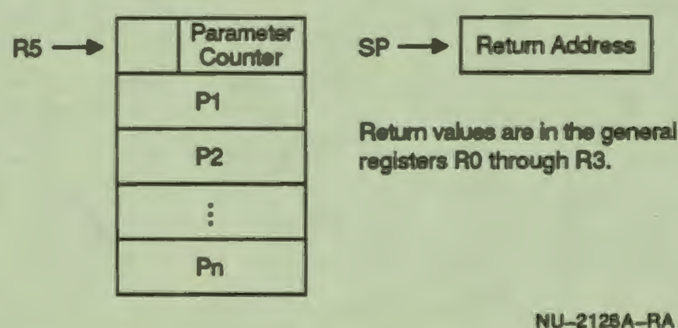
It is impossible to return larger values by using the FORTRAN linkage.

When a function is called by the FORTRAN linkage, the calling function must set the R5 parameter list as shown in Figure 10–2 and save any registers it needs to preserve across the call. R5 cannot be used for other purposes because it is reserved as an argument pointer.

PDP–11 C uses a jacket routine to call the FORTRAN function rather than calling a FORTRAN linkage function directly. The overhead of the jacket routine makes calling a FORTRAN linkage function from C less efficient than calling a C or Pascal linkage function.

The advantage of using the FORTRAN linkage is that a function declared with the FORTRAN linkage may not have the restrictions that a function declared with the C or Pascal linkage has because its parameters are referenced by way of R5 and not the top of the stack. For example, a function placed in a nondefault cluster library cannot reference its parameters by way of the top of the stack; therefore, a routine that is to be placed in a nondefault cluster library must be declared with the FORTRAN linkage. For more information on nondefault cluster libraries, see the appropriate task builder reference manual.

Figure 10–2: Register 5 Usage Using FORTRAN Linkage



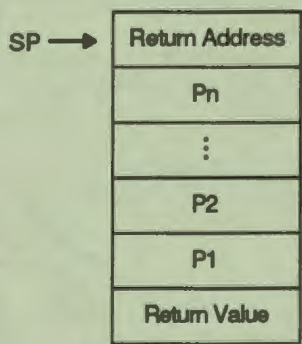
10.3 Pascal Linkage

When using the Pascal linkage to call a function, the calling function must create the argument block shown in Figure 10–1. It is not necessary for the calling function to save any of the registers. The return value is on the top of the stack when the call is returned. The calling function does not have to clear the stack because it is done by the called function.

A function called using the Pascal linkage receives the argument block shown in Figure 10–3. The values of any registers used, including R1 and F1, must be saved prior to their use and restored with the values at the end of the call. Before returning, a function declared using the Pascal linkage removes the parameters from the top of the stack.

Pascal linkage cannot pass a variable number of arguments to a function; however, it can efficiently remove parameters from the stack rather than force the calling function to remove them. If the same function is called from several different locations, the code to remove the parameters appears only once in the called function. However, using C linkage, the code to remove the parameters appears after every call site.

Figure 10–3: Stack Usage Using Pascal Linkage



NU–2129A–RA

10.4 RSX AST And SST Linkages

The RSX AST and RSX SST linkage allow the programmer to write an AST or SST trap handler in PDP–11 C. This functionality should only be used by those programmers with a solid knowledge of trap handlers. Before writing any trap handlers in PDP–11 C, please read the appropriate operating system manuals carefully.

Functions with these linkages may be declared or have their addresses taken. Any other use of these functions will be flagged as an error by the compiler. Furthermore, all functions declared to have linkage RSX AST or RSX SST must be of type **void** and their parameters must be of size **int**.

The PDP–11 C functions which are declared with the AST and SST linkages have an additional restriction placed on them. PDP–11 C does not support calling PDP–11 C library functions from a trap handling function. While it may be possible to call certain library functions, others can not be called. Since it is very difficult to determine which functions are safe, PDP–11 C does not support any of these calls.

RSX AST Linkage

The RSX AST linkage is used to declare a function to be an RSX AST trap handler. A function is declared as an RSX AST linkage function in the following manner:

```
#pragma linkage rsx_ast <name>
void <name> ( int <efmw>, int <ps>, int <pc>, int <dsw> [,...] );
```

A RSX AST linkage function has a minimum of four parameters. The first parameter is the event-flag mask word. The second parameter is the Processor Status Word. The third parameter is the PC. The fourth parameter is the Directive Status Word. Any other parameters are specific to the type of AST the function is expected to handle. For more information see the *RSX–11M/M–PLUS and Micro/RSX Executive Reference Manual*.

When an RSX AST linkage function executes a return, any parameters following the <dsw> will be automatically removed from the stack, and an ASTX$S directive will be executed.

RSX SST Linkage

The RSX SST linkage is used to declare a function to be an RSX SST trap handler. A function is declared as an RSX SST linkage function in the following manner:

```
#pragma linkage rsx_sst <name>
void <name> ( int <ps>, int <pc> [,...] );
```

A RSX SST linkage function has a minimum of two parameters. The first parameter is the Processor Status Word. The second parameter is the PC. Any other parameters are specific to the type of SST the function is expected to handle. For more information see the *RSX–11M/M–PLUS and Micro/RSX Executive Reference Manual*.

When an RSX SST linkage function executes a return, any parameters following the <pc> will be automatically removed from the stack, and an RTI will be executed.

10.5 The RSX CSM Linkage

The RSX CSM linkage allows the programmer to place a C function in a supervisor-mode resident library. Because the default C linkage places its parameters on the top of the stack, functions which use the C linkage can not be placed in a supervisor-mode resident library. By using the CSM linkage, the compiler adjusts its parameter references to account for the four words of overhead created when the function is placed in a supervisor-mode library.

Placing a PDP–11 C function in a supervisor-mode library is an advanced programming practice. This should only be attempted by those programmers who have created supervisor-mode libraries in the past. Of special note, only those functions declared with an RSX CSM linkage should be included in the symbol table of the resident library. All other global symbols, especially PDP–11 C OTS routines included in the library, must be globally excluded from the symbol table when the library is built.

The syntax of an RSX CSM function is identical to those with the default C linkage. It is simply necessary to use the **#pragma** linkage **rsx_csm** directive before the function is declared.

It is not possible to invoke a function which is declared to take this linkage. Functions with this linkage may be declared or have their addresses taken. Any other use of these functions will be flagged as an error by the compiler.

10.6 Linkages and Other Languages

Any C function may be assigned the C, FORTRAN, or PASCAL linkages following the guidelines discussed in the previous sections. A linkage may be assigned to a function declared within a module or to an external function called by the function in the module. When a linkage is assigned to a function, all calls to that function must declare the function using the same linkage.

Not all PDP–11 programming languages are able to assign specific linkages to functions written or called in the language being used. For example, an application written in FORTRAN–77 can only be called using the FORTRAN linkage and can only call other functions that use the FORTRAN linkage. The FORTRAN linkage is used by the following PDP–11 languages: FORTRAN–77, BASIC–PLUS–2, and COBOL–81. The Pascal linkage is used by PDP–11 Pascal. See Section 10.8 for other restrictions.

PDP–11 C can call or be called from other languages because it allows the use of different linkages. When C functions are called from another language, the C program must define those functions to use the linkage required by that language. A PDP–11 C program calling a function written in another language must assign the proper linkage to the external definition of that function. Consider the following examples:

```
/*A fortran application calls CFUNCT, a function written in C.*/
#pragma linkage fortran CFUNCT
<ctype> CFUNCT([<params>])
{
     <body of function>
}
```

```
/*A program written in C calls FORFUN, a function written in FORTRAN.*/
#pragma linkage fortran FORFUN
extern <type> FORFUN(<params>);
```

10.7 Data Sharing with Fortran and BP2

In addition to sharing data through passed parameter values, you can allow a subprogram written in PDP–11 C to access data declared in either a Fortran common area or a BP2 mapped region.

The two examples in this section show the declaration of Fortran and BP2 external data. Both examples contains a 16-bit integer, a 32-bit integer, and a single precision floating point variable.

If the PDP–11 C subprogram wishes to access the declared FORTRAN or BP2 external variables, it must use the **#pragma psect** directive. The **#pragma psect** directive provides the mapping into the FORTRAN or BP2 common data area and should be declared with the same psect attributes as the FORTRAN or BP2 data area. You can determine the psect attributes of the data area from a map file produced by the linker.

The C declarations shown in each example give PDP–11 C a mapping into the data area. Any modifications to these variables within the PDP–11 C subprogram or the FORTRAN or BP2 subprogram can be seen by both subprograms.

The following example shows the declaration of a Fortran data area.

F77 data area

```
INTEGER*2    ICOUNT
INTEGER*4    LCOUNT
REAL*4       RTYPE

COMMON /BLOCK1/ICOUNT,LCOUNT,RTYPE
```

C data area

```
#pragma psect static_rw BLOCK1 rel,d,gbl,rw,ovr
static   short icount;
static   long  lcount;
static   float rtype;
#pragma psect static_rw
```

The following example shows the declaration of a BP2 mapped region.

BP2 data area

```
COMMON (BLOCK1) word ICOUNT, long LCOUNT, single RTYPE
```

C data area

```
#pragma psect static_rw BLOCK1 rel,d,gbl,rw,ovr
static   short icount;
static   long  lcount;
static   float rtype;
#pragma psect static_rw
```

10.8 Restrictions and Notes

The following list notes and explains the existing exceptions to using PDP–11 C with other languages:

- Only the parameter-passing mechanisms are supported.

  Certain language features may not work when called either directly or indirectly from PDP–11 C. The reason is twofold: initializations required for those features are not done by PDP–11 C, or the language feature may attempt to use memory already allocated by PDP–11 C.
- Users should not change the contents of the C OTS work area (PSECT $$C).
- When PDP–11 C is called from other languages, whether directly or indirectly, many of the Standard Library functions will not work for the previously mentioned reasons.

- If a call to the routine C$INIT is made before the first invocation of a C function and the routine C$FINI is called after the last invocation to a C function, some Standard Library functions may work. The routines C$INIT and C$FINI perform a number of initializations and clean-up routines for the Standard Library functions.
- In general, when mixing C with another high-level language such as FORTRAN-77 or BASIC-PLUS-2, the main program must be in the other high-level language.
- PDP-11 C parameters are always passed and received by value.

  To pass a variable to a routine which expects to receive a parameter by reference, pass the variable's address using the C "&" operator. For example, FORTRAN passes and receives parameters by reference. To pass an integer variable "foo" to a FORTRAN routine from a C routine, the C routine must use "&foo" which is the address of the variable, not "foo", the variable itself.

  To pass an integer parameter from a FORTRAN routine to a C routine, the C routine receives the address of the parameter not the parameter itself. The parameter should be declared by the C function as a pointer to an int (int *foo). The "*" operator is used to access the actual value (*foo).
- Complex parameters

  When calling between languages, use only integer and floating-point parameters. Use other data types only after careful investigation, because not all languages support all C types.
- Other high level languages may have their own restrictions that prevent them from calling or being called by PDP-11 C.

Reference Section

This reference section describes the functions and macros contained in the PDP–11 C Run-Time Library. For each function and macro, you will find an overview, the function or macro format, descriptions of the arguments, a detailed description of the function or macro if more information is needed beyond what is given in the overview section, and return values.

The Reference Section is divided into three parts:

1. PDP–11 C Standard Library Macros and Functions
2. FCS Extension Library Macros
3. RMS Extension Library Macros

Within each of these parts, the functions and macros appear in alphabetical order.

1 PDP–11 C Standard Library Macros and Functions

abort

The **abort** function causes the program to terminate immediately.

Format

#include <stdlib.h>

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

Arguments

None.

Description

The **abort** function raises the SIGABRT signal and returns the EXIT_FAILURE completion code to the operating system. PDP–11 C attempts to flush or close any open output streams.

Return Values

None.

abs

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

Format

#include <stdlib.h>

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

Arguments

x
Is an integer expression.

Return Values

Returns the absolute value of x.

acos

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

Format

#include <math.h>

double acos (double *x*);

Arguments

x
Is the cosine of the angle.

Description

When $|x| > 1$, the value of acos(x) is 0, and the **acos** function sets *errno* to EDOM.

Return Values

Returns the arc cosine of *x* in radians.

__alr50

The **__alr50** function converts the first six characters of the input string to an unsigned 32-bit integer corresponding to the radix-50 translation.

Format

#include *<stdlib.h>*

short int __alr50 *(***char** **__ascii_string,* **unsigned long int** **__rad50_string);*

Arguments

__ascii_string
Is a pointer to a six-character ASCII string to convert. The string does not have to be a NUL terminated string.

__rad50_string
Is a pointer to an unsigned long integer to receive the converted radix-50 string.

Return Values

Non-zero value	Indicates success.
Zero value	Indicates an error has occurred.

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
```

Each field has a constant width.

Format

#include *<time.h>*

char *asctime (const struct tm ***timeptr*);**

Arguments

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

Description

The **tm** structure is defined in the <time.h> header file as follows:

```
struct tm
{
   int  tm_sec,      /* seconds after the minute  -- [ 0, 60 ] */
        tm_min,      /* minutes after the hour    -- [ 0, 59 ] */
        tm_hour,     /* hours since midnight      -- [ 0, 23 ] */
        tm_mday,     /* day of the month          -- [ 1, 31 ] */
        tm_mon,      /* months since January      -- [ 0, 11 ] */
        tm_year,     /* years since 1900          -- [ 0, .. ] */
        tm_wday,     /* days since Sunday         -- [ 0,  6 ] */
        tm_yday,     /* days since January 1      -- [ 0,365 ] */
        tm_isdst;    /* Daylight Saving Time Flag -- [-1,  1 ] */
                     /* -1 info. not available                 */
                     /*  0 D.S.T. IS-NOT in effect             */
                     /*  1 D.S.T. IS     in effect             */
}
```

The **asctime** function converts the contents pointed to by *timeptr* into a 26-character string, as shown in the previous example, and returns a pointer to the string. Subsequent calls to **asctime** or **ctime** point to the same static string, which is overwritten by each call.

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 argument.

Format

#include *<math.h>*

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

Description

When $|x| > 1$, the value of asin(x) is 0, and the **asin** function sets *errno* to EDOM.

Return Values

Returns the arc sine of x in radians.

__asr50

The **__asr50** function converts the first three characters of the input string to an unsigned 16-bit integer corresponding to the radix-50 translation.

Format

#include *<stdlib.h>*

short int __asr50 *(***char** **__ascii_string,* **unsigned short int** **__rad50_string);*

Arguments

__ascii_string
Is a pointer to a three-character ASCII string to convert. The string does not have to be NUL terminated.

__rad50_string
Is a pointer to an unsigned short integer to receive the converted radix-50 string.

Return Values

Non-zero value	Indicates success.
Zero value	Indicates an error has occurred.

assert

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

Format

#include *<assert.h>*

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

Arguments

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

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. This information is written on the standard error file in an implementation-defined format and includes the following: 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 functions __FILE__ and __LINE__. Then, the **assert** macro calls the **abort** function.

The **assert** macro writes a message in the following form:

```
assert error:expression= in file (filename), at line nnn.
```

where *expression* is the string equivalent of the expression in the user's code.

If *expression* is true (that is, evaluates to nonzero), the **assert** function has no effect.

Compiling with the command qualifier /DEFINE=NDEBUG or with the preprocessor directive **#define** NDEBUG ahead of the **#include** <assert.h> statement causes the **assert** function to have no effect.

The **assert** function is implemented as a macro, not as a function. If you use **#undef** to remove the macro definition, the behavior is undefined.

Return Values

None.

atan

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

Format

#include *<math.h>*

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

Arguments

x
Is the tangent of the angle.

Return Values

Returns the arc tangent of *x* in radians.

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.h>*

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

Arguments

y
Is an expression of type **double**.

x
Is an expression of type **double**.

Return Values

Returns the arc tangent of y/x in radians.

atexit

The **atexit** function registers a function that will be called at normal program termination.

Format

#include *<stdlib.h>*

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

Arguments

func
Is a pointer to the function to be registered.

Description

Up to 32 functions can be registered. When a registered function is called, it is called without arguments. When the program exits, the registered functions are called in the reverse order from which they were registered.

Return Values

0	Indicates that the registration has succeeded.
Nonzero	Indicates registration failed.

atof

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

This function recognizes an optional sequence of "white-space" characters (as defined by **isspace** in <ctype.h>), 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. See also **strtod**.

Format

#include *<stdlib.h>*

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

Arguments

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

Description

The function call **atof(str)** is equal to **strtod(str,(char **)0)**, arithmetic exceptions notwithstanding.

- If the correct value causes an overflow, HUGE_VAL is returned and *errno* is set to ERANGE.
- If the correct value causes an underflow, 0 is returned and *errno* set to ERANGE.

See also **strtod**.

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.h>*

int atoi (const char **nptr*);

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

Arguments

nptr
Is a pointer to the character string to be converted to **int (atoi)** or **long (atol)**.

Description

The **atoi** and **atol** functions account for overflows resulting from the conversion. Truncation from **long** to **int** can take place upon assignment or by an explicit cast (arithmetic exceptions notwithstanding). 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)**.

See also **strtol**.

Return Values

n	Indicates the converted value.

bsearch

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

Format

#include *<stdlib.h>*

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-void.

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-void.

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.

Because the **bsearch** function is declared as type "pointer-to-void", the returned value must be cast or assigned into a specified pointer-to-object type.

Return Values

x	Indicates a pointer to the matching member of the array.
NULL	Indicates that the key cannot be found in the array.

cabs

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

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

This function is provided for compatibility with VAX C and is only available if compiled with the /NOSTANDARD switch.

Format

#include *<math.h>*

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

Arguments

z
Is a structure of type **cabs_t**.

Description

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

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

Return Values

Returns the square root of the sum of the squared arguments x and y.

calloc

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

Format

#include *<stdlib.h>*

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 0s.

See also **malloc** and **realloc**.

Return Values

NULL	Indicates an inability to allocate the space.
x	Indicates the address of the first byte.

ceil

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

Format

#include *<math.h>*

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

Description

The **ceil** function computes the smallest integer value that is not less than *x*.

Return Values

Returns the smallest integer value, not less than *x*, expressed as a **double**.

clearerr

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

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Points to a file.

Description

The **clearerr** function clears the end-of-file and error indicators for the file pointed to by the file pointer.

Return Values

None.

clock

The **clock** function determines the elapsed processor time used since the beginning of the program execution.

Format

#include *<time.h>*

clock_t clock *(void);*

Description

The value returned by the **clock** function must be divided by the value of the macro **CLOCKS_PER_SEC**, as defined in the <time.h> header file, to obtain the time in seconds.

Return Values

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

cos

The cos function returns the cosine of its radian argument.

Format

#include *<math.h>*

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

Arguments

x
x is an object of type **double**.

Return Values

Returns the cosine value of *x*.

cosh

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

Format

#include *<math.h>*

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

Arguments

x

x is an object of type **double**.

Return Values

Returns the hyperbolic cosine value of *x*.

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.h>*

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

Arguments

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

Description

Successive calls to **ctime** overwrite any previous time values. The type time_t is defined in the <time.h> header file as follows:

