VMS Software

VSI C Run-Time Library Reference Manual for OpenVMS Systems

Document Number: DO-VIBHAA-011

Publication Date: April 2024

Operating System and Version: VSI OpenVMS IA-64 Version 8.4-1H1 or higher
VSI OpenVMS Alpha Version 8.4-2L1 or higher
VSI OpenVMS x86-64 Version 9.2-1 or higher

Software Version: VSI C Version 7.4-1 for OpenVMS

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI C Run-Time Library Reference Manual for OpenVMS Systems

Legal Notice

Preface

This manual describes the VSI C Run-Time Library (C RTL) for OpenVMS Alpha, OpenVMS Integrity server, and OpenVMS x86-64 operating systems.

This manual provides reference information about the C RTL functions and macros that perform input/output (I/O) operations, character and string manipulation, mathematical operations, error detection, subprocess creation, system access, screen management, and emulation of selected UNIX features. It also notes portability concerns between operating systems, where applicable.

The VSI C Run-Time Library contains XPG4-compliant internationalization support, providing functions to help you develop software that can run in different languages and cultures.

The complete VSI C Run-Time Library needed for use with the VSI C and VSI C++ compilers is distributed with the OpenVMS Alpha, OpenVMS Integrity server, and OpenVMS x86-64 operating systems in both shared image and object module library form.

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

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

3. Document Structure

This manual has the following chapters, reference section, and appendixes:

- Chapter 1 provides an overview of the C RTL.
- Chapter 2 discusses the Standard I/O, Terminal I/O, and UNIX I/O functions.
- Chapter 3 describes the character, string, and argument-list functions.
- Chapter 4 describes the error-handling and signal-handling functions.
- Chapter 5 explains the functions used to create subprocesses.
- Chapter 6 describes the Curses Screen Management functions.
- Chapter 7 discusses the math functions.
- Chapter 8 explains the memory allocation functions.
- Chapter 10 describes the functions used to interact with the operating system.
- Chapter 11 gives an introduction to the facilities provided in the VSI C environment on OpenVMS systems for developing international software.
- Chapter 12 describes the date/time functions.

- Chapter 13 describes symbolic links and POSIX pathname support.
- The Reference Section describes all the functions in the C RTL.
- Appendix A contains version-dependency tables that list the C RTL functions supported on different OpenVMS versions.
- Appendix B lists the function prototypes that are duplicated in more than one header file.

4. Related Documents

The following documents may be useful when programming in VSI C for OpenVMS systems:

- *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/]—For C programmers who need information on using VSI C for OpenVMS systems.
- *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/]—Provides language reference information for VSI C on VSI OpenVMS systems.
- *VSI TCP/IP Services for OpenVMS Sockets API and System Services Programming*—For information on the socket routines used for writing Internet application programs for the OpenVMS product or other implementations of the TCP/IP protocol.
- *VSI TCP/IP Services for OpenVMS Guide to IPv6*—For information on TCP/IP Services for OpenVMS IPv6 features, how to install and configure IPv6 on your system, changes in the socket application programming interface (API), and how to port your applications to run in an IPv6 environment.
- *X/Open Portability Guide, Issue 3*—Documents what is commonly known as the XPG3 specification.
- *X/Open CAE Specification System Interfaces and Headers, Issue 4*—Documents what is commonly known as the XPG4 specification.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4,* Version 2—Documents what is commonly known as XPG4 V2.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 5*—Documents what is commonly known as the XPG5 specification.
- *Technical Standard. System Interfaces, Issue 6*—Combined Open Group Technical Standard and IEEE standard. IEEE Std 1003.1-2001, sometimes known as XPG6.
- *Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 2: Threads Extension [C Language]*—Documents what is also known as POSIX 1003.1c-1995.
- *ISO/IEC 9945-2:1993 - Information Technology - Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities*—Documents what is also known as ISO POSIX-2.
- *ISO/IEC 9945-1:1990 - Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) (C Language)*—Documents what is also known as ISO POSIX-1.
- *ANSI/ISO/IEC 9899:1999 - Programming Languages - C*—The C99 standard, published by ISO in December, 1999 and adopted as an ANSI standard in April, 2000.

- *ISO/IEC 9899:1990-1994 - Programming Languages - C, Amendment 1:Integrity*—Documents what is also known as ISO C, Amendment 1.
- *ISO/IEC 9899:1990[1992] - Programming Languages - C*—Documents what is also known as ISO C. The normative part is the same as *X3.159-1989, American National Standard for Information Systems - Programming Language C*, also known as ANSI C.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

7. Typographical Conventions

The conventions found in the following table are used in this document.

Convention	**Meaning**
OpenVMS systems	Refers to the OpenVMS operating system on all supported platforms, unless otherwise specified.
Ctrl/*x*	While holding down the Ctrl key, press the key specified by *x*.
`switch` statement `int` data type `fprintf` function `<stdio.h>` header file	Monospace type identifies language keywords and the names of C RTL functions and header files. Monospace type is also used when referring to a specific variable name used in an example.
arg1	Italic type indicates a placeholder, such as an argument or parameter name.

`float x;`
`.`
`.`
`.`
`x = 5;`
	A vertical ellipsis indicates that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.

`option, ...`	A horizontal ellipsis indicates that additional parameters, options, or values can be entered. A comma that precedes the ellipsis indicates that successive items must be separated by commas.
`[output-source, ...]`	Square brackets, in function synopses and a few other contexts, indicate that a syntactic element is optional. Square brackets are not optional, however, when used to delimit a directory name in an OpenVMS file specification or when used to delimit the dimensions of a multidimensional array in VSI C source code.

Convention	Meaning
[a | b]	Brackets surrounding two or more items separated by a vertical bar (|) indicate a choice; you must choose one of the two syntactic elements.
Δ	A delta symbol is used in some contexts to indicate a single ASCII space character.

8. Platform Labels

A **platform** is a combination of operating system and hardware that provides a distinct environment. This manual contains information applicable to the OpenVMS operating system running on x86-64, Itanium, and Alpha processors.

The information in this manual applies to all OpenVMS operating systems, except when specifically labeled as follows:

Label	Explanation
(Alpha system)	Specific to an HPE Alpha system running the OpenVMS Alpha operating system.
(Integrity server system)	Specific to an HPE Integrity server running the OpenVMS Integrity server operating system.
(x86-64 system)	Specific to an x86-64 server running the OpenVMS x86-64 operating environment.

Chapter 1. Introduction

The ISO/ANSI C standard defines a library of functions, as well as related types and macros, to be provided with any implementation of ANSI C. The *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] describes the ANSI-conformant library features common to all VSI C platforms. The *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/] provides a more detailed description of these routines and their use in the OpenVMS environment. It also documents additional header files, functions, types, and macros that are available on the OpenVMS system.

All library functions are declared in a *header file*. To make the contents of a header file available to your program, include the header file with an `#include` preprocessor directive. For example:

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

Each header file contains function prototypes for a set of related functions, and defines any types and macros needed for their use.

To list the header files, use the following commands:

```
$ LIBRARY/LIST SYS$LIBRARY:SYS$STARLET_C.TLB
$ LIBRARY/LIST SYS$LIBRARY:DECC$RTLDEF.TLB
$ DIR SYS$COMMON:[DECC$LIB.REFERENCE.DECC$RTLDEF]*.H;
$ DIR SYS$LIBRARY:*.H;
```

The first command lists the text module form of the header files for the OpenVMS system interfaces. The second lists the text module form of the header files for the VSI C language interface. The third lists *.H header files for the VSI C language interfaces. The fourth lists *.H header files for layered products and other applications.

Note

The SYS$COMMON:[DECC$LIB.REFERENCE.DECC$RTLDEF] directory is only a reference area for your viewing. The compiler still looks in the *.TLB files for `#include` file searches.

However, duplicate files (such as `<stdio.h>`) found in SYS$LIBRARY probably support the VAX C Version 3.2 environment and should not be used with VSI C.