```
typedef long int time_t
```

Return Values

Pointer	Points to the 26-character ASCII string.

difftime

The **difftime** function computes the difference in seconds between the two times specified by the time0 and time1 arguments.

Format

#include *<time.h>*

double difftime *(***time_t** *time1,* **time_t** *time0);*

Arguments

time1
Is of type time_t, which is defined in the <time.h> header file.

time0
Is of type time_t, which is defined in the <time.h> header file.

Description

The **difftime** function subtracts **time1** from **time0** to compute the difference between two calendar times.

Return Values

n	Indicates the difference 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.h>*

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.h> header file as follows:

```
typedef struct
           {
            int quot
            int rem;
           }
            div_t;
```

Return Values

Returns a structure of type **div_t** which contains the quotient and remainder of numer/denom.

exit

The **exit** function terminates the program.

Format

#include *<stdlib.h>*

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

Arguments

status
The argument is passed to the operating system when the program exits. EXIT_SUCCESS and EXIT_FAILURE are defined in the <stdlib.h> header file as values for success and failure.

Description

The **exit** function terminates the program and returns the value in status to the operating system. It also calls functions registered with **atexit**, flushes and closes streams, and deletes **tmpfile** files.

Return Values

None.

exp

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

Format

#include *<math.h>*

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

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.h> header file is defined to be the largest possible floating-point value.

Return Values

Returns the exponential value of the argument. If an overflow occurs, **exp** returns the largest possible floating-point value.

fabs

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

Format

#include *<math.h>*

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

Description

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

Return Values

Returns the absolute value of the argument.

__fbuf

The **__fbuf** function returns the current buffer length associated with a file pointer.

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a file pointer.

Description

The **__fbuf** function retrieves the current buffer length that has been associated with a previously allocated file pointer.

Return Values

Nonzero value	Indicates success.
Zero value	Indicates an error has occurred

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.h>*

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 called automatically for all open files.

Return Values

0	Indicates success.
EOF	Indicates that the buffered data cannot be written to the file, or the file control block is not associated with an open file. EOF is a preprocessor constant defined in the <stdio.h> header file.

feof

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

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a file pointer.

Return Values

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

ferror

The **ferror** function returns a nonzero integer if an error occurs during a read or write operation.

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a file pointer.

Description

A call to the **ferror** function continues to return this indication until the file is closed or until the **clearerr** function is called.

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.h>*

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

Arguments

file_ptr
Is a file pointer.

Description

If the file_ptr is NULL, all files open for output are flushed.

Return Values

0	Indicates that the operation is successful.
EOF	Indicates that an error occurred in writing out the data. (EOF is a preprocessor constant defined in the <stdio.h> header file.)

__fger

The **__fger** function returns the low level error code that is associated with a previously called file operation.

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a file pointer.

Description

The **__fger** function returns the underlying file system's error code that was associated with a previously called file operation.

Return Values

Returns the underlying file system's error code.

fgetc

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

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a pointer to the file to be accessed.

Description

The **fgetc** function gets the next character pointed to by the file pointer from the input stream and advances the file indicator for that file.

Return Values

EOF	Indicates end-of-file or error. (EOF is a preprocessor constant defined in the <stdio.h> header file.)
x	Indicates the character returned.

fgetpos

The **fgetpos** function stores the file position indicator.

Format

#include *<stdio.h>*

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

Arguments

str
Is the stream whose file position indicator value is desired.

pos
Is the location where the file position indicator for *str* is stored.

Description

The **fgetpos** function finds the current value of the file position indicator for a stream and stores it in a variable of type **fpos_t** pointed to by *pos*.

Return Values

0	Indicates success.
Nonzero	Indicates failure. A positive value is stored in *errno*.

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 or end of file, whichever comes first. The function stores the string in the *str* argument. The **fgets** function terminates the line with a NUL (\0) character.

Format

#include *<stdio.h>*

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

Arguments

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

maxchar
Specifies one character greater than the maximum number of characters to fetch.

file_ptr
Is a file pointer.

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.h> header file to be the NULL pointer value.

__fgnm, fgetname

The **__fgnm** or **fgetname** function returns a pointer to a file specification associated with a file variable.

Format

#include *<stdio.h>*

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

or

char * __fgnm *(***FILE** **pfile,* **char** ** buffer, ...);*

Arguments

pfile_ptr
Is a pointer to a file which has been previously opened.

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

. . .
Represents an optional additional argument for VAX C compatibility. PDP–11 C ignores this argument.

Description

The **__fgnm** or **fgetname** function places the file specification at the address given in buffer and returns the address of the buffer. The buffer should be an array large enough to contain a fully qualified file specification. When an error occurs, **fgetname** or **__fgnm** returns 0.

The function name, **fgetname**, is provided for compatibility with VAX C, but the name is not compatible with the ANSI Standard. Therefore, the function is not provided when compiling /STANDARD=ANSI.

The function **__fgnm** is ANSI compatible and is defined when the compile time switch /STANDARD=ANSI is used.

Return Values

x	Indicates the character string returned for the file specified.
NULL	Indicates that an error has occurred.

floor

The **floor** function returns the largest integer that is less than or equal to its argument.

Format

#include *<math.h>*

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

Arguments

x
Is a real value.

Description

The **floor** function returns a **double** which represents the largest integer that is less than or equal to the number given as the argument to the function.

Return Values

Returns the largest integer that is less than or equal to its argument.

__flun

The **__flun** function returns the logical unit number associated with a file pointer.

Format

#include *<stdio.h>*

int __flun *(***FILE** **file_ptr);*

Arguments

file_ptr
Is a file pointer.

Description

The **__flun** function retrieves the logical unit number (LUN) from a previous allocated file pointer and returns this value to the requesting routine.

Return Values

Zero value	Indicates that an error has occurred.
1 – 255	Indicates success.

fmod

The **fmod** function computes a floating-point remainder.

Format

#include *<math.h>*

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

Arguments

x
Is a real value.

y
Is a real value.

Description

The **fmod** function computes the floating-point remainder of the first argument to **fmod** divided by the second. If *y* is 0, the **fmod** function returns 0 and sets *errno* to EDOM.

Return Values

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.

Format

#include *<stdio.h>*

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 following character strings:

- "r" opens text file for read
- "w" opens text file for write
- "a" appends to a text file
- "rb" opens binary file for read
- "wb" opens a binary file for write
- "ab" appends to a binary file
- "r+" opens a text file for update
- "w+" writes a text file for update
- "a+" appends to a text file
- "r+b" or "rb+" opens a binary file for update
- "w+b" or "wb+" writes binary file for update
- "a+b" or "ab+" appends to binary file

The access modes have the following effects:

- "r" opens an existing file for reading.
- "w" creates a new file and opens it for writing. On RSX systems, if the file already exists, a new file is created with the same name and a higher version number.
- "a" opens the file for append access. An existing file is positioned at end-of-file, and its data written to the end-of-file. If the file does not exist, it will be created.

NOTE

> The **setvbuf** function should be used to set the buffer size to a multiple of 512 when opening an existing file for append if any record that is to be written to the file has a size of 512 bytes or greater.

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 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 end-of-file (writing). If the file does not exist, the PDP–11 C Run-Time library creates it.
- "b" is binary access mode. No conversion of carriage control information is attempted.

Description

When the mode string contains "+" or "b", the file opens in binary mode; otherwise, it opens in text mode. For example, "a+" mode opens a file for append/binary mode even if the file would otherwise be treated as a text file.

Though update mode allows both reading and writing to the same stream, there are certain restrictions. Output may not be directly followed by input without an intervening call to the **fflush** function or to the file positioning functions **fseek, fsetpos**, or **rewind**. Input may not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

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

Up to FOPEN_MAX files may be opened simultaneously.

See also **freopen**.

Return Values

File pointer	Points to an object of type **FILE** which identifies the open file to other Standard Library functions.
NULL	Indicates an error. The constant NULL is defined in the <stdio.h> header file 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, and failure to open the specified file.

fprintf

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

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a pointer to the file to which you direct 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, the output sources may be omitted; otherwise, the function calls must have exactly as many optional expressions as there are conversion specifications, and the conversion specifications must match the types of the optional expressions. Conversion specifications are matched to optional expressions in simple left-to-right order. Refer to the Section 2.4.2 for more information.

Description

An example of a conversion specification follows:

```
#include <stdio.h>
int main()
{
   int  temp = 4, temp2 = 17;

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

Sample output (to the file *stdout*) from the previous example is as follows:

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

Return Values

Negative number	Indicates an error has occurred.
Number of characters transmitted	Indicates success.

fputc

The **fputc** function writes a single character to a specified file.

Format

#include *<stdio.h>*

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

Arguments

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

file_ptr
Is a pointer to the file where the character is written.

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 PDP–11 C, **putc** and **fputc** are functionally equivalent.

See also **putc**.

Return Values

EOF	Indicates that an output error has occurred. EOF is defined in the <stdio.h> header file.
Character	Indicates success.

fputs

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

Format

#include *<stdio.h>*

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

EOF	Indicates an error has occurred.
Number of characters written	Indicates success.

fread

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

Format

#include *<stdio.h>*

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 the information being read will be placed.

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 to the file from which the items are to be read.

Description

The type **size_t** is defined in the <stdio.h> header file. 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.

Return Values

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

__frec

The **__frec** function returns the current record length associated with a file pointer.

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a file pointer.

Description

The **__frec** function retrieves the current record length that has been associated with a previously allocated file pointer.

Return Values

Zero value	Indicates that an error has occurred.
Nonzero value	Indicates success.

free

The **free** function releases for relocation the area allocated by a previous **calloc**, **malloc**, or **realloc** call.

Format

#include *<stdlib.h>*

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

Arguments

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

Description

The contents of the deallocated area should not be used by the user program after it has been freed.

Return Values

None.

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.h>*

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 file specification. After the function call, the given file pointer is associated with this file.

a_mode
Is an access mode indicator. See **fopen** for additional information on the access mode indicator.

file_ptr
Is a file pointer which points to a previously opened file.

Description

The **freopen** function closes the file pointed to by *file_ptr* and opens the file named by *file_spec*. Use the **freopen** function to associate *stdin*, *stdout*, or *stderr* with a file.

Return Values

File pointer	Indicates success.
NULL	Indicates that an error has occurred. The constant NULL is defined in the <stdio.h> header file to be the NULL pointer value.

frexp

The **frexp** function converts a floating point number into a normalized fraction and an integral power of 2.

Format

#include *<math.h>*

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

Arguments

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

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

Description

The expression given for *value* is broken into a normalized function which is returned as the return value of the function, and an integral power of 2 which is placed in the **int** pointed to by *eptr*.

Return Values

The mantissa of *value* with a magnitude less than 1.

fscanf

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

Format

#include *<stdio.h>*

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, refer to Chapter 2.

. . .
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 simple left-to-right order.

fscanf

Description

An example of a conversion specification follows:

```
#include <stdio.h>
main ()
{
   int   temp, temp2;

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

NOTE

A common programming error is to omit the ampersand (&) of &temp in line 4 of the program. If the ampersand is omitted, the address is not passed.

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

```
4 17
```

Sample input from the previous example is as follows:

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

Return Values

x	Indicates the number of successfully matched and assigned input items.
EOF	Indicates that the end-of-file has been encountered before any conversions. EOF is a preprocessor constant defined in the <stdio.h> header file.

fseek

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

Format

#include *<stdio.h>*

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 (SEEK_CUR), forward from the beginning of the file (SEEK_SET), or backwards from the end-of-file (SEEK_END).

Description

The **fseek** function sets the file position of the stream specified by file_ptr.

For binary streams, if the direction is SEEK_SET, the position is measured in bytes from the beginning of the file. If the direction is SEEK_CUR, the position is measured from the current position in the file.

For text streams, the offset should either be zero or a value returned by an earlier call to **ftell**. In all cases, direction shall be SEEK_SET.

PDP–11 C does not support the direction value of SEEK_END.

A successful call to **fseek** clears the end-of-file and undoes any effects of the **ungetc** function.

Return Values

0	Indicates successful seeks.
EOF	Indicates improper seeks. EOF is a preprocessor constant defined in the <stdio.h> header file.

fsetpos

The **fsetpos** function sets the current file position indicator. The position must be specified by using a value returned by the **fgetpos** function.

Format

#include *<stdio.h>*

int fsetpos (**FILE** **file_ptr*, **const fpos_t** **pos*);

Arguments

file_ptr
Is a pointer to a file.

pos
Is a pointer to the file position indicator value obtained from a previous call to the **fgetpos** function.

Return Values

Zero value	Indicates success. A successful call clears the end-of-file and undoes any effects of the **ungetc** function.
Nonzero value	Indicates an error has occurred.

ftell

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

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a file pointer.

Description

The **ftell** function returns the current position in the stream pointed to by file_ptr.

For a binary stream, the value returned is the number of bytes from the beginning of the file.

For a text stream, the value returned is information which is only usable by the **fseek** function for returning the file to the current position.

Return Values

EOF	Indicates an error has occurred.
x	Current value of the position indicator.

fwrite

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

Format

#include *<stdio.h>*

size_t fwrite *(***const void** **ptr,* **size_t** *size,* **size_t** *nmemb,)*
FILE **file_ptr ;*

Arguments

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

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

nmemb
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

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

The type size_t is defined in the <stdio.h> header file.

Return Values

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

getc

The **getc** function returns a character from a specified file.

Format

#include *<stdio.h>*

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

Arguments

file_ptr
Is a pointer to the file to be accessed.

Description

The **getc** function gets the next character pointed to by the file pointer from the input stream and advances the file indicator for that file.

Return Values

x	Indicates the next character as an **int** from the specified file.
EOF	Indicates the end-of-file or an error. (EOF is a preprocessor constant defined in the <stdio.h> header file.)

getchar

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

Format

#include *<stdio.h>*

int getchar *(void);*

Description

The **getchar** function works the same as the **fgetc** function. It is equivalent to an **fgetc** (*stdin*).

Return Values

EOF	Indicates the end-of-file or an error. (EOF is a preprocessor constant defined in the <stdio.h> header file.)
x	Indicates the next character as an **int** from *stdin*.

getenv

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

Format

#include *<stdlib.h>*

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

Arguments

name
Can be one of the following values:

- "HOME"—The default directory (RSTS/E and RSX).
- "TERM"—The type of terminal being used (RSTS/E and RSX).
- "PATH"—The default device and directory (RSTS/E and RSX).
- "USER"—The UIC of the user who initiated the process (RSTS/E and RSX).
- "OPSYS"—The name of the operating system (all operating systems).

If the argument to **getenv** does not match any of the environment strings, the return value is NULL. If "TERM" is used as the argument and standard I/O is not being used, the return value is a pointer to a NULL string.

Return Values

x	Indicates a translated symbol.
NULL	Indicates that the translation failed.

gets

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

Format

#include *<stdio.h>*

char *gets *(char *s);*

Arguments

s
Pointer to the array to which the characters are read.

Description

The **gets** function reads characters from the standard input stream into the array pointed to by *s* until end-of-file or a new character is encountered. The newline character is discarded and a NUL character is written immediately after the last character read into the array.

Return Values

NULL	Indicates that end-of-file was encountered and no characters were read, or that an error has occurred.
x	A pointer to *s*.

gmtime

The **gmtime** function converts a given calendar time into a broken-down time, expressed as Coordinated Universal Time (UTC).

Format

#include *<time.h>*

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 returns a pointer to a structure of type **tm** which contains the time expressed as UTC. The current time zone must be set by using the **__tzset** function; otherwise, **gmtime** returns a NULL pointer.

See also **__tzset**.

hypot

The **hypot** function returns the square root of the sum of the squared arguments.

Format

#include *<math.h>*

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

Arguments

x
Is a real value.

y
Is a real value.

Description

The **hypot** function returns the following:

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

This function is provided for compatibility with VAX C and is only available if compiled with the /NOSTANDARD switch.

Return Values

Returns the square root of the sum of the squared arguments of x and y.

isalnum

The **isalnum** function is used to determine if a character is an alphanumeric in the current locale.

WARNING

This function is affected by the current locale setting.

Format