Function definitions themselves are not included in the header files, but are contained in the VSI C Run-Time Library (C RTL) shipped with the OpenVMS operating system. Before using the C RTL, you must be familiar with the following topics:

- The linking process
- The macro substitution process
- The difference between function definitions and function calls
- The format of valid file specifications
- The OpenVMS-specific methods of input and output (I/O)

- The VSI C for OpenVMS extensions and nonstandard features

A knowledge of all these topics is necessary to effectively use the C RTL. This chapter shows the connections between these topics and the C RTL. Read this chapter before any of the other chapters in this manual.

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

Chapters 2 through 11 describe the various tasks supported by the C RTL. The Reference Section alphabetically lists and describes all the functions and macros available to perform these tasks.

1.1. Using the VSI C Run-Time Library (C RTL)

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

First, if you plan to use C RTL functions in your C programs, make sure that a function named `main` or a function that uses the `main_program` option exists in your program. For more information, see the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] or the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/].

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

You can use the C RTL as a shareable image or you can use the C RTL object libraries.

When you use the C RTL as a shareable image, the code for the RTL resides in an image file in SYS$SHARE and is shared by all VSI C programs. After execution, control returns to your program. This process has a number of advantages:

- You reduce the size of a program's executable image.
- The program's image takes up less disk space.
- The program swaps in and out of memory faster due to decreased size.
- With VSI C and VSI C++, you no longer need to define an options file when linking your program against the shareable image. Linking against the RTL shareable image is now much simpler than it was with VAX C. In fact, it is the default method of linking to the C RTL.

When linking to the C RTL, you do not need to define any LNK$LIBRARY logicals. In fact, you should deassign LNK$LIBRARY because linking with the shareable image is more convenient than linking with the C RTL object libraries.

See your OpenVMS, VSI C, or VSI C++ release notes for any supplemental information about linking with the C RTL.

1.2. RTL Linking Options

The following sections describe several ways of linking VSI C and VSI C++ programs with the C RTL on OpenVMS Alpha, Integrity server, and x86-64 systems.

1.2.1. Linking with the Shareable Image

Most linking needs should be satisfied by using the C RTL shareable image DECC$SHR.EXE in the ALPHA$LIBRARY (Alpha systems), IA64$LIBRARY (Integrity server systems), or X86$LIBRARY (x86-64 systems) directory.

The shareable images VAXCRTL.EXE and VAXCRTLG.EXE do not exist on OpenVMS Alpha, Integrity server, and x86-64 systems. The only C RTL shareable image is ALPHA$LIBRARY:DECC$SHR.EXE (Alpha systems), IA64$LIBRARY:DECC$SHR.EXE (Integrity server systems), and X86$LIBRARY:DECC$SHR.EXE (x86-64 systems), which the linker automatically finds through IMAGELIB.OLB.

The fact that VAXCRTL*.EXE does not exist on Alpha and Integrity server systems has the following ramifications:

- You must change any existing VAX C link procedures to eliminate any references to the VAXCRTL*.EXE images. An explicit reference to DECC$SHR.EXE is unnecessary because IMAGELIB.OLB is searched automatically by the linker (see the *VSI OpenVMS Linker Utility Manual*).

- Because DECC$SHR.EXE exports only prefixed universal symbols (ones that begin with DECC$), to successfully link against it make sure you cause prefixing to occur for all C RTL entry points that you use.

  If you use only the C RTL functions defined in the ANSI C Standard, all entry points will be prefixed.

  If you use C RTL functions not defined in the ANSI C Standard, you must compile in one of two ways to ensure prefixing:

  - Compile with the /PREFIX_LIBRARY_ENTRIES= ALL_ENTRIES qualifier.

  - Compile with the /STANDARD=VAXC or /STANDARD=COMMON qualifier; you get /PREFIX_LIBRARY_ENTRIES= ALL_ENTRIES as the default.

To link against the shareable image, use the LINK command. For example:

```
$ LINK PROG1
```

The linker automatically searches IMAGELIB.OLB to find DECC$SHR.EXE, and resolves all C RTL references.

1.2.2. Linking with the Object Libraries (Alpha only)

The C RTL object libraries on OpenVMS Alpha systems are used solely for linking programs compiled without /PREFIX=ALL. Please note that these object libraries do not exist on OpenVMS Integrity server systems.

On OpenVMS Alpha systems, the C RTL provides the following object libraries in the ALPHA$LIBRARY directory:

- VAXCCURSE.OLB

- VAXCRTLD.OLB

- VAXCRTLT.OLB
- VAXCRTL.OLB
- VAXCRTLX.OLB
- VAXCRTLDX.OLB
- VAXCRTLTX.OLB

The object library VAXCCURSE.OLB, which provides access to the Curses functions, contains unprefixed entry points that vector to the appropriate prefixed entry points.

The object libraries VAXCRTL.OLB, VAXCRTLD.OLB, VAXCRTLT.OLB, VAXCRTLX.OLB, VAXCRTLDX.OLB, and VAXCRTLTX.OLB also contain unprefixed entry points that vector to the appropriate prefixed entry points, depending on the floating-point type specified by the object library used:

- VAXCRTL.OLB contains all C RTL routine name entry points as well as VAX G-floating double-precision, floating-point entry points.
- VAXCRTLD.OLB contains a limited support of VAX D-floating double-precision, floating-point entry points.
- VAXCRTLT.OLB contains IEEE T-floating double-precision, floating-point entry points.
- VAXCRTLX.OLB contains G_floating support and support for the /L_DOUBLE_SIZE=128 compiler qualifier.
- VAXCRTLDX.OLB contains D_floating support and support for the /L_DOUBLE_SIZE=128 compiler qualifier.
- VAXCRTLTX.OLB contains IEEE T_floating support and support for the /L_DOUBLE_SIZE=128 compiler qualifier.

/L_DOUBLE_SIZE=128 is the default.

On the LINK command, specify only one of the VAXCRTL*.OLB libraries and, if needed, the VAXCCURSE.OLB library.

In the default mode of the compiler (/STANDARD=RELAXED_ANSI89) and also in strict ANSI C mode, all calls to ANSI C standard library routines are automatically prefixed with DECC$. With the /[NO]PREFIX_LIBRARY_ENTRIES qualifier, you can change this to prefix all C RTL names with DECC$, or to not prefix any C RTL names. Other options are also available for this qualifier. See the /[NO]PREFIX_LIBRARY_ENTRIES qualifier in this chapter for more information.

When linking with /NOSYSSHR, if calls to the C RTL routines are prefixed with DECC$, then the modules in STARLET.OLB are the only ones you need to link against. Since STARLET.OLB is automatically searched by the linker (unless the link qualifier /NOSYSLIB is used), all prefixed RTL external names are automatically resolved.

If any calls to the C RTL routines are not prefixed, then you need to explicitly link against VAXCRTL.OLB, VAXCRTLD.OLB, VAXCRTLT.OLB (or VAXCRTLX.OLB, VAXCRTLDX.OLB, or VAXCRTLDX.OLB), or VAXCCURSE.OLB, depending on which floating-point types you need, or if you want Curses functions. If you are linking with /NOSYSSHR, prefixed C RTL entry points are resolved in STARLET.OLB. If you are linking with /SYSSHR (the default), prefixed C RTL entry points are resolved in DECC$SHR.EXE.

1.2.3. Examples

The following examples show several different ways you might want to link with the C RTL. See Figure 1.1 for a graphical summary of these examples.

1. Most of the time, you just want to link against the shareable image:

   ```
   $ CC/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES PROG1
   $ LINK PROG1
   ```

   The linker automatically searches IMAGELIB.OLB to find DECC$SHR.EXE.

2. If you want to use just object libraries (to write privileged code or for ease of distribution, for example), use the /NOSYSSHR qualifier of the LINK command:

   ```
   $ CC/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES PROG1
   $ LINK/NOSYSSHR PROG1
   ```

   Prefixed RTL symbol references in the user program are resolved in the C RTL object library contained in STARLET.OLB.