```
#include <ctype.h>
int isalnum (int c);
```

Arguments

c
Is an expression of type **int**.

Description

The **isalnum** function returns a nonzero integer if its argument is an alphanumeric character; otherwise, it returns 0. Refer to Chapter 3 for more information.

Return Values

Returns a nonzero integer if its argument is an alphanumeric character; otherwise, returns a zero.

isalpha

The **isalpha** function is used to determine if a character is an alphabetic character in the current locale.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int isalpha *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Description

The **isalpha** function returns a nonzero integer if its argument is an alphabetic character; otherwise, it returns 0. In PDP–11 C, **isalpha** is true only for characters having **isupper** or **islower** true. Refer to Chapter 3 for more information.

Return Values

Returns a nonzero integer if its argument is an alphabetic character; otherwise, returns a zero.

isascii

The **isascii** macro is used to determine if a character is ASCII.

Format

#include *<ctype.h>*

int isascii *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Description

The **isascii** macro returns a nonzero integer if its argument is any ASCII character; otherwise, it returns 0. This macro is provided for compatibility with VAX C and is only available when compiled with the /NOSTANDARD switch. Refer to Chapter 3 for more information.

Return Values

Returns a nonzero integer if its argument is any ASCII character; otherwise, returns a zero.

__ischar

The **__ischar** function returns a nonzero integer if its argument is contained in the current character set. Refer to Chapter 3 for more information.

Format

#include *<ctype.h>*

int __ischar *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is contained in the current character set; otherwise, returns a zero.

iscntrl

The **iscntrl** function returns a nonzero integer if its argument is a delete character or any nonprinting character for each of the character sets supported by PDP–11 C; otherwise, it returns 0. Refer to Chapter 3 for more information.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int iscntrl *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is a delete character or any nonprinting character; otherwise, returns a zero.

isdigit

The **isdigit** function returns a nonzero integer if its argument is a decimal digit character (0–9); otherwise, it returns 0. Refer to Chapter 3 for more information.

Format

#include *<ctype.h>*

int isdigit *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is a decimal digit character; otherwise, returns a zero.

isgraph

The **isgraph** function returns a nonzero integer if its argument is any printing character except 040 (SP); otherwise, it returns 0. Refer to Chapter 3 for more information.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int isgraph (**int** *c*);

Arguments

c
Is an expression of type **int**.

Description

Graphic ASCII characters are those with octal codes greater than or equal to 041 (!) and less than or equal to 0176 (?). They make up the set of characters you can print, except the space.

Return Values

Returns a nonzero integer if its character is any printing character except space; otherwise, it returns a zero.

islower

The **islower** function returns a nonzero integer if its argument is a lowercase alphabetic character; otherwise, it returns 0. Refer to Chapter 3 for more information.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int islower (**int** *c*);

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is a lowercase alphabetic character; otherwise returns a zero.

isprint

The **isprint** function returns a nonzero integer if its argument is a printing character including space, 040 (SP); otherwise, it returns 0. Refer to Chapter 3 for more information.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int isprint *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is a printing character; otherwise, returns a zero.

ispunct

The **ispunct** function returns a nonzero integer if its argument is a punctuation character, that is, if it is a printing character that is nonalphanumeric and not the space character; otherwise, it returns 0. Refer to Chapter 3 for more information.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int ispunct *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is a punctuation character; otherwise, returns a zero.

isspace

The **isspace** function returns a nonzero integer if its argument is white space; that is, if it is a space, tab (horizontal or vertical), carriage-return, form-feed, or newline character; otherwise, it returns 0. Refer to Chapter 3 for a list of additional characters that are in the Digital Multinational and ISO Latin–1 sets.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int isspace *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is white space; otherwise, returns a zero.

isupper

The **isupper** function returns a nonzero integer if its argument is an uppercase alphabetic character; otherwise, it returns 0. Refer to Chapter 3 for more information.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int isupper *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is an uppercase alphabetic character; otherwise, returns a zero.

isxdigit

The **isxdigit** function returns a nonzero integer if its argument is a hexadecimal digit (0 to 9, A to F, or a to f). Refer to Chapter 3 for more information.

Format

#include *<ctype.h>*

int isxdigit *(***int** *c);*

Arguments

c
Is an expression of type **int**.

Return Values

Returns a nonzero integer if its argument is a hexadecimal digit; otherwise, returns a zero.

labs

The **labs** function returns the absolute value of a **long int**.

Format

#include *<stdlib.h>*

long int labs *(***long int** *x);*

Arguments

x
Is a long **int**.

Return Values

Returns the absolute value of an integer as a long **int**.

ldexp

The **ldexp** function returns its first argument multiplied by 2 raised to the power of its second argument; that is, $x(2^{exp})$.

Format

#include *<math.h>*

double ldexp *(***double** *x*, **int** *exp);*

Arguments

x
Is a base value of type **double** that is to be multiplied by 2^{exp}.

exp
Is the integer exponent value to which 2 is raised.

Description

If there is a range error,the function sets *errno* to ERANGE and returns the constant HUGE_VAL. (HUGE_VAL is defined in the <math.h> header file to be the largest possible value of the appropriate sign.)

Return Values

0	Indicates that underflow has occurred.
x	$x(2^{exp})$

ldiv

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

Format

#include *<stdlib.h>*

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 div_t is defined in the standard include module <stdlib.h> header file as follows:

```
typedef struct LDIV_T
          {
           long int quot
           long int rem;
          }
           ldiv_t;
```

Return Values

Returns the quotient and remainder.

localeconv

The **localeconv** function obtains the appropriate values for formatting numeric quantities as controlled by the current locale.

Format

#include *<locale.h>*

struct lconv *localeconv*(void);*

Description

The **localconv** function returns a pointer to an object of type struct **lconv** which contains the values for the currently set locale. The **lconv** structure has the following members:

char *decimal_point	Character used for formatting nonmonetary quantities.
char *thousands_sep	Separates groups of digits before the decimal point in formatted nonmonetary quantities.
char *grouping	A string indicating the size of each group of digits in formatted nonmonetary quantities.
char *int_curr_symbol	International currency symbol for the current locale.
char *currency_symbol	Local currency symbol for the current locale.
char *mon_decimal_point	Character used for formatting monetary quantities.
char *mon_thousands_sep	Separates groups of digits before the decimal point in formatted monetary amounts.
char *mon_grouping	A string indicating the size of each group of digits in formatted monetary amounts.
char *positive_sign	A string indicating a positive formatted monetary amount.

char *negative_sign	A string indicating a negative formatted monetary amount.
char int_frac_digits	The number of fractional digits displayed in an internationally formatted monetary amount.
char frac_digits	The number of fractional digits displayed in a formatted monetary amount.
char p_cs_precedes	Is set to 1 if currency_symbol comes before the value for a positive formatted monetary quantity or to 0 if it comes after it.
char p_sep_by_space	Is set to 1 if currency_symbol is separated from the value of a positive formatted monetary quantity by a space or to 0 if it is not.
char n_cs_precedes	Is set to 1 if currency_symbol comes before the value of a negative formatted monetary amount or to 0 if it comes after it.
char n_sep_by_space	Is set to 1 if currency_symbol is separated by a space from the value of a negative formatted monetary amount or to 0 if it is not.
char p_sign_posn	Indicates the position of positive_sign for a positive formatted monetary amount.
char n_sign_posn	Indicates the position of negative_sign for a negative formatted monetary amount.

Return Values

Returns the pointer to the lconv object, filled in for the currently set locale.

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, expressed as local time.

Format

#include *<time.h>*

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. This time can be generated by the **time** function, or you can supply a time.

Description

The type tm is defined in the <time.h> header file as follows:

```
typedef struct tm
      {
         int  tm_sec,      /* seconds after the minute  -- [ 0, 60 ] */
              tm_min,      /* minutes after the hour    -- [ 0, 59 ] */
              tm_hour,     /* hours since midnight      -- [ 0, 23 ] */
              tm_mday,     /* day of the month          -- [ 1, 31 ] */
              tm_mon,      /* months since January      -- [ 0, 11 ] */
              tm_year,     /* years since 1900          -- [ 0, .. ] */
              tm_wday,     /* days since Sunday         -- [ 0,  6 ] */
              tm_yday,     /* days since January 1      -- [ 0,365 ] */
              tm_isdst;    /* Daylight Saving Time Flag -- [-1,  1 ] */
                           /* -1 info. not available   */
                           /*  0 D.S.T. IS-NOT in effect        */
                           /*  1 D.S.T. IS     in effect   */
      } tm_t;

/*
```

Successive calls to the **localtime** function 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.h>*

double log *(***double** *x);*

double log10 *(***double** *x);*

Description

The **log** and **log10** functions return the logarithm of their arguments. During error conditions, *errno* is set to EDOM if x is negative; *errno* is set to ERANGE if x is zero.

Return Values

log	Natural (base-e) logarithm of x
log10	Base-10 logarithm of x.

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, not by a series of **return** statements. The **longjmp** function restores the context of the environment buffer.

Please note that using **longjmp** calls across non-C functions may cause unpredictable results.

Format

#include *<setjmp.h>*

void longjmp **(jmp_buf** *env*, **int** *val);*

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.h> header file. The contents of the general-purpose registers, including the program counter (PC), are stored in the buffer.

val
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 the **setjmp** function is called to save a context, it returns the value 0. If the **longjmp** function is then called naming the same environment as a previous call to **setjmp**, control returns 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.

WARNING

You may invoke the **longjmp** function from a signal handler that has been established for any signal supported by the PDP-11 C Run-Time Library, subject to the following nesting restrictions:

- The **longjmp** function will not work if invoked from nested signal handlers. When invoked from a signal handler that has been entered as a result of an exception generated in another signal handler, the result of the **longjmp** function 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.

See also **setjmp**.

Return Values

0	First call, first return.
Nonzero value	Indicates a later call to the **longjmp** function using the same values.

__lr50a

The **__lr50a** function converts an unsigned 32-bit radix-50 string to the corresponding 6-character ASCII character string.

Format

#include *<stdlib.h>*

short int __lr50a *(***unsigned long int** **__rad50,* **char** **__ascii_string);*

Arguments

__rad50
Is a pointer to an unsigned 32-bit radix-50 string to be converted to ASCII.

__ascii_string
Is a pointer to a string to hold the converted six-character ASCII string.

Description

When **__lr50a** converts the radix-50 string to the ASCII character string, the string will not be NUL terminated.

Return Values

n	The number of characters translated.

malloc

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

Format

#include *<stdlib.h>*

void *malloc **(size_t** *size);*

Arguments

size
Specifies 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. The number of bytes is rounded to the next highest number evenly divisible by 4.

See also **calloc**.

Return Values

NULL	Indicates that it is unable to allocate enough memory.
x	The address of the first byte, which is aligned on a word boundary.

mblen

The **mblen** function determines the number of bytes in the multibyte character pointed to by its character pointer argument.

Format

#include *<stdlib.h>*

int mblen **(const char** ***s*****, size_t** ***n*****);**

Arguments

s
Is a character pointer.

n
Specifies the maximum number of bytes in the multibyte character that will be examined.

Description

The **mblen** function determines the number of bytes that make up the multibyte character pointed to by **s* if *s* is not a NULL pointer.

See also **mbtowc**.

Return Values

x	The number of characters that make up the next multibyte character in the multibyte string pointed to by *s*. The argument *s* cannot be a NULL pointer.

–1	Indicates the next character is not a valid multibyte character.
Nonzero	Indicates *s* is a NULL pointer, and the multibyte characters have state-dependent encoding; otherwise, 0 is returned.

mbstowcs

The **mbstowcs** function copies a sequence of characters from the string pointed to by *s* and stores them in the array pointed to by *pwcs*.

Format

#include *<stdlib.h>*

size_t mbstowcs *(***wchar_t** **pwcs,* **const char** **s,* **size_t** *n);*

Arguments

pwcs
Points to an array where the multibyte characters pointed to by *s* will be stored.

s
Points to an array of characters which are to be copied.

n
Specifies the maximum number of bytes in the multibyte character pointed to by *s*.

Description

The **mbstowcs** function returns the number of copied array elements. This does not include a terminating 0 code.

The sequence of characters pointed to by the character pointer argument is stored in the array pointed to by **pwcs**.

The **size_t** type is an **unsigned int** type defined in the <stddef.h> header file. The **wchar_t** type is an integral type representing distinct codes for all members of the largest extended character set specified by the supported locales.

See also **wcstombs**.

Return Values

Returns the number of copied array elements.

mbtowc

The **mbtowc** function copies the character pointed to by its character pointer argument into *pwc*.

Format

#include *<stdlib.h>*

int mbtowc *(***wchar_t** **pwc,* **const char** **s,* **size_t** *n);*

Arguments

pwc
Is a pointer to an object.

s
Is a character pointer.

n
Specifies the maximum number of bytes expected in the multibyte character pointed to by *s*.

Description

The **mbtowc** function determines the number of characters in the multibyte string *s* that make up the next multibyte character. The argument *s* cannot be a NULL pointer. The next multibyte character is converted to a wide character value; the value is placed in **pwc* if *pwc* is not a NULL pointer.

The **size_t** type is an **unsigned int** type defined in the <stddef.h> header file. The **wchar_t** type is an integral type representing distinct codes for all members of the largest extended character set specified by the supported locales. It is defined in the <stddef.h> header file.

See also **mblen**.

Return Values

x	The number of characters pointed to by **s* that make up the next multibyte character.
–1	Indicates the next or remaining characters are invalid multibyte characters.
Nonzero	Indicates *s* is a NULL pointer, and the multibyte characters have state-dependent encoding; otherwise, 0 is returned.

memchr

The **memchr** function locates the first occurrence of the specified byte within the initial *size* bytes of a given object pointed to by *s1*.

Format

#include *<string.h>*

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 the **strchr** function, the **memchr** function does not stop when it encounters a NUL character.

Return Values

Pointer	Is a pointer to the first occurrence of the character.
NULL	The character does not occur in the identified object string,

memcmp

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

Format

#include *<string.h>*

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

Arguments

s1
Is a pointer to the first object.

s2
Is a pointer to the second object.

n
Is the maximum number of characters to compare.

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 NUL character is encountered.

See also **strcmp**.

Return Values

<0	Indicates the object pointed to by *s1* is less than the object pointed to by *s2*.
0	Indicates the object pointed to by *s1* is equal to the object pointed to by *s2*.
>0	Indicates the object pointed to by *s1* is greater than the object pointed to by *s2*.

memcpy

The **memcpy** function copies a specified number of bytes from one object to another.

Format

#include *<string.h>*

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

Arguments

s1
Is a pointer to the first object.

s2
Is a pointer to the second object.

n
Is the number of characters pointed to by *s2*.

Description

The **memcpy** function copies *n* bytes from *s2* to *s1*. It does not check for the overflow of the receiving memory area (*s1*). Unlike the **strcpy** function, the **memcpy** function does not stop when a NUL character is encountered. The objects should not overlap.

See also **memmove** and **strcpy**.

Return Values

x	Indicates the value of *s1*.

memmove

The **memmove** function copies a specified number of bytes from one object to another, as if it first copied them into a temporary array of characters that does not overlap the objects pointed to by *s1* and *s2*, and then copied from the temporary array into the object pointed to by *s1*.

Format

#include *<string.h>*

void *memmove *(***void** **s1,* **const void** **s2,* **size_t** *n);*

Arguments

s1
Is a pointer to the first object.

s2
Is a pointer to the second object.

n
Is the number of characters to copy.

Description

The **memmove** function copies the specified number of bytes from one object to another.

The objects pointed to by *s1* and the object pointed to by *s2* may overlap.

Return Values

Returns the value of *s1*.

memset

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

Format

#include *<string.h>*

void *memset *(***void** **s*, **int** *c*, **size_t** *n);*

Arguments

s
Is a pointer to the object.

c
Is the value to be placed in each byte of *s*. It is converted to an unsigned char before it is copied.

n
Is the number of characters in *s* to be set to *c*.

Description

The **memset** function returns the value of *s*.

Return Values

Returns the value of *s*.

mktime

The **mktime** function converts the broken-down time in the structure pointed to by *timeptr* into a calendar time value.

Format

#include *<time.h>*

time_t mktime *(***struct tm** **timeptr);*

Arguments

timeptr
Pointer to a structure of type **tm**, which contains the broken-down time. The **tm** structure is defined in the <time.h> header file. See the **localtime** function for more information.

Return Values

–1	Indicates the calendar time cannot be represented.
Values other than –1	Returns the specified calendar time.

modf

The **modf** function returns the fractional part of the argument value with the same sign as the argument value and assigns the integral part, expressed as an object of type **double**, to the object whose address is specified by the second argument.

Format

#include *<math.h>*

double modf *(***double** *value,* **double** **iptr);*

Arguments

value
Must be an expression of type **double**.

iptr
Is a pointer to an expression of type **double** where the integral part of the result is stored.

Return Values

Returns the positive fractional part of the argument *value*.

perror

The **perror** function writes a short error message to *stderr* describing the last error encountered during a call to the PDP–11 C Run-Time Library from a C program.

Format

#include *<stdio.h>*

void perror *(***const char** **str);*

Arguments

str
Typically contains the name of the program that incurred 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 new line. The format of the message is:

```
string: error message
```

If a NULL is passed as the value, only the text of the error message is printed; the string is not printed.

Return Values

None.

pow

The **pow** function returns the first argument raised to the power of the second argument.

Format

#include *<math.h>*

double pow *(***double** *base,* **double** *exp);*

Arguments

base
Is an expression 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

Under the following conditions, *errno* is set to EDOM and zero is returned:

- If both arguments are 0.
- If *base* is 0 and *exp* is less than or equal to 0.
- If *base* is negative and *exp* is not an integer.

If a range error occurs, *errno* is set to ERANGE, and the result is set to HUGE_VAL or zero.

The constant HUGE_VAL is defined in the <math.h> header file to be the largest representable **double** value.

Return Values

x	The first argument raised to the power of the second argument.

printf

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

Format

#include *<stdio.h>*

int printf *(***const char** **format, . . .);*

Arguments

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

. . .
Represents optional expressions whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, the optional expression may be omitted; otherwise, the function call must have exactly as many optional expression as there are conversion specifications, and the conversion specifications must match the types of the optional expression. Conversion specifications are matched to output sources in left-to-right order. Refer to Chapter 2 for detailed information on conversion specifications.

Description

The following is an example of a conversion specification:

```
#include <stdio.h>
int main()
{
   int  temp = 4, temp2 = 17;

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

Sample output from the previous example is as follows:

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

Return Values

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

putc

The **putc** function writes a single character to a specified file.

Format

#include *<stdio.h>*

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

Arguments

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

file_ptr
Is a file pointer to the file in which the character is written.

Description

The **putc** function writes a single character to a file and returns the character. The file pointer is positioned after the character. In PDP–11 C, the **fputc** function and **putc** function are functionally equivalent. See also **fputc**.

Return Values

EOF	Indicates that an output error has occurred. EOF is defined in the <stdio.h> header file.
Character	Indicates success.

putchar

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

Format

#include *<stdio.h>*

int putchar *(***int** *character);*

Arguments

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

Description

The **putchar** function is identical to the **fputc** function (c, *stdout*). See also **fputc**.

Return Values

EOF	Indicates that an output error has occurred. EOF is defined in the <stdio.h> header file.
Character	Indicates success.

puts

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

Format

#include *<stdio.h>*

int puts *(***const 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 NUL character to the output stream.

Return Values

EOF	Indicates an error has occurred.
Number of characters written	Indicates success.

qsort

The **qsort** function sorts an array of objects in place.

Format

#include *<stdlib.h>*

void qsort *(***void** **base,* **size_t** *nmemb,* **size_t** *size,* **int** *(*compar)*
*(***const void** **x,***const void** **y));*

Arguments

base
Is a pointer to the initial member of the array. The pointer should be of type **pointer-to-element** and cast to type **pointer-to-void.**

nmemb
Is the number of objects in the array.

size
Is the size of an object in bytes.

compar
Is a pointer to the compare function.

x
Is an argument to the compare function.

y
Is an argument to the compare 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 than, 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 output order of two objects that compare as equal is unpredictable.

The **qsort** function must allocate one temporary having the size of a single element. If the **qsort** function is unable to allocate this temporary, it will place the value ENOMEM in *errno* and leave the array unsorted.

Return Values

Returns no values.

raise

The **raise** function generates a specified software signal. Generating a signal causes the action established by the **signal** function to be taken.

Format

#include *<signal.h>*

int raise *(***int** *sig);*

Arguments

sig
Identifies the signal to be generated.

Return Values

0	Indicates success.
Nonzero	Indicates failure.

rand

The **rand** function returns pseudorandom numbers in the range 0 to **RAND_MAX** ($2^{15}-1$). The **RAND_MAX** macro is defined by the standard library header, stdlib.h>.

Format

#include *<stdlib.h>*

int rand *(***void***);*

Return Values

Returns a pseudorandom integer.

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.h>*

void *realloc (**void** **ptr*, **size_t** *size*);

Arguments

ptr
Points to an area allocated by **malloc**, **calloc**, or **realloc**, or is NULL.

size
Specifies the new size of the allocated area.

Description

If ptr is the NULL pointer constant, the behavior of the **realloc** function is equivalent to that of the **malloc** function.

The contents of the area are unchanged up to the lesser of the old and new sizes. If the size is zero, **realloc** behaves similarly to the function **free**.

After a call to **realloc**, the storage area pointed to by *ptr* may be undefined unless **realloc** returns NULL.

Return Values

x	Indicates the address of the area because 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.
NULL	Indicates an inability to reallocate the space (for example, if there is not enough room).

remove

The **remove** function deletes a file.

Format

#include *<stdio.h>*

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

Arguments

file_spec
Is a pointer to the string that contains a file specification.

Description

The **remove** function deletes the file pointed to by *file_spec*.

Return Values

0	Indicates success.
Nonzero value	Indicates failure.

rename

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

Format

#include *<stdio.h>*

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 rename will fail. 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.h>*

void rewind (**FILE** **file_ptr);*

Arguments

file_ptr
Is a file pointer.

Description

The **rewind** function is equivalent to **fseek** (file_ptr, 0L, SEEK_SET). You can use the **rewind** function with either record or stream files.

Return Values

Returns no values.

scanf

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

Format

#include *<stdio.h>*

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

Arguments

format_spec
Uses conversion characters to specify how input is to be converted and placed in memory using subsequent arguments as pointers to the objects receiving the input. For a list of conversion characters, refer to Chapter 2.

. . .
Represents optional arguments that are pointers to the objects receiving the converted input according to the 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 pointers in simple left-to-right order.

Description

An example of a conversion specification is as follows:

```
#include <stdio.h>
int main()
{
   int  temp, temp2;

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

NOTE

A common programming error is to omit the ampersand (&) of &temp in line 4 of the program. If the ampersand is omitted, the address is not passed.

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

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

Return Values

x	Indicates the number of successfully matched and assigned input items.
EOF	Indicates end-of-file is encountered. EOF is a preprocessor constant defined in the <stdio.h> header file.

setbuf

The **setbuf** function associates a buffer with an input or output file.

Format

#include *<stdio.h>*

void setbuf *(***FILE** **file_ptr,* **char** **buf);*

Arguments

file_ptr
Is a pointer to a file.

buf
Is a pointer to an array. The buffer must be large enough to hold an entire input or output record. This is equivalent to the **setvbuf** call setvbuf(file_ptr, buf, _IOFBF, BUFSIZ).

If *buf* is NULL, I/O operations to that file will be unbuffered. This is equivalent to the **setvbuf** call setvbuf(file_ptr, NULL, _IONBF, 0). **_IONBF** is defined in the <stdio.h> header file.

Description

You can use the **setbuf** function after a file is opened, but you must use it before any input or output operations are performed.

A common error is to allocate buffer space as an "automatic" variable in a code block and then fail to close the file in the same block.

A buffer is normally obtained by calling **malloc**. For more information, see the **malloc** function.

See also **setvbuf**.

Return Values

Returns no values.

setjmp

The **setjmp** macro is used in transferring 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** macro saves the context of the calling function in an environment buffer.

Please note that using **longjmp** calls across non-C functions may cause unpredictable results.

Format

#include *<setjmp.h>*

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.h> header file. The contents of the general-purpose registers, including the program counter (PC), are stored in the buffer.

Description

When the **setjmp** macro is called to save a context, it returns the value 0. If the **longjmp** function is then called naming the same environment as the call to the **setjmp** macro, control returns 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 and is nonzero.

Return Values

0	First call, first return.
Value supplied by user in the **longjmp** call.	Second call, second return.

setlocale

The **setlocale** function sets the indicated character set, collating sequence, monetary format, decimal-point character, and time and date format in the Run-Time environment.

Format

#include *<locale.h>*

char *setlocale *(***int** *category,* **const char** **locale);*

Arguments

category
The following macros, which are defined in <locale.h>, may be specified by the category argument:

- LC_ALL specifies the program's entire locale.
- LC_COLLATE affects the behavior of the **strcoll** and **strxfrm** functions.
- LC_CTYPE affects the behavior of the character and multibyte handling functions.
- LC_MONETARY selects the monetary formatting as returned by the **localeconv** function.
- LC_NUMERIC selects the decimal-point character for formatted I/O, string conversion functions, and nonmonetary formatting information.
- LC_TIME sets the format of the time given by the **strftime** function.

locale
A value of "C" for *locale* sets the minimal C translation environment. To specify the implementation-defined native environment, which is identical to the "C" local, *locale* is given the value "" or one or more of the supported character sets.

setlocale

Description

The **setlocale** function returns a pointer to the string associated with the category argument for the new locale if the call is successful; otherwise, a NULL pointer is returned and the program's locale is not changed.

A subsequent call with the string value and its associated category restores part of the program's locale. The string returned by **setlocale** should not be modified; it may be overwritten by subsequent calls to the **setlocale** function. For more information, refer to Chapter 4.

Return Values

Pointer to a string	Indicates success.
NULL pointer	Indicates an unsuccessful call.

setvbuf

The **setvbuf** function associates a buffer with an input or output file.

Format

#include *<stdio.h>*

int setvbuf (**FILE** **file_ptr*, **char** **buf*, **int** *mode*, **size_t** *size*);

Arguments

file_ptr
Is a pointer to a file.

buf
Is a pointer to an array. If either _IOFBF or _IOLBF is specified as a value for mode, I/O operations use the array pointed to by *buf*. The buffer must be large enough to hold an entire input or output record.

If *buf* is NULL, I/O operations use a buffer automatically allocated by the PDP-11 C Run-Time Library. If _IONBF is specified for mode, I/O operations are completely unbuffered and the pointer in *buf* is ignored.

mode
Is a value that determines how the file will be buffered.

The following values for mode are defined in <stdio.h> header file:

- _IOFBF causes I/O to be fully buffered, if possible. Can be used for I/O requests made to files.
- _IOLBF causes output to be line buffered, if possible. The buffer is flushed when a newline character is written, when the buffer is full, or when input is requested. Can be used for I/O requests made to files.
- _IONBF causes I/O to be completely unbuffered, if possible, and *buf* and *size* to be ignored. Can only be used for I/O requests to and from your terminal.

size
Is the number of bytes in the array pointed to by *buf*. The constant BUFSIZ in <stdio.h> is recommended as an adequate buffer size.

For binary files: when using _IOFBF for the buffering mode, the *size* argument must be in multiples of 512 bytes, and the *size* must be at least 512 bytes.

Description

You can use the **setvbuf** function after a file is opened but you must use it before any input or output operations are performed.

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

A buffer is normally obtained by calling **malloc**. For more information, see the **malloc** function.

See also **setbuf**.

Return Values

Nonzero value	Indicates an invalid value is given for type or size.
0	Indicates success.

signal

The **signal** function allows the user to specify how a signal is to be handled.

Format

#include *<signal.h>*

void *(****signal** *(***int** *sig,* **void** *(*func)* **(int)***))***(int)***;*

Arguments

sig
Is the number or macro associated with a signal. The sig argument is usually one of the macros defined in the <signal.h> header file.

func
Is either the action to be taken when the signal is raised or the address of a function needed to handle the signal.

If func is the constant SIG_DFL, the action for the given signal is reset to the default action, 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 have been caught.) SIG_DFL and SIG_IGN are defined in the <signal.h> header file.

Description

You must call the **signal** function each time you want to catch a signal.

Return Values

x	Indicates the address of the function previously (or initially) established to handle the signal.
SIG_ERR	Indicates that the sig argument is out of range. The variable *errno* is set to EINVAL. SIG_ERR is defined in the <signal.h> header file, and EINVAL is defined in the <errno.h> header file.

sin

The sin function returns the sine of its radian argument.

Format

#include *<math.h>*

double sin *(***double** *x);*

Return Values

Returns the sine value of *x*.

sinh

The **sinh** function returns the hyperbolic sine of its argument.

Format

#include *<math.h>*

double sinh *(***double** *x);*

Arguments

x
x is the hyperbolic sine of the angle.

Description

The value of sinh(*x*), if it causes an overflow, is a **double** value with the largest possible magnitude and the appropriate sign. An overflow condition causes *errno* to be set to the value ERANGE.

Return Values

Returns the hyperbolic sine value.

__sleep, sleep

The __sleep or sleep function suspends execution for a specified time interval.

Format

#include *<signal.h>*

unsigned long int __sleep *(***unsigned long int** **itime);*

or

unsigned long int sleep *(***unsigned long int** **itime);*

Arguments

itime
Is an unsigned long integer which is in units of seconds.

Description

The **sleep** function causes the calling process to be suspended for *itime* seconds. The actual time can be up to one second less than *itime* due to granularity in system timekeeping. The entry point name **sleep** is VAX C compatible and is defined only when the compile-time switch of /NOSTANDARD is used.

The **__sleep** function is the same routine as the **sleep** function and can be used regardless of the value of the /NOSTANDARD switch.

Return Values

Value passed into the function.

sprintf

The **sprintf** function performs formatted output to a string in memory.

Format

#include *<stdio.h>*

int sprintf *(***char** **str,* **const char** **format_spec, . . .);*

Arguments

str
Is the address of the string that receives the formatted output.

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

. . .
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. For more information, refer to Chapter 2.

Description

An example of a conversion specification is as follows:

```
#include <stdio.h>
int main()
{
    int  temp = 4, temp2 = 17;
    char string[80];

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

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

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

Return Values

Returns the number of characters written to the array, not including the terminating NUL character.

sqrt

The **sqrt** function returns the square root of its argument.

Format

#include *<math.h>*

double sqrt **(double** *x);*

Description

The argument and the returned value are both objects of type **double**. The returned value will always be the positive square root. If *x* is negative, the function sets *errno* to EDOM and returns zero. EDOM is defined in the <errno.h> header file.

Return Values

Returns the value of the square root.

srand

The **srand** function sets the seed for a new sequence of pseudorandom numbers returned by subsequent calls to the **rand** function.

Format

#include *<stdlib.h>*

void srand (unsigned int *seed);*

Arguments

seed
Starting point for new number from which a particular sequence of pseudorandom numbers is generated.

Description

The random number generator is reinitialized by calling the **srand** function with the value 1, or it can be set to a specific point by calling **srand** with any other number.

See also **rand**.

Return Values

None.

__sr50a

The **__sr50a** function converts an unsigned 16-bit radix-50 string to the corresponding 3-character ASCII character string.

Format

#include *<stdlib.h>*

void __sr50a *(***unsigned short int** *__rad50,* **char** **__ascii_string);*

Arguments

__rad50
Is an unsigned 16-bit radix-50 string to be converted to ASCII.

__ascii_string
Is a pointer to a string to hold the converted three-character ASCII string.

Description

When __sr50a converts the radix-50 string to the ASCII character string, the string will not be NUL terminated. Three characters will always be returned. This function is undefined for inputs above 63999, as such inputs are invalid radix-50 strings.

Return Values

None.

sscanf

The **sscanf** function performs formatted input from a character string in memory.

Format

#include *<stdio.h>*

int sscanf *(***const 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_ptrs. Conversion specifications are matched to input sources in left-to-right order. For more information, refer to Chapter 2.

Description

An example of a conversion specification is as follows:

```
#include <stdio.h.>

int main ()
{
   int   temp, temp2;
   char  *astring = "4 17";

   sscanf(astring, "%d %d", &temp, &temp2);

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

4 17
```

Sample output from the previous example is as follows:

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

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.h> header file.

strcat

The **strcat** function concatenates one string to the end of the other.

Format

#include *<string.h>*

char *strcat *(char *s1,* **const char** **s2);*

Arguments

s1
Is a pointer to a string to which characters are appended.

s2
Is a pointer to a string from which the characters are appended to the string pointed to by *s1*.

s1, s2
Must be NUL-terminated character strings.

Description

The address of the first argument, *s1*, is assumed to point to a space large enough to hold the concatenated result. See also **strncat**.

Return Values

x	Indicates the address of the first argument, *s1*.

strchr

The **strchr** function returns the address of the first occurrence of *c*, converted to char, in a NUL-terminated string.

Format

#include *<string.h>*

char *strchr *(***const char** **s*, **int** *c);*

Arguments

s
Is a pointer to a NUL-terminated character string.

c
Is an expression of type **int** converted to a character.

Description

See also **strrchr**.

Return Values

x	Indicates the address of the first occurrence of the specified character.
NULL pointer	Indicates that the character does not occur in the string.

strcmp

The **strcmp** function compares two character strings and returns a negative integer, 0, or a positive integer, indicating that the value of the first string is less than, equal to, or greater than the value of the second string.

Format

#include *<string.h>*

int strcmp *(***const char** **s1,* **const char** **s2);*

Arguments

s1, s2
Are pointers to character strings.

Description

The comparison continues up to and including a NUL character in one of the strings; comparisons are terminated after the NUL is encountered.

See also **strncmp**.

Return Values

> 0	Indicates *s1* > *s2*.
= 0	Indicates *s1* = *s2*.
< 0	Indicates *s1* < *s2*.

strcoll

The **strcoll** function compares the string pointed to by *s1* to the string pointed to by *s2*.

Format

#include *<string.h>*

int strcoll *(***const char** **s1*, **const char** **s2);*

Arguments

s1, s2
Are pointers to character strings.

Description

The interpretation of the two strings by the **strcoll** function is dependent on the current locale.

See also **setlocale**.

Return Values

> 0	Indicates *s1* > *s2*.
= 0	Indicates *s1* = *s2*.
< 0	Indicates *s1* < *s2*.

strcpy

The **strcpy** function copies the NUL-terminated string pointed to by *s2* into a string beginning at *s1*.

Format

#include *<string.h>*

char *strcpy *(***char** **s1,* **const char** **s2);*

Arguments

s1, s2
Are pointers to character strings.

Description

The **strcpy** function copies the string pointed to by *s2* into the array pointed to by *s1*, stopping after copying a NUL character from *s2*. The strings pointed to by *s1* and *s2* may not overlap.

See also **strncmp**.

Return Values

x	Indicates the address of *s1*.

strcspn

The **strcspn** function computes the maximum initial segment of the string pointed to by *s1* containing none of the characters in the string pointed to by *s2*.

Format

#include *<string.h>*

size_t strcspn *(***const char** **s1*, **const char** **s2);*

Arguments

s1
Is a pointer to a character string. If the argument string is a NULL string, 0 is returned.

s2
Is a pointer to a character string containing the characters for which the function searches.

Description

The **strcspn** function scans the characters in string *s1*, stops when it encounters a character found in *s2*, and returns the length of the string's segment formed up to but not including the character found in *s2*.

See also **strspn** and **strpbrk**.

Return Values

x	Indicates the length of the initial segment of the string.

strerror

The **strerror** function maps the error number in its argument to an error message string.

Format

#include *<string.h>*

char *strerror *(***int** *errnum);*

Arguments

errnum
Is the error number to be mapped to an error message string.

Description

The following are the messages that the **strerror** function returns:

errnum	String
0	Not an error
1	Not owner
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Arg list too long
8	Exec format erro
9	Bad file number

errnum	String
10	No children
11	No more processes
12	Not enough core
13	Permission denied
14	Bad address
15	Block device required
16	Mount device busy
17	File exists
18	Cross-device link
19	No such device
20	Not a directory
21	Is a directory
22	Invalid argument
23	File table overflow
24	Too many open files
25	Not a typewriter
26	Text file busy
27	File too large
28	No space left on device
29	Illegal seek
30	Read-only file system
31	Too many links
32	Broken pipe
33	Math argument
34	Math result too large
35	I/O operation would block channel
all others	Invalid error value

Return Values

x	Indicates a pointer to a buffer that contains 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.

strftime

The **strftime** function gives the time to the **LC_TIME** category of the current locale. The appropriate characters are determined by the **LC_TIME** category of the current locale and by the values pointed to by **timeptr**.

Format

#include *<time.h>*

size_t **strftime** *(***char** **s,* **size_t** *maxsize,* **const char** **format,* **const struct tm** **timeptr);*

Arguments

s
Pointer to an array of characters where the result string is put.

maxsize
Maximum number of characters placed into the location pointed to by *s*.

format
String of 0 or more conversion characters (see table below).

timeptr
Structure containing broken down time.

Description

The following list describes the characters used in the format string to determine the behavior of the conversion specifier:

%a	—	Locale's abbreviated weekday name.
%A	—	Locale's full weekday name.

%b	—	Locale's abbreviated month name.
%B	—	Locale's full month name.
%c	—	Locale's appropriate date and time.
%d	—	Day of month as a decimal number (01–31).
%H	—	Hour (24-hour clock) as a decimal number (00–23).
%I	—	Hour (12-hour clock) as a decimal number (01–12.)
%j	—	Day of year as a decimal number (001–366).
%m	—	Month as a decimal number (01–12).
%M	—	Minute as a decimal number (00–59).
%P	—	Locale's equivalent of AM/PM format of a 12-hour clock.
%S	—	Second as a decimal number (00–61).
%U	—	Week number of the year (first Sunday as first day of week 1) as a decimal number (00–53).
%w	—	Weekday as a decimal number (00–06) (Sunday is 00).
%W	—	Week number of the year (first Monday as first day of week 1) as a decimal number (00–53).
%x	—	Locale's appropriate date representation.
%X	—	Locale's appropriate time representation.
%y	—	Year without century as a decimal number (00–99).
%Y	—	Year with century as a decimal number.
%Z	—	Time zone name or abbreviation. No characters if the time zone is indeterminable.
%%	—	Replaced by "%".

If the conversion specifier is not listed in the table, the behavior is undefined.

Return Values

x	The number of characters in the array pointed to by *s*.
0	Indicates the contents of the array are indeterminate.

strlen

The **strlen** function returns the length of a string of characters. The returned length does not include the terminating NUL character (\0). The type **size_t** is defined in the <stddef.h> and <string.h> header files.

Format

#include *<string.h>*

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 one string to the end of another.

Format

#include *<string.h>*

char *strncat *(***char** **s1,* **const char** **s2,* **size_t** *maxchar);*

Arguments

s1, s2
Must be NUL-terminated character strings that may not overlap.

maxchar
Specifies the number of characters to concatenate from *s2*, unless the **strncat** first encounters a NUL terminator in *s2*. If *maxchar* is 0 or negative, no characters are copied from *s2*.