   Notes

   - When linking VSI C programs against the C RTL object libraries using the /NOSYSSHR qualifier, applications that previously linked without undefined globals may result in undefined globals for the CMA$TIS symbols. To resolve these undefined globals, add the following line to your link options file:

     ```
     SYS$LIBRARY:STARLET.OLB/LIBRARY/INCLUDE=CMA$TIS
     ```

   - If a program linked with the /NOSYSSHR qualifier makes a call to a routine that resides in a dynamically activated image, and the routine returns a value indicating an unsuccessful status, `errno` is set to ENOSYS, and `vaxc$errno` is set to C$_NOSYSSHR. The error message corresponding to C$_NOSYSSHR is "Linking /NOSYSSHR disables dynamic image activation." An example of this situation is a program linked with /NOSYSSHR that makes a call to a socket routine.

   ---

3. (Alpha only). On OpenVMS Alpha systems, when compiling with prefixing disabled, in order to use object libraries that provide alternate implementations of C RTL functions, you need to use the VAXC*.OLB object libraries. In this case, compile and link as follows:

   ```
   $ CC/NOPREFIX_LIBRARY_ENTRIES PROG1
   $ LINK PROG1, MYLIB/LIBRARY, ALPHA$LIBRARY:VAXCRTLX.OLB/LIBRARY
   ```

   Unprefixed C RTL symbol references in the user program are resolved in MYLIB and in VAXCRTL.OLB.

   Prefixed C RTL symbol references in VAXCRTLX.OLB are resolved in DECC$SHR.EXE through IMAGELIB.OLB.

   In this same example, to get IEEE T-floating double-precision floating-point support, you might use the following compile and link commands:

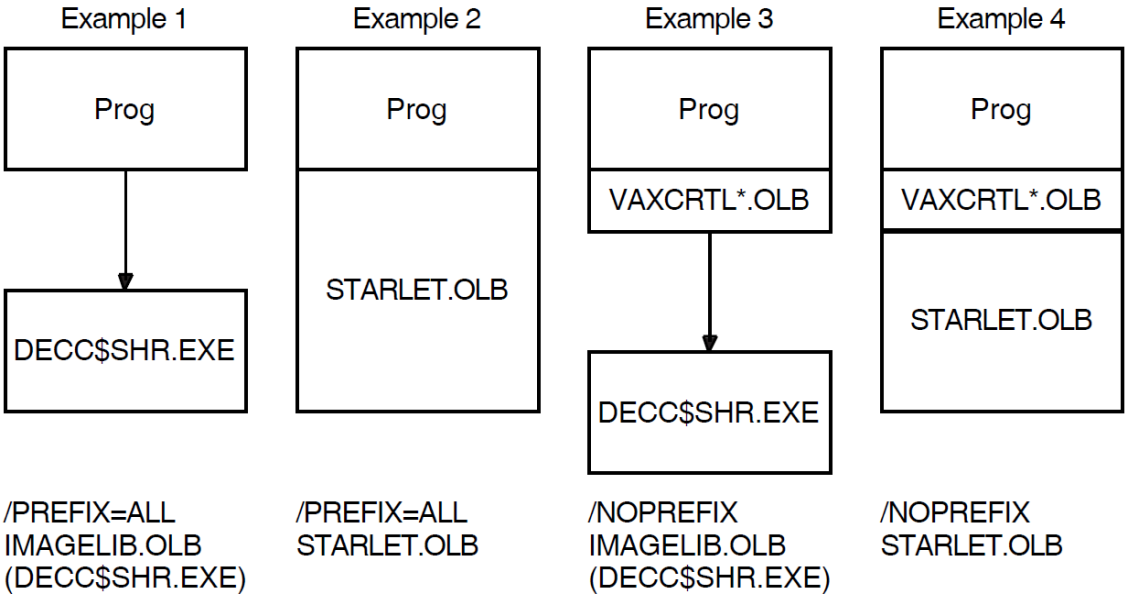
   ```
   $ CC/NOPREFIX_LIBRARY_ENTRIES/FLOAT=IEEE_FLOAT PROG1
   $ LINK PROG1, MYLIB/LIBRARY, ALPHA$LIBRARY:VAXCRTLTX.OLB/LIBRARY
   ```