Description

If **strncat** reaches the specified maximum, it sets the next byte in *s1* to the character (0), NUL. The address of the first argument, *s1*, is assumed to point to an array large enough to hold the concatenated result.

See also **strcat**.

Return Values

x	Indicates the address of the first argument, *s1*.

strncmp

The **strncmp** function compares not more than *n* elements of the two character strings and returns a negative integer, 0, or a positive integer, indicating that the value of the first string is less than, equal to, or greater than the value of the second string.

Format

#include *<string.h>*

int strncmp *(***const char** **s1*, **const char** **s2*, **size_t** *n);*

Arguments

s1, s2
Are pointers to character strings.

n
Specifies a maximum number of characters (beginning with the first) to compare in both *s1* and *s2*. If *n* is 0, no comparison is performed and 0 is returned (the strings are considered equal).

Description

The comparison is terminated when a NUL is encountered in one of the strings or when the first n characters of the strings have been compared.

See also **strcmp**.

Return Values

<0	Indicates the prefix length of n in the string pointed to by *s1* is less than the prefix length of n in the string pointed to by *s2*.
=0	Indicates *s1* = *s2*.
>0	Indicates *s1* > *s2*.

strncpy

The **strncpy** function copies all or part of a string.

Format

#include *<string.h>*

char *strncpy *(***char** **s1*, **const char** **s2*, **size_t** *n);*

Arguments

s1, s2
Are pointers to character strings.

n
Specifies the maximum number of characters to copy from *s2* to *s1*.

Description

The function **strncpy** copies no more than *n* characters from *s2* to *s1*, up to and including the NUL terminator of *s2*. If *s2* contains less than *n* characters, *s1* is padded with NUL characters. If *s2* contains greater than or equal to *n* characters, the first *n* characters of *s2* are copied to *s1*.

NOTE

The argument *s1* is not necessarily terminated by a NUL character.

See also **strcpy**.

Return Values

x	Indicates the address of *s1*.

strpbrk

The **strpbrk** function searches a string for the occurrence of one of a specified set of characters.

Format

#include *<string.h>*

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, NULL is returned.

charset
Is a pointer to a character string containing the characters for which the function searches.

Description

The **strpbrk** function scans the characters in the string, stops when it encounters a character found in *charset*, and returns a pointer to the first character in *str* found in *charset*.

Return Values

x	Indicates the address of the first character in the string that is in the set.
NULL pointer	Indicates that no character is in the set.

strrchr

The **strrchr** function returns the address of the last occurrence of *c*, converted to char, in a NUL-terminated string.

Format

#include *<string.h>*

char *strrchr *(***const char** **s,* **int** *c);*

Arguments

s
Is a pointer to a NUL-terminated character string.

c
Is the character for which **strrchr** searches.

Description

See also **strchr**.

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 sequentially searches a string for the first occurrence of a character that is not in a specified set of characters.

Format

#include *<string.h>*

size_t strspn *(***const char** **s1,* **const char** **s2);*

Arguments

s1
Is a pointer to a character string. If the argument string is a NULL string, 0 is returned.

s2
Is a pointer to a character string containing the set of characters for which the function searches.

Description

The **strspn** function scans the characters in the string *s1* stopping when it encounters a character not found in *s2*. It then returns the length of *s1*'s initial segment formed by characters found in *s2*.

If the characters in the character strings pointed to by *s1* and *s2* match, **strspn** returns the length of *s1*; otherwise, it returns 0.

See also **strcspn** and **strpbrk**.

Return Values

x	Indicates the length of the matching prefix of the segment.
0	Indicates no characters match.

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.h>*

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

Arguments

s1
Is the address of the character string the **strstr** function searches.

s2
Is the address of the character string for which the **strstr** function searches.

Return Values

NULL	Indicates that the string was not found.
x	A pointer to the located string within *s1*.

strtod

The **strtod** function converts a given string to an object of type **double**.

Format

#include *<stdlib.h>*

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

Arguments

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

endptr
Is the address of an object that stores 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.

Description

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

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

The **strtod** function returns the converted value. 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 *errno* is set to ERANGE. HUGE_VAL is defined in the <math.h> header file, and ERANGE is defined in the <errno.h> header file.

- If the correct value causes an underflow, 0 is returned and *errno* is set to ERANGE.

If the string starts with an unrecognized character, no conversion is performed, **endptr is set to nptr (unless nptr is NULL), and 0 is returned.

See also **atof**.

Return Values

x	Is the converted value, if any.
0	Indicates that no conversion was made.

strtok

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

Format

#include *<string.h>*

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 initial character in the first token and writes a NUL character into *s1* immediately following the returned token. Each subsequent call (with the value of the first argument 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 NUL characters inserted into *s1* by the **strtok** function; therefore, *s1* cannot be a **const** object. The **strtok** function is nonreentrant because it 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 pointer	Indicates that no token was found.

strtol

The **strtol** function converts a string to an object of type **long**.

Format

#include *<stdlib.h>*

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 that stores 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 0s 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.h>) 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 notwithstanding). The function call **atol (str)** is equivalent to **strtol (str, (char**)0, 10)**.

See also **atoi** and **atol**.

Return Values

x	Indicates the converted value.
LONG_MAX or LONG_MIN	Indicates the correct value will cause an overflow (according to the sign of the value). *errno* is set to ERANGE. LONG_MAX and LONG_MIN are defined in the <limits.h> header file.
0	Indicates that the string starts with an unrecognized character. The argument **endptr is set to nptr.

strtoul

The **strtoul** function converts a string to an **unsigned long** integer.

Format

#include *<stdlib.h>*

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

Arguments

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

endptr
Is the address of an object that stores 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 0s 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, and the value of ERANGE is stored in *errno*. ULONG_MAX is defined in the <limits.h> header file.

strxfrm

The **strxfrm** function transforms the string pointed to by *s2* according to the collating sequence established by the **setlocale** function and places the transformed string into an array pointed to by *s1*.

Format

#include *<string.h>*

size_t strxfrm *(***char** **s1*, **const char** **s2*, **size_t** *n);*

Arguments

s1
Is the location for the placement of the transformed string.

s2
Is the location of the string to be transformed.

n
Is the maximum number of transformed characters to be placed in s1.

Return Values

Less than n	Returns the length of the transformed string.
n or more	Indicates the contents of the array pointed to by *s1* are indeterminate.

system

The **system** function passes a given string to the host environment to be executed by a command processor.

NOTE

Passing commands to the host environment and the command line processor is only available on the RSX Operating System.

Format

#include *<stdlib.h>*

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

Arguments

string
Is a pointer to the string to be executed.

Description

The **system** function spawns the default command language interpreter and executes the command specified by *string*. The **system** function waits for the command to complete before returning the exit status as the return value of the function.

On the RSX operating system, if the **system** function is called with a NULL pointer, a nonzero value is returned indicating that passing a command line to the command line interpreter is available.

On the RT–11 and RSTS/E operating systems, if the **system** function is called with a NULL pointer, a zero is returned indicating that passing a command line to the command line interpreter is not available.

Return Values

string is NULL:	
Nonzero value	Indicates passing a command line to a command line interpreter is available (RSX operating system only).
0	Indicates passing a command line to a command line interpreter is available (RT–11 and RSTS/E operating systems).
string is not NULL:	
Nonzero	Value passed by operating system (RSX operating system only).
0	Value not passed by operating system (RT–11 and RSTS/E operating systems.)

tan

The **tan** function returns a **double** value that is the tangent of its radian argument.

Format

#include *<math.h>*

double tan *(***double** *x);*

Arguments

x

x is the tangent of the angle.

Description

The value of tan(x) at its "singular points" (. . . $-3\pi/2, -\pi/2, \pi/2$ . . .) is the largest possible **double** value, and *errno* is set to ERANGE. ERANGE is defined in <errno.h> header file.

Return Values

Returns the tangent value of x.

tanh

The **tanh** function returns a **double** value that is the hyperbolic tangent of its **double** argument.

Format

#include *<math.h>*

double tanh *(***double** *x);*

Arguments

x

x is the hyperbolic tangent of the angle.

Return Values

Returns the hyperbolic tangent value of *x*.

time

The **time** function returns the time elapsed since 00:00:00, January 1, 1970, in seconds.

Format

#include *<time.h>*

time_t time (**time_t** **timer);*

Arguments

timer
Is either NULL or a pointer to the place where the returned time is also stored.

Return Values

x	Specifies current calendar time.
–1	Indicates an error has occurred.

tmpfile

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

Format

#include *<stdio.h>*

FILE *tmpfile *(***void***);*

Description

The file is created in mode "wb+".

When using the RSX operating system with FCS file I/O, the file is deleted if the task exits abnormally, or if the **abort** function is called. If the task exits abnormally and RMS is being used, the file may become a "lost" file.

When using the RSTS/E operating system, a RSTS/E temporary file is created and will be deleted at logout.

When using the RT–11 operating system, a file named CTEMPC.TMP is created. The file is deleted when it is closed. If the program terminates abnormally, the file may not be deleted.

Return Values

x	Indicates the address of a FILE object associated with the file (defined in the <stdio.h> header file).
NULL	Indicates that there is an error.

tmpnam

The **tmpnam** function creates a character string that you can use in place of the filename argument in other function calls.

Format

#include *<stdio.h>*

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

Description

PDP–11 C generates names in the following form:

```
CC<system dependent><1 letter>.TMP
```

The names are always generated beginning with capital "CC" and ending with ".TMP." The <1 letter> field contains the final letter before the file extension. This letter varies each time the **tmpnam** function is called starting with an "A" the first call, a "B" the second call, and so on to "Z". The cycle repeats itself after the letter "Z".

The <system dependent> field generates a unique set of characters depending on the operating system. Each operating system uses a different method of identifying processes as follows:

- RSX Operating System

  The field is six characters long and is the name of the task running (with dots removed).

- RSTS Operating System

  The field is two characters long and is the job number of the task running.

- RT–11 Operating System

  The field is two characters long and is the job number of the task running.

Arguments

name
Is a pointer to a character string to receive a name to use in place of filename arguments in other functions. If name is NULL, an internal storage area is used. Successive calls to **tmpnam** cause the function to overwrite the contents of the string.

Return Values

name	A pointer to the filename.

toascii

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

Format

#include *<ctype.h>*

int toascii *(***char** *character);*

Arguments

character
Is an expression of type **char**.

Description

This macro is provided for VAX C compatibility and is only available when compiled using the /NOSTANDARD switch.

Return Values

x	Specifies a 7-bit ASCII character.

tolower

The **tolower** function converts its argument, an uppercase character, to lowercase. If the argument is not an uppercase character, it is returned unchanged.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int tolower *(***int** *character);*

Arguments

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

Return Values

Returns a lowercase character.

_tolower

The **_tolower** macro converts its argument, an uppercase character, to lowercase. If the argument is not an uppercase character, it is returned unchanged.

WARNING

This macro is affected by the current locale setting.

Format

#include *<ctype.h>*

int _tolower *(***int** *character);*

Arguments

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

Description

This macro is provided for VAX C compatibility and is only available when compiled using the /NOSTANDARD switch.

Return Values

Returns a lowercase character.

toupper

The **toupper** function converts its argument, a lowercase character, to uppercase. If the argument is not a lowercase character, it is returned unchanged.

WARNING

This function is affected by the current locale setting.

Format

#include *<ctype.h>*

int toupper (**int** *character);*

Arguments

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

Return Values

Returns an uppercase character.

_toupper

The **_toupper** macro converts its argument, a lowercase character, to uppercase. If the argument is not a lowercase character, it is returned unchanged.

WARNING

This macro is affected by the current locale setting.

Format

#include *<ctype.h>*

int toupper *(***int** *character);*

Arguments

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

Description

This macro is provided for VAX C compatibility and is only available when compiled using the /NOSTANDARD switch.

Return Values

Returns an uppercase character.

__tzset

The **__tzset** function sets the system time zone and daylight time variables. If the time zone is not set, **gmtime** does not work. See also **gmtime** and **localtime**.

Format

#include *<time.h>*

void __tzset *(***int** *zone,* **int** *daylight);*

Arguments

zone
A positive integer represents the number of hours West of the UTC zone, and a negative integer represents the number of hours East of the UTC zone.

daylight
Represents daylight time. If daylight is false, the return value of tm_isdst of struct tm from the **localtime** function is set to 0; otherwise, it is set to 1.

The following two examples show how to set the time zone to Eastern Standard Time and to Eastern Daylight Time:

```
__tzset (5,0);        /* Current time zone set to Eastern Standard Time
                         which is five hours west of GMT. */

__tzset (5,1);        /* Current time zone set to Eastern Daylight Time. */
```

Return Values

Returns no values.

ungetc

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

Format

#include *<stdio.h>*

int ungetc *(***int** *c,* **FILE** **file_ptr);*

Arguments

c
Specifies the character to be pushed back onto the stream pointed to by stream.

file_ptr
Is a file pointer.

Description

When the **ungetc** function is used, the character is "pushed back" onto the file and 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**, **fsetpos**, and **rewind** functions erase all memory of pushed-back characters. The pushed-back character is not written to the underlying file. The EOF character may not be pushed back.

Return Values

x	Indicates the push-back character.
EOF	Indicates it cannot push the character back.

va_arg

The **va_arg** macro returns the values of successive arguments in turn.

Format

#include *<stdarg.h>*

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

Arguments

ap
Is an object of type va_list used to traverse the argument list. The user must always declare and use the argument *ap*, which is the same as the parameter initialized by the **va_start** macro. For further information, refer to the **va_start** macro.

type
Is a type name specified so that ap will be assigned a pointer to an object having the type *type*. If there is no next argument or the type of the next argument is not compatible with *type*, the behavior is undefined.

Description

The **va_arg** macro expands to a value having the type of the next called argument. Subsequent calls to **va_arg** modify ap so that the values of successive arguments are returned in succession.

Return Values

The next argument in a variable-length argument list.

va_end

The **va_end** macro sets its argument to NULL.

Format

#include *<stdarg.h>*

void va_end (**va_list** *ap);*

Arguments

ap
Is the object used to traverse the variable-length argument list. You must always declare and use the argument *ap*.

Description

If the **va_end** macro is not called before the return or there is no corresponding call to the **va_start** macro, the behavior is undefined.

Return Values

None.

va_start

The **va_start** macro is used to initialize a variable to the beginning of the variable argument list.

Format

#include *<stdarg.h>*

void va_start *(***va_list** *ap, parmN)* **;**

Arguments

ap
Is an object pointer. You must always declare and use the argument *ap*.

parmN
Is the identifier of the rightmost fixed argument in the variable argument list of the function definition.

Description

The pointer *ap* is initialized to point to the first optional argument that follows *parmN* in the argument list.

Return Values

None.

vfprintf

The **vfprintf** function prints formatted output based on an argument list.

Format

#include *<stdio.h>*

#include *<stdarg.h>*

int vfprintf (**FILE** **file_ptr,* **const char** **format,* **va_list** *arg);*

Arguments

file_ptr
Is a pointer to a file.

format
Contains characters to be written literally to the output or converted as specified.

arg
Is a list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The **vfprintf** function is the same as the **fprintf** function, except it is called with an argument list that has been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls) instead of being called with a variable number of arguments. It does not invoke the **va_end** macro. Refer to the **va_arg** macro for further information.

See also **vprintf** and **vsprintf**.

Return Values

x	Indicates the number of transmitted characters.
Negative value.	Indicates an output error.

vprintf

The **vprintf** function prints formatted output based on an argument list.

Format

#include *<stdio.h>*

#include *<stdarg.h>*

int vprintf *(***const char** **format,* **va_list** *arg);*

Arguments

format
Contains characters to be written literally to the output or converted as specified.

arg
Is a list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The **vprintf** function is the same as the **printf** function, except it is called with an argument list that has been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls) instead of being called with a variable number of arguments. For further information, refer to the **va_arg** and **va_start** macros.

See also **vfprintf** and **vsprintf**.

Return Values

x	Indicates the number of transmitted characters.
Negative value.	Indicates an output error.

vsprintf

The **vsprintf** function prints formatted output based on an argument list.

Format

#include *<stdio.h>*

#include *<stdarg.h>*

int vsprintf **(char** **str*, **const char** **format*, **va_list** *arg);*

Arguments

str
Is a pointer to a string.

format
Contains characters to be written literally to the output or converted as specified.

arg
Is a list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The **vsprintf** function is the same as the **sprintf** function, except it is called with an argument list that has been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls) instead of being called with a variable number of arguments. For further information, refer to the **va_arg** and **va_start** macros.

Return Values

x	Indicates the number of characters written to the array, excluding the terminating NUL character.
Negative value.	Indicates an output error.

wcstombs

The **wcstombs** function converts a sequence of codes corresponding to multibyte characters into a sequence of multibyte characters and stores them in the array pointed to by the character pointer argument.

Format

#include *<stdlib.h>*

size_t wcstombs *(***char** **s,* **const wchar_t** **pwcs,* **size_t** *n);*

Arguments

s
Is a character pointer argument.

pwcs
Points to the array of multibyte characters corresponding to a sequence of codes converted by the **wcstombs** function.

n
Specifies the number of stored characters.

Description

The **wcstombs** function returns the number of modified bytes. This does not include a terminating NUL character. If the code does not match a valid multibyte character, **wcstombs** returns (size_t)–1.

The multibyte characters produced by the conversion of codes pointed to by **pwcs** beginning in the initial shift state are stored in the array pointed to by the character pointer argument.

Return Values

(size_t)–1	Indicates the code does not match a valid multibyte character.
x	Indicates the number of modified bytes excluding the terminating NUL character.

wctomb

The **wctomb** function determines the number of bytes needed to represent the multibyte character whose code value equals **wchar**.

Format

#include *<stdlib.h>*

int wctomb *(***char** **s*, **wchar_t** *wchar);*

Arguments

s
Points to the array of multibyte character representation corresponding to the code whose value is *wchar*.

wchar
Is the value of the code needed to represent the multibyte character pointed to by *s*.

Description

The **wctomb** function returns a nonzero or 0 value if the character pointer argument is a NULL pointer.

If the character pointer argument is not a NULL pointer, the return value is either the number of bytes in the multibyte character corresponding to the value of **wchar**, or a –1 if it does not correspond to **wchar**.

Return Values

0 or nonzero value	Indicates the character pointer argument is a NULL pointer.
Value of **wchar**	Indicates the character pointer argument is not a NULL pointer.
–1	Indicates the character pointer argument is not a NULL pointer and the value does not correspond to **wchar**.

2 FCS Extension Library Macros

FCS$ASCPP

The **FCS$ASCPP** function converts a directory string from ASCII to its equivalent binary UIC.

Format

#include *<fcs.h>*

short FCS$ASCPP **(char** **dds*, **short** **uic)*

Arguments

dds
Specifies a pointer to the directory string descriptor.

uic
Specifies a pointer to the word location to which the binary UIC is to be returned.

Description

The **FCS$ASCPP** function converts the directory string contained in *dds* to its equivalent binary UIC.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$ASLUN

The **FCS$ASLUN** function assigns a logical unit number (LUN) to a specified device and unit and returns the device information to a specified FDB and filename block.

Format

#include *<fcs.h>*

short FCS$ASLUN **(fcs$fdb** ***fdb*, fcs$fnb** ***fnb*)**

Arguments

fdb
Specifies a pointer to the the desired FDB.

fnb
Specifies a pointer to the filename block.

Description

The **FCS$ASLUN** function returns to the specified filename block and the specified FDB, information identical to that returned by the device and unit logic of the **FCS$PARSE** function.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$CLOSE$

The **FCS$CLOSE$** function terminates file processing in an orderly manner.

Format

#include *<fcs.h>*

short FCS$CLOSE$ *(***fcs$fdb** **fdb,* **void** *(*err)())*

Arguments

fdb
Specifies a pointer to the associated FDB.

err
Specifies a pointer to the optional, user-coded, error-handling routine.

Description

The **FCS$CLOSE$** function terminates file processing in an orderly manner. If an error condition is detected during the **FCS$CLOSE$** operation, the user-specified, error-handling routine is called.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$CTRL

The **FCS$CTRL** function performs device-specific control functions.

Format

#include *<fcs.h>*

short FCS$CTRL **(fcs$fdb** **fdb*, **short** *function*, **short** *blocks*, **short** *0)*

Arguments

fdb
Specifies a pointer to the associated FDB.

function
Specifies the function code.

blocks
If the function is FCSFFSPC, this specifies the number of blocks to be spaced forward or backward; otherwise, it must be zero.

0
Last argument is always 0.

Description

The **FCS$CTRL** function performs device-specific control functions, such as:

- Rewind a magnetic tape volume set.
- Position to the logical end of a magnetic tape volume set.
- Space forward or backward n blocks on a magnetic tape.
- Rewind a file on a magnetic tape or terminal (record-oriented device).

- Clear the terminal end-of-file.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$DELET$

The **FCS$DELET$** function removes a named file from the associated volume directory and deallocates the space occupied by the file.

Format

#include *<fcs.h>*

short FCS$DELET$ *(***fcs$fdb** **fdb,* **void** *(*err)())*

Arguments

fdb
Specifies a pointer to the associated FDB.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The **FCS$DELET$** function causes the directory information for the file associated with the specified FDB to be deleted from the appropriate User File Directory (UFD). The space occupied by the file is then deallocated and returned for reallocation to the pool of available storage on the volume.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$DLFNB

The **FCS$DLFNB** function deletes a file by filename block.

Format

#include *<fcs.h>*

short FCS$DLFNB *(***fcs$fdb** **fdb)*

Arguments

fdb
Specifies a pointer to the associated FDB.

Description

The **FCS$DLFNB** function assumes that the filename block is completely filled; when called, it closes the file if necessary, and then deletes the file.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$ENTER

The **FCS$ENTER** function inserts an entry by file name into a directory.

Format

#include *<fcs.h>*

short FCS$ENTER **(fcs$fdb** **fdb*, **fcs$fnb** **fnb)*

Arguments

fdb
Specifies a pointer to the desired FDB.

fnb
Specifies a pointer to the filename block.

Description

The **FCS$ENTER** function inserts an entry by file name into a directory.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$EXPLG

The **FCS$EXPLG** function expands a logical name and returns a pointer to the task that points to the expanded string.

Format

#include *<fcs.h>*

short FCS$EXPLG *(int **dsd)*

Arguments

dsd
Specifies a pointer to the data set descriptor of the string to be expanded.

Description

The **FCS$EXPLG** function expands the string into the same buffer that the **FCS$PARSE** function uses for input files; therefore, caution is advised in using this function. In addition, the call accepts only logical names that expand into a correct FCS file specification.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$EXTND

The **FCS$EXTND** function extends either contiguous or noncontiguous files. The file to be extended can be either open or closed.

Format

#include *<fcs.h>*

short FCS$EXTND (**fcs$fdb** **fdb*, **short** *extnd_size*, **short** *ecb*)

Arguments

fdb
Specifies a pointer to the associated FDB.

extnd_size
Specifies a numeric value specifying the number of blocks to be added to the file.

ecb
Specifies the extension control bits, as appropriate.

Description

The **FCS$EXTND** function disables file truncation. Explicitly calls the **FCS$TRNCL** function to truncate a file after calling the **FCS$EXTND** function.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$FDBDF$

The **FCS$FDBDF$** macro allocates space in the program for an FDB.

Format

#include *<fcs.h>*

short FCS$FDBDF$ *(class, name)*

Arguments

class
Specifies the storage class used in allocating the storage for the FDB that is being declared.

name
Specifies the name of the FDB that is being declared.

Description

The **FCS$FDBDF$** macro must be specified in the program once for each input or output file that the program simultaneously opens during execution.

Return Values

None.

FCS$FIND

The **FCS$FIND** function locates a directory entry by file name and lists it in the file identification field in both the Master File Directory (MFD) and User File Directory (UFD).

Format

#include *<fcs.h>*

FCS$FIND *(***fcs$fdb** **fdb,* **fcs$fnb** **fnb)*

Arguments

fdb
Specifies a pointer to the desired FDB.

fnb
Specifies a pointer to the filename block.

Description

The **FCS$FIND** function searches the directory file specified in the filename block. The file is searched for an entry that matches the specified file name, file type, and file version number.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$FINIT$

The **FCS$FINIT$** function initializes coding to set up the FSR.

Format

#include *<fcs.h>*

FCS$FINIT$

Arguments

None.

Description

In the case of a program that is written so that it can be restarted, it is necessary to issue the FCS$FINIT$ function call in the program's initialization code because such a program performs all its initialization at run time, rather than at assembly time.

Return Values

None.

FCS$FLUSH

The **FCS$FLUSH** function writes the block buffer to the file being written in record mode.

Format

#include *<fcs.h>*

short FCS$FLUSH (**fcs$fdb** **fdb*)

Arguments

fdb
Specifies a pointer to the associated FDB.

Description

The **FCS$FLUSH** function writes file attributes each time it is called. It should be used whenever data needs to be immediately written to a file.

Closing the file also guarantees that the block buffer is flushed and that the file attributes are written back to the file header.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$FSRSZ$

The **FCS$FSRSZ$** function establishes the size of the FSR.

Format

#include *<fcs.h>*

FCS$FSRSZ$ *(**int** fbufs, **int** bufsiz)*

Arguments

fbufs
Specifies the number of files to be opened.

bufsiz
Specifies the total block buffer pool space (in bytes) needed to support the maximum number of files that can be opened simultaneously.

Description

The **FCS$FSRSZ$** function does not generate executable code; it merely allocates space for a block-buffer pool.

Return Values

None.

FCSGET

The **FCSGET** function reads logical data records from a file.

Format

#include *<fcs.h>*

short FCSGET (**fcs$fdb** **fdb*, **char** **urba*, **short** *urbs,)*
void *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The **FCSGET** function reads logical records from a file.

Return Values

1	Indicates success.
0	Indicates failure.

FCSGETR

The **FCSGETR** function reads fixed-length records from a file in random mode.

Format

#include *<fcs.h>*

short FCSGETR **(fcs$fdb** **fdb*, **char** **urba*, **short** *urbs,)*
short *lrcnm*, **short** *hrcnm*, **void** *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

lrcnm
Specifies the low-order 16 bits of the number of the record to be read.

hrcnm
Specifies the high-order 15 bits of the number of the record to be read.

err
Specifies the address of the optional, user-coded, error-handling routine.

FCSGETR

Description

By definition, issuing the **FCSGETR** function requires familiarity with the structure of the file to be read and that the number of the record to be read is precisely specified.

Return Values

1	Indicates success.
0	Indicates failure.

FCSGETS

The **FCSGETS** function reads records from a file in sequential mode.

Format

#include *<fcs.h>*

short FCSGETS (**fcs$fdb** **fdb*, **char** **urba*, **short** *urbs,)*
void *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The **FCSGETS** function is specifically for use in an overlaid environment in which the amount of memory available to the program is limited and files are to be read in strictly sequential mode.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$GTDID

The **FCS$GTDID** function inserts directory information into a specified filename block.

Format

#include *<fcs.h>*

short FCS$GTDID *(***fcs$fdb** **fdb,* **fcs$fnb** **fnb)*

Arguments

fdb
Specifies a pointer to the associated FDB.

fnb
Specifies a pointer to the filename block into which the directory information is to be placed.

Description

The **FCS$GTDID** function uses the binary value found in the default UIC word as the desired UFD, unlike the **FCS$GTDIR** function, which allows the specification of the directory string.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$GTDIR

The **FCS$GTDIR** function inserts directory information from a directory string descriptor into a specified filename block.

Format

#include *<fcs.h>*

short FCS$GTDIR **(fcs$fdb** **fdb,* **fcs$fnb** **fnb,* **int** **dsd)*

Arguments

fdb
Specifies a pointer to the associated FDB.

fnb
Specifies a pointer to the filename block into which the directory information is to be placed.

dsd
Specifies a pointer to the 2-word directory string descriptor.

Description

The FCS$GTDIR function returns the directory ID to the 3 words of the specified filename block, preserving information in offset locations N.FNAM, N.FYTP, N.FVER, N.DVNM, and N.UNIT of the filename block, but clearing the rest of the filename block.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$MARK

The **FCS$MARK** function points to a byte or record within a specified file.

Format

#include *<fcs.h>*

short FCS$MARK (**fcs$fdb** **fdb*, **short** **highbits*, **short** **lowbits*, **short** **bytenum*)

Arguments

fdb
Specifies a pointer to the associated FDB.

highbits
Specifies a pointer to the location to store the high-order bits of the virtual block number.

lowbits
Specifies a pointer to the location to store the low-order bits of the virtual block number.

bytenum
Specifies a pointer to the location to store the number of the next byte within the virtual block.

Description

The **FCS$MARK** function saves current position information of a file for later use. By saving the current position information of a file, the file can be closed and later reopened to the same position. The **FCS$MARK** function also allows records to be altered within a file.

Return Values

None.

FCS$MRKDL

The **FCS$MRKDL** function marks a temporary file for deletion.

Format

#include *<fcs.h>*

short FCS$MRKDL **(fcs$fdb** **fdb)*

Arguments

fdb
Specifies a pointer to the associated FDB.

Description

The **FCS$MRKDL** function is called prior to closing a temporary file; the file is deleted when it is closed.

NOTE

If the file contains sensitive information, it should be cleared before closing, or the disk should be reformatted to destroy the information.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$OFID$x

The **FCS$OFID$x** functions open an existing file by using file identification information in the filename block.

Format

#include *<fcs.h>*

short FCS$OFID$x (**fcs$fdb** **fdb*, **short** *lun*, **short** **dspt,*)
short *racc*, **char** **urba*, **short** *urbs*,
void *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

lun
Specifies the LUN associated with the desired file.

dspt
Specifies a pointer to the data-set descriptor.

racc
Specifies record access byte.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

FCS$OFID$x

Description

The **FCS$OFID$x** functions open a file by using information stored in the file identification field of the filename block in the FDB (not in the default filename block). The suffixes (x) have the following meanings:

Suffix	Meaning
A	Append (add) data to the end of an existing file.
M	Modify an existing file without changing its length.
R	Read an existing file.
U	Update an existing file and extend its length, if necessary.
W	Write (create) a new file.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$OFNB$x

The **FCS$OFNB$x** functions open a file by using file name information in the filename block.

Format

#include *<fcs.h>*

short FCS$OFNB$x *(***fcs$fdb** **fdb,* **short** *lun,* **short** **dspt,)* **short** *racc,* **char** **urba,* **short** *urbs,* **void** *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

lun
Specifies the LUN associated with the desired file.

dspt
Specifies a pointer to the data-set descriptor.

racc
Specifies record access byte.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

FCS$OFNB$x

Description

The **FCS$OFNB$x** functions differ from the **FCS$OFID$x** functions in two respects: they can be issued to create a new file, and they can be issued to open a file by filename block. The suffixes (x) have the following meanings:

Suffix	Meaning
A	Append (add) data to the end of an existing file.
M	Modify an existing file without changing its length.
R	Read an existing file.
U	Update an existing file and extend its length, if necessary.
W	Write (create) a new file.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$OPEN$x

The **FCS$OPEN$x** functions are generalized open routines for specifying file access.

Format

#include *<fcs.h>*

short FCS$OPEN$x *(***fcs$fdb** **fdb,* **short** *lun,* **short** **dspt,)* **short** *racc,* **char** **urba,* **short** *urbs,* **void** *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

lun
Specifies the LUN associated with the desired file.

dspt
Specifies a pointer to the data-set descriptor.

racc
Specifies record access byte.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

FCS$OPEN$x

Description

The **FCS$OPEN$x** functions are used to open a file. The suffixes (x) have the following meanings:

Suffix	Meaning
A	Append (add) data to the end of an existing file.
M	Modify an existing file without changing its length.
R	Read an existing file.
U	Update an existing file and extend its length, if necessary.
W	Write (create) a new file.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$OPNS$x

The **FCS$OPNS$x** functions open and prepare a file for processing and allow shared access to that file.

Format

#include *<fcs.h>*

short FCS$OPNS$x (**fcs$fdb** **fdb*, **short** *lun*, **short** **dspt,)* **short** *racc*, **char** **urba*, **short** *urbs*, **void** *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

lun
Specifies the LUN associated with the desired file.

dspt
Specifies a pointer to the data-set descriptor.

racc
Specifies record access byte.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The suffixes (x) have the following meanings:

Suffix	Meaning
A	Append (add) data to the end of an existing file.
M	Modify an existing file without changing its length.
R	Read an existing file.
U	Update an existing file and extend its length, if necessary.
W	Write (create) a new file.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$OPNT$D

The **FCS$OPNT$D** function creates and opens a temporary file. The presumption in issuing the **FCS$OPNT$D** function is that the created file is to be used only once.

Format

#include *<fcs.h>*

short FCS$OPNT$D (**fcs$fdb** **fdb*, **short** *lun*, **short** **dspt*, **short** *racc*, **char** **urba*, **short** *urbs*, **void** *(*err)()*)

Arguments

fdb
Specifies a pointer to the associated FDB.

lun
Specifies the LUN associated with the desired file.

dspt
Specifies a pointer to the data-set descriptor.

racc
Specifies record access byte.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

FCS$OPNT$D

Description

The **FCS$OPNT$D** function creates and opens a temporary file. This file cannot be opened by another program. When the file is closed, it is deleted; its space is returned to the pool of available storage for reallocation.

NOTE

> If the **FCS$OPNT$D** function is used for a temporary file containing sensitive information, it is recommended that you zero the file before closing it, or reformat the disk to destroy the sensitive information. (Although a temporary file is deleted after use, the information physically remains on the volume until written over with another file, and it could be analyzed by unauthorized users.)

Return Values

1	Indicates success.
0	Indicates failure.

FCS$OPNT$W

The **FCS$OPNT$W** function creates and opens a temporary file for processing data.

Format

#include *<fcs.h>*

short FCS$OPNT$W *(***fcs$fdb** **fdb,* **short** *lun,* **short** **dspt,)*
short *racc,* **char** **urba,* **short** *urbs,*
void *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

lun
Specifies the LUN associated with the desired file.

dspt
Specifies a pointer to the data-set descriptor.

racc
Specifies record access byte.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

FCS$OPNT$W

Description

The **FCS$OPNT$W** function creates and opens a temporary file for some special purpose of limited duration. If a temporary file is to be used only once, it is best created through the **FCS$OPNT$D** function described above.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$PARSE

The **FCS$PARSE** function performs any necessary logical expansion and parses the resultant string.

Format

#include *<fcs.h>*

short FCS$PARSE (**fcs$fdb** **fdb,* **fcs$fnb** **fnb,* **short** **dsd,* **fcs$fnb** **fnb)*

Arguments

fdb
Specifies a pointer to the associated FDB.

fnb
Specifies a pointer to the filename block to be filled in.

dsd
Specifies a pointer to the desired data-set descriptor.

fnb
Specifies a pointer to the default filename block.

Description

The **FCS$PARSE** function first zeros the filename block and then stores the filename information into the filename block.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$POINT

The **FCS$POINT** function points to a byte or record within a specified file.

Format

#include *<fcs.h>*

short FCS$POINT (**fcs$fdb** **fdb*, **short** *highbits*, **short** *lowbits*, **short** *bytenum*)

Arguments

fdb
Specifies a pointer to the associated FDB.

highbits
Specifies the high-order bits of the virtual block number.

lowbits
Specifies the low-order bits of the virtual block number.

bytenum
Specifies the number of the next byte within the virtual block.

Description

The **FCS$POINT** function positions a file pointer to a specified byte in a specified virtual block. Use of this function is restricted to files accessed with the **FCSGET** and **FCSPUT** functions.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$POSIT

The **FCS$POSIT** function returns specified record position information.

Format

#include *<fcs.h>*

short FCS$POSIT *(***fcs$fdb** **fdb,* **short** **highbits,* **short** **lowbits,* **short** **bytenum)*

Arguments

fdb
Specifies a pointer to the associated FDB.

highbits
Specifies a pointer to the location to store the high-order bits of the virtual block number.

lowbits
Specifies a pointer to the location to store the low-order bits of the virtual block number.

bytenum
Specifies a pointer to the location to store the number of the next byte within the virtual block.

Description

The **FCS$POSIT** function calculates the virtual block number and the byte number locating the beginning of a specified record. Unlike the **FCS$POSRC** function, which sets up the position information of the file to the specified record, **FCS$POSIT** calculates the positional information of a specified record, so that a **FCS$POINT** operation can be performed.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$POSRC

The **FCS$POSRC** function sets up the position information for a file to a specified fixed-length record within a file.

Format

#include *<fcs.h>*

short FCS$POSRC *(***fcs$fdb** **fdb)*

Arguments

fdb
Specifies a pointer to the associated FDB.

Description

The **FCS$POSRC** function sets up the position information for a file to a specified fixed-length record within a file. This function is used to perform random access **FCSPUT** operations in locate mode.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$PPASC

The **FCS$PPASC** function converts a binary UIC directory string to ASCII.

Format

#include *<fcs.h>*

void FCS$PPASC *(***char** ***name,* **short** *uic,* **short** *control)*

Arguments

*****name***
Specifies the address of a storage area holding the ASCII string.

uic
Contains the UIC.

control
Contains the control code.

Description

The **FCS$PPASC** function converts a binary UIC to its corresponding ASCII directory string.

Return Values

None.

FCS$PRINT$

The **FCS$PRINT$** function queues a file for printing on a specified device.

Format

#include *<fcs.h>*

short FCS$PRINT$ *(***fcs$fdb** **fdb,* **void** *(*err)())*

Arguments

fdb
Specifies a pointer to the associated FDB.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The **FCS$PRINT$** function queues a file for printing on a specified device. The device must be a unit record, carriage-controlled device, such as a line printer or terminal. The default device is a line printer (LP).

Return Values

1	Indicates success.
0	Indicates failure.

FCS$PRSDI

The **FCS$PRSDI** function is similar to **FCS$PARSE** but performs only those operations associated with requisite directory identification information.

Format

#include *<fcs.h>*

short FCS$PRSDI **(fcs$fdb** **fdb,* **fcs$fnb** **fnb,* **short** **dsd,* **fcs$fnb** **fnb)*

Arguments

fdb
Specifies a pointer to the desired FDB.

fnb
Specifies a pointer to the desired filename block.

dsd
Specifies a pointer to the desired data-set descriptor.

fnb
Specifies a pointer to the desired default filename block.

Description

The **FCS$PRSDI** function performs a **FCS$PARSE** operation on the directory identification information field in the specified data-set descriptor or default filename block. The **FCS$PRSDI** function does not perform any logical name expansion.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$PRSDV

The **FCS$PRSDV** function works the same as **FCS$PARSE** but performs only those operations associated with requisite device and unit information.

Format

#include *<fcs.h>*

short FCS$PRSDV (**fcs$fdb** **fdb*, **fcs$fnb** **fnb*, **short** **dsd*, **fcs$fnb** **fnb*)

Arguments

fdb
Specifies a pointer to the desired FDB.

fnb
Specifies a pointer to the desired filename block.

dsd
Specifies a pointer to the desired data-set descriptor.

fnb
Specifies a pointer to the desired default filename block.

Description

The **FCS$PRSDV** function zeros the filename block, calls the **FCS$PARSE** routine to operate on the device and unit fields in the specified data-set descriptor or default filename block, and assigns the LUN contained in the offset location of the specified FDB.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$PRSFN

The **FCS$PRSFN** function works the same as **FCS$PARSE** but performs only operations associated with requisite file name, file type, and file version information.

Format

#include *<fcs.h>*

short FCS$PRSFN *(***fcs$fdb** **fdb,* **fcs$fnb** **fnb,* **short** **dsd,* **fcs$fnb** **fnb)*

Arguments

fdb
Specifies a pointer to the desired FDB.

fnb
Specifies a pointer to the desired filename block.

dsd
Specifies a pointer to the desired data-set descriptor.

fnb
Specifies a pointer to the desired default filename block.

Description

The **FCS$PRSFN** function performs a **FCS$PARSE** operation on the file name, file type, and file version information fields in the specified data-set descriptor or default filename block. It does not perform any logical name expansion.

Return Values

1	Indicates success.
0	Indicates failure.

FCSPUT

The **FCSPUT** function writes logical data records to a file.

Format

#include *<fcs.h>*

short FCSPUT (**fcs$fdb** **fdb*, **char** **urba*, **short** *urbs,)*
void *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

If the **FCSPUT** function is operating in random access mode, the number of the record to be written is maintained by FCS in the offset location of the associated FDB. This value increases by one after each **FCSPUT** or **FCSPUTR** operation to point to the next sequential record position.

Return Values

1	Indicates success.
0	Indicates failure.

FCSPUTR

The **FCSPUTR** function writes fixed-length records to a file in random mode.

Format

#include *<fcs.h>*

short FCSPUTR (**fcs$fdb** **fdb*, **char** **urba*, **short** *urbs*, **short** *lrcnm*, **short** *hrcnm*, **void** *(*err)()*)

Arguments

fdb
Specifies a pointer to the associated FDB.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

lrcnm
Specifies the low-order 16 bits of the number of the record to be read.

hrcnm
Specifies the high-order 15 bits of the number of the record to be read.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The **FCSPUTR** function differs from the **FCSPUT** function in that it allows the specification of the desired record number.

Return Values

1	Indicates success.
0	Indicates failure.

FCSPUTS

The **FCSPUTS** function writes records to a file in sequential mode.

Format

#include *<fcs.h>*

short FCSPUTS (**fcs$fdb** **fdb*, **char** **urba*, **short** *urbs,*)
void *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

urba
Specifies a pointer to the record buffer.

urbs
Specifies the numeric value that defines the size (in bytes) of the record buffer.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The **FCSPUTS** function is specifically for use in an overlaid environment in which the amount of memory available to the program is limited and files are to be written in sequential mode.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$RDFDR

The **FCS$RDFDR** function reads a directory string descriptor.

Format

#include *<fcs.h>*

void FCS$RDFDR *(***short** **size,* **char** ***pdds)*

Arguments

size
Specifies a location to store the size (in bytes) of the default directory string.

pdds
Specifies a location to store the default directory string.

Description

The **FCS$RDFDR** function reads the default directory string descriptor words previously written by the **FCS$WDFDR** function.

Return Values

None.

FCS$RDFFP

The **FCS$RDFFP** function reads the default file protection word in a location in the program section of the FSR.

Format

#include *<fcs.h>*

void FCS$RDFFP *(***short** **uic)*

Arguments

uic
Is a pointer to a location to store the default protection word.

Description

FCS uses the default file protection to establish the default file protection values for the new file. The **FCS$RDFFP** function allows the user to read the current default file protection word.

Return Values

None.

FCS$RDFUI

The **FCS$RDFUI** function reads the default UIC.

Format

#include *<fcs.h>*

void FCS$RDFUI **(short** **uic)*

Arguments

uic
Specifies a pointer to a location to store the binary-encoded default UIC.

Description

The **FCS$RDFUI** function reads the default UIC. Unlike the default directory string descriptor that describes an ASCII string, the default UIC is maintained as a binary value.

Return Values

None.

FCS$READ$

The **FCS$READ$** function reads virtual data blocks from a file.

Format

#include *<fcs.h>*

short FCS$READ$ (**fcs$fdb** **fdb*, **char** **bkda*, **short** *bkds,)*
long **bkvd*, **short** *bkef*, **short** **bkst*,
void *(*bkdn)()*, **void** *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

bkda
Specifies a pointer to the I/O block buffer.

bkds
Specifies the size (in bytes) of the virtual block to be written.

bkvb
Specifies a pointer to a 2-word block containing the number of the virtual block to be written.

bkef
Specifies the event flag number used in synchronizing block I/O operations.

bkst
Specifies a pointer to the IOSB.

bkdn
Specifies the entry point address of an AST service routine.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The **FCS$READ$** function is issued to read a virtual block of data to a block-oriented device, for example, magnetic tape or disk.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$REMOV

The **FCS$REMOV** function deletes an entry from a directory by file name.

Format

#include *<fcs.h>*

short FCS$REMOV *(***fcs$fdb** **fdb,* **fcs$fnb** **fnb)*

Arguments

fdb
Specifies a pointer to the desired FDB.

fnb
Specifies a pointer to the filename block.

Description

The **FCS$REMOV** function deletes only a specified directory entry; it does not delete the associated file.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$RENAM

The **FCS$RENAM** function changes the name of a file in its associated directory.

Format

#include *<fcs.h>*

short FCS$RENAM *(***fcs$fdb** **oldfdb,* **fcs$fdb** **newfdb)*

Arguments

oldfdb
Specifies a pointer to the FDB associated with the file with the original name.

newfdb
Specifies a pointer to the FDB containing the desired file name information, LUN assignment, and the event flag.

Description

If the renamed file is open when the **FCS$RENAM** is called, that file is closed before the renaming operation is attempted.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$RFOWN

The **FCS$RFOWN** function reads the contents of the file owner word in the program section.

Format

#include *<fcs.h>*

short FCS$RFOWN **(short** ***fow)**

Arguments

fow
Specifies a pointer to a location to store the file owner word.

Description

The **FCS$RFOWN** function reads the contents of the file owner word.

Return Values

None.

FCS$TRNCL

The **FCS$TRNCL** function truncates a file to the logical end of the file, deallocates any space beyond that point, and closes the file.

Format

#include *<fcs.h>*

short FCS$TRNCL *(**fcs$fdb** *fdb)*

Arguments

fdb
Specifies a pointer to the associated FDB.

Description

The **FCS$TRNCL** function truncates a file to the logical end of the file. The file must have been opened with both write and extend privileges; otherwise, the truncation will fail.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$WAIT$

The **FCS$WAIT$** function suspends program execution until a requested block input/output transfer is completed.

Format

#include *<fcs.h>*

short FCS$WAIT$ (**fcs$fdb** **fdb*, **short** *bkef*, **short** **bkst,)*
void *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

bkef
Specifies the event flag number to be used for synchronizing block I/O operations.

bkst
Specifies a pointer to the IOSB.

err
Specifies the address of the optional, user-coded, error-handling routine.

Description

The **FCS$WAIT$** function, which is issued only with **FCS$READ$** and **FCS$WRITE$** operations, suspends program execution until the requested block I/O transfer is completed. This function may be used to synchronize a block I/O operation that depends on the successful completion of a previous block I/O transfer.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$WDFDR

The **FCS$WDFDR** function writes directory string descriptors in program section $$FSR2.

Format

#include *<fcs.h>*

void FCS$WDFDR (**short** *size*, **char** **pdds*)

Arguments

size
Specifies the size (in bytes) of the default directory string.

pdds
Specifies a pointer to the default directory string.

Description

The **FCS$WDFDR** function creates the default directory string descriptor words read by the **FCS$RDFDR** function.

Return Values

None.

FCS$WDFFP

The **FCS$WDFFP** function writes a new default file protection word into the program section $$FSR2.

Format

#include *<fcs.h>*

void FCS$WDFFP *(***short** *uic)*

Arguments

uic
Specifies the new default protection word to be written.

Description

FCS uses the default file protection word only when a file is created to establish the default file protection values for the new file.

Return Values

None.

FCS$WDFUI

The **FCS$WDFUI** function writes the default UIC to a program section in the FSR.

Format

#include *<fcs.h>*

void FCS$WDFUI *(***short** *uic)*

Arguments

uic
Specifies the binary-encoded default UIC.

Description

The **FCS$WDFUI** function writes a new default UIC. Unlike the default directory string descriptor that describes an ASCII string, the default UIC is maintained as a binary value. Unless the default UIC is changed through the **FCS$WDFUI** function, the default UIC always corresponds to the UIC under which the task is running.

Return Values

None.

FCS$WFOWN

The **FCS$WFOWN** function initializes the file owner word in the program section $$FSR2.

Format

#include *<fcs.h>*

void FCS$WFOWN **(short** *fow)*

Arguments

fow
Contains the file owner word to be written.

Description

The **FCS$WFOWN** function initializes the file owner word (UIC).

Return Values

None.

FCS$WRITE$

The **FCS$WRITE$** function writes virtual data blocks to a file.

Format

#include *<fcs.h>*

short FCS$WRITE$ *(***fcs$fdb** **fdb,* **char** **bkda,* **short** *bkds,)*
long **bkvd,* **short** *bkef,* **short** **bkst,*
void *(*bkdn)(),* **void** *(*err)()*

Arguments

fdb
Specifies a pointer to the associated FDB.

bkda
Specifies a pointer to the I/O block buffer.

bkds
Specifies the size (in bytes) of the virtual block to be written.

bkvb
Specifies a pointer to a 2-word block containing the number of the virtual block to be written.

bkef
Specifies the event flag number used in synchronizing block I/O operations.

bkst
Specifies a pointer to the IOSB.

bkdn
Specifies the entry point address of an AST service routine.

err
Specifies a pointer to the optional, user-coded, error-handling routine.

Description

The **FCS$WRITE$** function is issued to write a virtual block of data to a block-oriented device, for example, magnetic tape or disk.

Return Values

1	Indicates success.
0	Indicates failure.

FCS$XQIO

The **FCS$XQIO** function executes a specified QIO$ function and waits for its completion.

Format

#include *<fcs.h>*

short FCS$XQIO (**fcs$fdb** **pfdb*, **short** *function*, **short** *nparams*, **short** **paramlist)*

Arguments

pfdb
Specifies a pointer to the desired FDB.

function
Specifies the desired function code.

nparams
Specifies the number of optional parameters, if any.

paramlist
Specifies a pointer to the beginning address of the list of optional directive parameters.

Description

The **FCS$XQIO** function executes a specified QIO$ function and waits for its completion.

Return Values

None.

3 RMS Extension Library Macros

RMS$CLOSE

The **RMS$CLOSE** function closes an open file.

Format

#include *<rmsops.h>*

void RMS$CLOSE *(***struct FAB** **pfab, . . .);*

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$CLOSE** macro closes an open file.

Return Values

None.

RMS$CONNECT

The **RMS$CONNECT** function connects a record stream to an open file and initializes the stream context.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$CONNECT

void RMS$CONNECT **(struct S_RAB** **prab, ...);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$CONNECT** macro connects a record stream to an open file and initializes the stream context.

Return Values

None.

RMS$CREATE

The **RMS$CREATE** function creates a new file and opens it for processing.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$CREATE

void RMS$CREATE **(struct FAB** **pfab, . . .);*

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$CREATE** function creates a new file and opens it for processing.

Return Values

None.

RMS$DELETE

The **RMS$DELETE** function removes a record from a relative or indexed file.

Format

#include *<rmsops.h>*

#pragma **linkage fortran RMS$DELETE**

void RMS$DELETE (**struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$DELETE** function removes a record from a relative or indexed file. The target of the DELETE operation is the current record. The current record must be locked. It was automatically locked when the current-record context was set, but you must not have unlocked it with a FREE operation.

Return Values

None.

RMS$DISCONNECT

The **RMS$DISCONNECT** function terminates a stream and disconnects the internal resources it was using.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$DISCONNECT

void RMS$DISCONNECT **(struct S_RAB** ***prab, . . .);***

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$DISCONNECT** macro terminates a stream and disconnects the internal resources it was using. You cannot re-establish the same stream context by reconnecting the stream with the CONNECT operation.

Return Values

None.

RMS$DISPLAY

The **RMS$DISPLAY** function Writes values into control block fields.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$DISPLAY

void RMS$DISPLAY *(***struct FAB** **pfab, . . .);*

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$DISPLAY** writes values into control block fields. The DISPLAY operation does not alter the file in any way.

Return Values

None.

RMS$ENTER

The **RMS$ENTER** function inserts a file name into a directory file. This macro is not supported on RSTS/E.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$ENTER

void RMS$ENTER (**struct FAB** **pfab, . . .*);

Arguments

pfab
Specifies a pointer to the associated FAB.

...
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$ENTER** function inserts a file into a directory file.

Return Values

None.

RMS$ERASE

The **RMS$ERASE** function erases a file and deletes its directory entry.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$ERASE

void RMS$ERASE *(***struct FAB** **pfab, . . .);*

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$ERASE** function erases a file and deletes its directory entry. Erasing a file, marks the file for deletion, but does not necessarily erase the file immediately. The file is erased when it has no accessing programs. The allocation for the file is released for use in other files.

Return Values

None.

RMS$EXTEND

The **RMS$EXTEND** function extends the allocation for an open file.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$EXTEND

void RMS$EXTEND *(***struct FAB** **pfab, . . .);*

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$EXTEND** function extends the allocation for an open file.

Return Values

None.

RMS$FIND

The **RMS$FIND** function with sequential or record file access transfers a record or part of a record from a file to an I/O buffer. The **RMS$FIND** function with key access transfers a record or part of a record from a sequential disk file, a relative file, or an indexed file to an I/O buffer.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$FIND

void RMS$FIND (struct S_RAB ***prab, . . .);***

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$FIND** function with sequential or record file access transfers a record or part of a record from a file to an I/O buffer. The **RMS$FIND** function with key access transfers a record or part of a record from a sequential disk file, a relative file, or an indexed file to an I/O buffer.

Return Values

None.

RMS$FLUSH

The **RMS$FLUSH** function writes any unwritten buffers for a stream.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$FLUSH

void RMS$FLUSH *(***struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

...
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded error-handling routine.

psucc
Specifies the address of the optional, user-coded success-handling routine.

Description

The **RMS$FLUSH** function writes any unwritten buffers for a stream. The FLUSH operation does not affect stream context, except that the current-record context is undefined for a following TRUNCATE or UPDATE operation.

Return Values

None.

RMS$FREE

The **RMS$FREE** function frees a locked bucket for a stream.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$FREE

void RMS$FREE **(struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$FREE** function frees a locked bucket for a stream.

Return Values

None.

RMS$GET

The **RMS$GET** function with sequential or record file access transfers a record from a file to an I/O buffer and a user buffer. The **RMS$GET** function with key access transfers a record from a sequential disk file, a relative file, or an indexed file to an I/O buffer and a user buffer.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$GET

void RMS$GET **(struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$GET** function with sequential or record file access transfers a record from a file to an I/O buffer and to a user buffer. The **RMS$GET** function with key access transfers a record from a sequential disk file, a relative file, or an indexed file to an I/O buffer and to a user buffer.

Return Values

None.

RMS$NXTVOL

The **RMS$NXTVOL** function advances the context for a stream to the beginning of the next magnetic tape volume. This macro is not supported on RSTE/E.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$NXTVOL

void RMS$NXTVOL *(***struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$NXTVOL** function advances the context for a stream to the beginning of the next magnetic tape volume. This macro is not supported on RSTS/E.

Return Values

None.

RMS$OPEN

The **RMS$OPEN** function opens a file for processing by the calling task.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$OPEN

void RMS$OPEN (struct FAB **pfab, . . .);*

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$OPEN** function opens a file for processing by the calling task.

Return Values

None.

RMS$PARSE

The **RMS$PARSE** function analyzes a file specification.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$PARSE

void RMS$PARSE (struct FAB ***pfab, . . .);***

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$PARSE** function analyzes a file specification.

Return Values

None.

RMS$PUT

The **RMS$PUT** function with sequential access transfers a record from a user buffer to an I/O buffer and to a file. The **RMS$PUT** function with key access transfers a record from a user buffer to an I/O buffer and to a sequential disk file, a relative file, or an indexed file.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$PUT

void RMS$PUT **(struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$PUT** function with sequential access transfers a record from a user buffer to an I/O buffer and to a file. The **RMS$PUT** function with key access transfers a record from a user buffer to an I/O buffer and to a sequential disk file, a relative file, or an indexed file.

Return Values

None.

RMS$READ

The **RMS$READ** function transfers blocks to an I/O buffer.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$READ

void RMS$READ (struct S_RAB ***prab, . . .);***

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$READ** function transfers blocks to an I/O buffer.

Return Values

None.

RMS$RELEASE

The **RMS$RELEASE** function is supplied for VMS compatibility only; it has no effect.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$RELEASE

void RMS$RELEASE (**struct FAB** **pfab, . . .);*

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Return Values

None.

RMS$REMOVE

The **RMS$REMOVE** function removes the directory entry for a file. This macro is not supported on RSTS/E.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$REMOVE

void RMS$REMOVE *(***struct FAB** **pfab, . . .);*

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$REMOVE** function removes the directory entry for a file. This macro is not supported on RSTS/E.

Return Values

None.

RMS$RENAME

The **RMS$RENAME** function changes the directory entry for a file.

Format

#include *<rmsops.h>*

#pragma **linkage fortran RMS$RENAME**

void $RMREN *(***struct FAB** **pfab1,* **void** *(*perr) (),* **void** *(*psucc) (),* **struct FAB** **pfab2);*

Arguments

pfab1
Specifies a pointer to the FAB for the operation.

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

pfab2
Specifies a pointer to the FAB that holds the new file specification.

Description

The **RMS$RENAME** function changes the directory entry for a file.

Return Values

None.

RMS$REWIND

The **RMS$REWIND** function resets the context for a stream to the beginning-of-file. This macro is not supported on RSTS/E.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$REWIND

void RMS$REWIND *(***struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$REWIND** function resets the context for a stream to the beginning-of-file. This macro is not supported on RSTS/E.

Return Values

None.

RMS$SEARCH

The **RMS$SEARCH** function scans a directory, returns a file specification, and identifies in NAM block fields.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$SEARCH

void RMS$SEARCH **(struct FAB** ***pfab, . . .);***

Arguments

pfab
Specifies a pointer to the associated FAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$SEARCH** function scans a directory, returns a file specification, and identifies in NAM blocks.

Return Values

None.

RMS$SPACE

The **RMS$SPACE** function moves a magnetic tape backward or forwards. This macro is not supported on RSTS/E.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$SPACE

void RMS$SPACE *(***struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$SPACE** function moves a magnetic tape backwards or forwards. This macro is not supported on RSTS/E.

Return Values

None.

RMS$TRUNCATE

The **RMS$TRUNCATE** function removes records from the latter part of a sequential file.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$TRUNCATE

void RMS$TRUNCATE *(***struct S_RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$TRUNCATE** function removes records from the latter part of a sequential file.

Return Values

None.

RMS$UPDATE

The **RMS$UPDATE** function transfers a record from a user buffer to a disk file, overwriting the existing record.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$UPDATE

void RMS$UPDATE (struct S_RAB **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$UPDATE** function transfers a record from a user buffer to a disk file, overwriting the existing record.

Return Values

None.

RMS$WAIT

The **RMS$WAIT** function suspends processing until an outstanding asynchronous operation on the stream is completed. This macro is not supported on RSTS/E.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$WAIT

void RMS$WAIT *(***struct RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$WAIT** function suspends processing until an outstanding asynchronous operation on the stream is completed. This macro is not supported on RSTS/E.

Return Values

None.

RMS$WRITE

The **RMS$WRITE** function writes blocks to file.

Format

#include *<rmsops.h>*

#pragma linkage fortran RMS$WRITE

void RMS$WRITE **(struct RAB** **prab, . . .);*

Arguments

prab
Specifies a pointer to the associated RAB.

. . .
Specifies the following optional addresses:

perr
Specifies the address of the optional, user-coded, error-handling routine.

psucc
Specifies the address of the optional, user-coded, success-handling routine.

Description

The **RMS$WRITE** function writes blocks to a file.

Return Values

None.

Appendix A

PDP–11 C and VAX C Compatibility Issues

Because of architectural differences between the PDP–11 and the VAX–11 systems and because the PDP–11 does not support all of the features of VAX C, some incompatibilities exist between the two implementations. This appendix describes the major differences between PDP–11 C and VAX C, as summarized in the following list:

1. Errors in program structure are handled differently by PDP–11 C than by VAX C. The following is a list of these differences:
   - If the user attempts to reference a parameter that is a redeclaration of one of the function's formal parameters, PDP–11 C issues an error message; VAX C issues a warning message.
   - If a numeric constant contains an illegal character or is otherwise invalid, PDP–11 C issues an error message; VAX C issues a warning message and ignores the illegal characters.
2. PDP–11 C does not support 8 and 9 as octal constant digits. An error is issued if an invalid octal constant is specified.
3. If defined, the logical name C$INCLUDE specifies the directory where PDP–11 C is to search for header files which are included by using the **#include** preprocessing directive. In a VMS compilation environment, the logical name may specify a search list.
4. If the specified header file cannot be found in the device/directory searched, PDP–11 C attempts to translate the user-defined logical name C$INCLUDE in the VMS and RSX–11M–PLUS compilation environments. In the VMS compilation environments, C$INCLUDE may specify a search list.
5. The module name and ident of PDP–11 C's **#module** preprocessing directive are limited to no more than six alphanumeric characters, space, dollar sign ($), or dot (.). Additional characters are ignored.

6. In PDP–11 C, preprocessor directives may begin anywhere on a line; however, VAX C requires preprocessor directives to begin with the # character as the first character of the line.
7. PDP–11 C defines **CC$gfloat** as 0, indicating that the G-float format is not being used for double objects; VAX C expands the **CC$gfloat** macro to 1 if the /G_FLOAT qualifier is asserted, 0 if not asserted.
8. PDP–11 C does not use the RMS file type RMS STREAM_LF as its external representation for binary and text streams.
9. PDP–11 C expands the macro **L_tmpnam** to the integer constant 13; VAX C expands it to a value of 255.
10. PDP–11 C does not provide the optional file attribute arguments for the **fopen** function.
11. PDP–11 C prints a pointer as an unsigned octal integer when the **fprintf** function is used with the conversion character "p".
12. PDP–11 C does not define all existing RMS masks and fields that are defined in VAX C.
13. In the header files that define RMS structures, the l_ convention used by VAX C for naming structure members that are pointers was retained for compatibility with the VAX C definitions for those items; however, the item is a 16-bit quantity rather than a 32-bit quantity.
14. The **RAB** data structure on the PDP–11 is two different sizes, one for synchronous RABs and one for asychronous RABs. The structure tags, **SRAB** and **ARAB** respectively, are used to identify these two different data structures. The existing code for VAX C compiled by PDP–11 C will have undefined structures for each RAB structure; therefore, when porting source code from VAX C to PDP–11 C, you need to determine which type of RAB is desired and to change the RAB to ARAB or SRAB as needed.
15. The RMS functions available through PDP–11 C do not return a value.
16. PDP–11 C adds the keyword [NO]MACHINE to the /SHOW switch rather than having a separate /[NO]MACHINE switch.
17. PDP–11 C supports the following command-line switches, which are not supported by VAX C:
    - CACHE
    - CODE
    - COMMAND
    - ENVIRONMENT
    - ERROR_LIMIT

- INTEGER_SIZE
- MACRO
- MEMORY
- MODULE
- TERMINAL
- WORK_FILE_SIZE

18. PDP–11 C does not support the following VAX C command-line switches:
    - ANALYSIS_DATA
    - CROSS_REFERENCE
    - DEBUG
    - DIAGNOSTICS
    - G_FLOAT
    - LIBRARY
    - [NO]MACHINE
    - PARALLEL
    - PRECISION
    - PREPROCESS_ONLY
    - STANDARD=[NO]PORTABLE
19. In PDP–11 C, objects may be declared to be of type **long double** but not of type **long float**. In VAX C, objects may be declared to be of type **long float** but not of type **long double**.
20. For compatibility with VAX C, the following functions are defined in the supplied standard header files. They are defined only when compiling with the /NOSTANDARD switch.
    - These functions are defined for VAX C compatibility. Each function is described in PDP–11 C Standard Library Macros and Functions of the Reference Section.

      cabs
      fgetname
      hypot
      isascii
      sleep
      toascii
      _tolower
      _toupper

- The type cabs_t and structure type CABS_T are defined as follows:

```
typedef struct CABS_T {double __x, __y;} cabs_t;
```

- These macros are defined for VAX C compatibility:

NSIGNALS	Number of signals
OPEN_MAX	Number of files that can be simultaneously opened (ANSI equivalent is FOPEN_MAX)
PATH_MAX	Size of maximum path name (ANSI equivalent is FILENAME_MAX)
SEEK_EOF	Equivalent to ANSI SEEK_END
STRINGS_MATCH	Value returned by standard library functions when strings match
CLK_TCK	Equivalent to ANSI CLOCKS_PER_SEC

Appendix B

PDP–11 C Run-Time Modules and Entry Points

This appendix summarizes the modules and entry points in the PDP–11 C Run-Time System. Table B–1 lists the entry points and the modules in the library and describes their function.

Table B–1: PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
abort	C$ABRT	Aborts the current process.
abs	C$ABS	Integer absolute value math library function.
acos	C$ACOS	Arc cosine math library function.
__alr50	C$ASL5	Converts first six characters in the input string to an unsigned 32-bit integer corresponding to the radix-50 translation.
asctime	C$ASTM	Converts broken-down time into a character string.
asin	C$ASIN	Arc sine math library function.
__asr50	C$ASR5	Converts the first three characters of the input string to an unsigned 16-bit integer corresponding to the radix-50 translation.
atan	C$ATAN	Arc tangent math library function.
atan2	C$ATN2	Arc tangent math library function.

(continued on next page)

Table B–1 (Cont.): PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
atexit	C$ATEX	Registers functions to be called without arguments at program termination.
atof	C$ATOF	Converts ASCII to floating-point binary.
atoi	C$ATOI	Converts ASCII to integer binary.
atol	C$ATOL	Converts long ASCII to binary.
bsearch	C$BSCH	Binary search routine.
cabs	C$CABS	Returns the square root of two squared arguments.
calloc	C$CLLC	Allocates and clears storage.
cc$rms_fab	C$RMS_PROTOTYPES	File access block prototype.
cc$rms_nam	C$RMS_PROTOTYPES	Block naming prototype.
cc$rms_rab	C$RMS_PROTOTYPES	Access-block recording prototype.
cc$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$RMS_PROTOTYPES	Terminal characteristics of the extended attribute block prototype.
ceil	C$CEIL	Ceiling math library function.
clearerr	$PCLEA	Clears end-of-file error.

(continued on next page)

Table B–1 (Cont.): PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
clock	C$CLCK	Determines CPU time.
cos	C$COS	Cosine math library function.
cosh	C$COSH	Hyperbolic cosine math library function.
ctime	C$CTIM	Converts time to an ASCII string.
difftime	C$DFTM	Computes the difference between two times.
div	C$DIV	Computes the quotient and remainder.
exit	C$EXIT	Closes files and exits.
exp	C$EXP	Base-e exponentiation math function.
fabs	C$FABS	Absolute math function.
__fbuf	C$FGBF	Returns the current buffer length associated with a file pointer.
fclose	$PCLOS	Closes a file.
feof	$PEOF	Tests the end-of-file indicator.
ferror	$PERRO	Tests the error indicator.
fflush	$PFLUS	Flushes a file buffer.
__fger	C$FGER	Returns the low level error code that is associated with a previously called file operation.
fgetc	$PFGTC	Gets a character from a file.
fgetname	C$FGNM	Returns a pointer to a file specification associated with a file variable.
fgetpos	C$PGETP	Stores the current value of the file position indicator for the stream pointed to by **stream**.
fgets	$PFGTS	Gets a string from a file.

(continued on next page)

Table B–1 (Cont.): PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
__fgnm	C$FGNM	Returns a pointer to a file specification associated with a file variable.
floor	C$FLOR	Returns the largest integer that is less than or equal to its argument.
__flun	C$FGLN	Returns the logical unit number associated with a file pointer.
fmod	C$FMOD	Computes the floating-point remainder of X/Y.
fopen	$POPE	Opens a file by file pointer.
fprintf	$PFPRI	Formats a string to a file.
fputc	$PFPTC	Writes a character to a file.
fputs	$PFPTS	Writes a string to a file.
fread	$PREAD	Reads from a file.
__frec	C$FGRC	Returns the current record length associated with a file pointer.
free	C$FREE	Deallocates storage.
freopen	$PREOP	Closes and reopens a file.
frexp	C$FRXP	Extract fraction exponent math function.
fscanf	$PFSCA	Scans input from a file.
fseek	$PSEEK	Positions to an offset in a file.
fsetpos	$PSETP	Sets file position indicator.
ftell	$PTELL	Returns current offset in a file.
fwrite	$PWRIT	Writes to a file.
getc	$GETC	Gets a character from standard input.
getchar	C$GTCR	Reads a character from standard input.
getenv	C$GENV	Returns the value of the environment.

(continued on next page)

Table B–1 (Cont.): PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
gets	$PGETS	Gets a string from standard input.
gmtime	C$GMTM	Converts calendar time into broken-down time.
hypot	C$HYPT	Euclidean distance math library function.
__ischar	C$ISCH	Returns a nonzero integer if its argument is contained in the current character set.
labs	C$LABS	Returns the absolute value of an integer as long integer.
ldexp	C$LDXP	Power of 2 math library function.
ldiv	C$LDIV	Computes long integer quotient and remainder.
localeconv	C$LCON	Sets components of an object with type **struct lconv**.
localtime	C$LCTM	Places time in a time structure.
log	C$LOG	Logarithm base-e math library function.
log10	C$LG10	Logarithm base-10 math library function.
longjmp	C$LGJP	Returns to a **setjmp** entry point.
__lr50	C$L5TA	Converts an unsigned 32-bit radix-50 string to the corresponding 6-character ASCII character string.
malloc	C$MLLC	Allocates memory.
mblen	C$MBLN	Determines the number of bytes in multibyte character.
mbstowcs	C$MBCS	Converts the multibyte characters to a sequence of corresponding codes.
mbtowc	C$MBWC	Determines the number of bytes in multibyte character.

(continued on next page)

Table B–1 (Cont.): PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
memchr	C$MCHR	Locates the first occurrence of a character.
memcmp	C$MCMP	Compares the lexical values of two arrays.
memcpy	C$MCPY	Moves characters from one array to another.
memmove	C$MMOV	Moves characters from one array to another.
memset	C$MSET	Puts a given character in *n* bytes of an array.
mktime	C$MKTM	Converts the broken-down time into calendar time.
modf	C$MODF	Extract the fraction and the integer math function.
perror	$PPERR	Prints an error message.
pow	C$POW	Raise to a power math library function.
printf	$PPRIN	Formats a string to standard output.
puts	$PPUTS	Writes a string to standard output.
qsort	C$QSRT	Sorts an array of data objects.
raise	C$RASE	Generates a signal.
rand	C$RAND	Computes a random number.
realloc	C$RLLC	Changes the size of an area of storage.
remove	$PREMO	Deletes a file.
rename	$PRENA	Renames a file.
rewind	$PREWI	Returns to the beginning of the file.
scanf	$PSCAN	Formats input from the standard input.
setbuf	C$PSETB	Associates buffer with I/O file.

(continued on next page)

Table B–1 (Cont.): PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
setlocale	C$SLOC	Selects the part of the program's locale as specified by **category** and **locale**.
setvbuf	$PSETV	Establishes I/O buffering for a file.
signal	C$SIGL	Sets a signal.
sin	C$SIN	Sine math library function.
sinh	C$SINH	Hyperbolic sine math library function.
sleep	C$SLEP	Suspends execution for a specified time interval.
__sleep	C$SLEP	Suspends execution for a specified time interval.
sprintf	C$SPRR	Formats a string to a memory buffer.
sqrt	C$SQRT	Square root math library function.
srand	C$SRND	Reinitializes the random number generator.
__sr50a	C$S5TA	Converts an unsigned 16-bit radix-50 string to the corresponding 3-character ASCII character string.
sscanf	C$SSCR	Formats the input from memory.
strcat	C$SCAT	Concatenates two strings.
strchr	C$SCHR	Searches for a character in a string.
strcmp	C$SCMP	Compares two strings.
strcoll	C$SCOL	Compares two strings.
strcpy	C$SCPY	Moves a string to another string.
strcspn	C$SCSP	Searches a string for a character.
strerror	C$SERR	Translates an error message code.
strftime	C$SFTM	Converts time and date format to a user-defined format.

(continued on next page)

Table B–1 (Cont.): PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
strlen	C$SLEN	Determines the length of a string.
strncat	C$SNCA	Concatenates two strings.
strncmp	C$SNCM	Compares two strings.
strncpy	C$SNCP	Moves one string to another.
strpbrk	C$SPBK	Searches a string for a character.
strrchr	C$SRCH	Searches a string for a character.
strspn	C$SSPN	Searches a string for a character.
strstr	C$SSTR	Locates the first occurrence of a sequence of characters from one string pointed to by another string.
strtod	C$STOD	Converts a string to a double-precision number.
strtok	C$STOK	Locates text tokens in a given string.
strtol	C$STOL	Converts a character string into a long integer value.
strtoul	C$STUL	Converts a character string into an unsigned value.
strxfrm	C$SXFR	Transforms a string and places the results into an array.
system	C$SYTM	Passes 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	Gets the epoch time.
tmpfile	$PTMPF	Creates a temporary file.
tmpnam	C$PTMPN	Generates a temporary file name.
tolower	C$TLWR	Converts uppercase to lowercase.
toupper	C$TUPR	Converts lowercase to uppercase.

(continued on next page)

Table B–1 (Cont.): PDP–11 C Run-Time Entry Points

Entry Point	Module	Description
__tzset	C$TZSE	Sets time variables.
ungetc	C$PUNGE	Pushes a character back into the stream.
vfprintf	C$PVFPR	Prints formatted output.
vprintf	$PVPRI	Prints formatted output.
vsprintf	C$VSPR	Prints formatted output.
wcstombs	C$WCSB	Converts the sequence of codes corresponding to multibyte characters into multibyte characters.
wctomb	C$WCMB	Determines the number of bytes needed to represent multibyte character.

Index

D

E

F

G

H

I

L

S

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

PDP–11 C
Run-Time Library Reference Manual
AA–NA45B–TC

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
PKO3–1/D30
129 PARKER STREET
MAYNARD, MA 01754–9975

Do Not Tear – Fold Here and Tape

Reader's Comments

PDP–11 C
Run-Time Library Reference Manual
AA–NA45B–TC

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
PKO3–1/D30
129 PARKER STREET
MAYNARD, MA 01754–9975

Do Not Tear - Fold Here and Tape

DECLIT AA PDP11 NA45B

PDP-11 C run-time library reference manual

digital

digital

Printed in U.S.A.