4. (Alpha only). Combining examples 2 and 3, you might want to use just the object libraries (for writing privileged code or for ease of distribution) and use an object library that provides C RTL functions. In this case, compile and link as follows:

```
$ CC/NOPREFIX_LIBRARY_ENTRIES PROG1
$ LINK/NOSYSSHR PROG1, MYLIB/LIBRARY, ALPHA$LIBRARY:VAXCRTLX.OLB/LIBRARY
```

   Prefixed C RTL symbol references in VAXCRTL.OLB are resolved in STARLET.OLB.

Figure 1.1. Linking with the C RTL on OpenVMS Alpha and Integrity server Systems



1.2.4. DECC$SHRP.EXE Image

OpenVMS installs a shareable image DECC$SHRP.EXE to implement C RTL functions requiring protected mode. This shareable image is invoked from either the DECC$SHR.EXE or DECC$SHR_EV56.EXE shareable image.

1.3. C RTL Function Prototypes and Syntax

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

1.3.1. Function Prototypes

In all chapters, the syntax describing each function follows the standard convention for defining a function. This syntax is called a *function prototype* (or just *prototype*). The prototype is a compact representation of the order of a function's arguments (if any), the types of the arguments, and the type of the value returned by a function. We recommend the use of prototypes.

If the return value of the function cannot be easily represented by a C data-type keyword, look for a description of the return values in the explanatory text. The prototype descriptions provide insight into the functionality of the function. These descriptions may not describe how to call the function in your source code.

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

```
#include <stdio.h>
int feof(FILE *file_ptr);
```

This syntax shows the following information:

- The `feof` prototype resides in the `<stdio.h>` header file. To use `feof`, you must include this header file. (Declaring C RTL functions yourself is not recommended.)
- The `feof` function returns a value of data type `int`.
- There is one argument, `file_ptr`, that is of type "pointer to FILE". FILE is defined in the `<stdio.h>` header file.

To use `feof` in a program, include `<stdio.h>` anywhere before the function call to `feof`, as in the following example:

```
#include <stdio.h>                 /* Include Standard I/O     */

main()
{

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

1.3.2. Syntax Conventions for Function Prototypes

Since some library functions take a varying number of parameters, syntax descriptions for function prototypes adhere to the following conventions:

- Ellipses (…) are used to indicate a varying number of parameters.
- In cases where the type of a parameter may vary, its type is not shown in the syntax.

Consider the `printf` syntax description:

```
#include <stdio.h>
int printf(const char *format_specification, ...);
```

The syntax description for `printf` shows that you can specify one or more optional parameters. The remaining information about `printf` parameters is in the description of the function.

1.3.3. UNIX Style File Specifications

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

The following file specification conversion functions are included in the C RTL to assist in porting C programs from UNIX systems to OpenVMS systems:

- `decc$match_wild`
- `decc$translate_vms`
- `decc$fix_time`
- `decc$to_vms`
- `decc$from_vms`

The advantage of including these file specification conversion functions in the C RTL is that you do not have to rewrite C programs containing UNIX system file specifications. VSI C can translate most valid UNIX system file specifications to OpenVMS file specifications.

Please note the differences between the UNIX system and OpenVMS file specifications, as well as the method used by the RTL to access files. For example, the RTL accepts a valid OpenVMS specification and most valid UNIX file specifications, but the RTL cannot accept a combination of both. Table 1.1 shows the differences between UNIX system and OpenVMS system file specification delimiters.

Table 1.1. UNIX and OpenVMS File Specification Delimiters

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

For example, Table 1.2 shows the formats of two valid specifications and one invalid specification.

Table 1.2. Valid and Invalid UNIX and OpenVMS File Specifications

System	File Specification	Valid/Invalid
OpenVMS	BEATLE::DBA0:[MCCARTNEY]SONGS.LIS	Valid
UNIX	beatle!/usr1/mccartney/songs.lis	Valid
—	BEATLE::DBA0:[MCCARTNEY.C]/songs.lis	Invalid

When VSI C translates file specifications, it looks for both OpenVMS and UNIX system file specifications. Consequently, there may be differences between how VSI C translates UNIX system file specifications and how UNIX systems translate the same UNIX file specification.

For example, if the two methods of file specification are combined, as in Table 1.2, C RTL can interpret [MCCARTNEY.C]/songs.lis as either [MCCARTNEY]songs.lis or [C]songs.lis. Therefore, when VSI C encounters a mixed file specification, an error occurs.

UNIX systems use the same delimiter for the device name, the directory names, and the filename. Due to the ambiguity of UNIX file specifications, VSI C may not translate a valid UNIX system file specification according to your expectations.

For instance, the OpenVMS system equivalent of `/bin/today` can be either [BIN]TODAY or [BIN.TODAY]. VSI C can make the correct interpretation only from the files present. If a file specification conforms to UNIX system filename syntax for a single file or directory, it is converted to the equivalent OpenVMS filename if one of the following conditions is true:

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

Note

Beginning with OpenVMS Version 7.3, you can instruct the C RTL to interpret the leading part of a UNIX style file specification as either a subdirectory name or a device name.

As with previous releases, the default translation of `foo/bar` (UNIX style name) is FOO:BAR (OpenVMS style device name).

To request translation of `foo/bar` (UNIX style name) to [.FOO]BAR (OpenVMS style subdirectory name), define the logical name DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION to ENABLE. DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION is checked only once per image activation, not on a file-by-file basis. Defining this logical affects not only the `decc$to_vms` function, but all C RTL functions that accept both UNIX style and OpenVMS style filenames as an argument.

In the UNIX system environment, you reference files with a numeric file descriptor. Some file descriptors reference Standard I/O devices; some descriptors reference actual files. If the file descriptor belongs to an unopened file, the C RTL opens the file. VSI C equates file descriptors with the following OpenVMS logical names:

File Descriptor	OpenVMS Logical	Meaning
0	SYS$INPUT	Standard input
1	SYS$OUTPUT	Standard output
2	SYS$ERROR	Standard error

1.3.4. Extended File Specifications

The ODS-5 volume structure provides enhanced support for mixed UNIX and OpenVMS style filenames. It supports long filenames, allows the use of a wider range of characters within filenames, and preserves case within filenames. With OpenVMS Alpha Version 7.3-1, the C RTL has greatly improved support of ODS-5 characters, with 250 of the 256 characters supported, as opposed to only 214 supported previously. Also, filenames without file types can now be accessed.

To enable the new support, you must define one or more C RTL feature logical names. These names include the following:

DECC$EFS_CHARSET
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION
DECC$FILENAME_UNIX_NO_VERSION
DECC$FILENAME_UNIX_REPORT
DECC$READDIR_DROPDOTNOTYPE
DECC$RENAME_NO_INHERIT

See Section 1.5 for more information on these and other feature logical names.

1.3.5. Symbolic Links and POSIX Pathnames

OpenVMS provides support for Open Group compliant symbolic links and POSIX pathname processing. See Chapter 13 for more information.

1.4. Feature-Test Macros for Header-File Control

Feature-test macros provide a means for writing portable programs. They ensure that the C RTL symbolic names used by a program do not clash with the symbolic names supplied by the implementation.

The C RTL header files are coded to support the use of a number of feature-test macros. When an application defines a feature-test macro, the C RTL header files supply the symbols and prototypes defined by that feature-test macro and nothing else. If a program does not define such a macro, the C RTL header files define symbols without restriction.

The feature-test macros supported by the C RTL fall into the following broad categories for controlling the visibility of symbols in header files according to the following:

- Standards
- Multiple-version support
- Compatibility

1.4.1. Standards Macros

The C RTL implements parts of the following standards:

- X/Open CAE Specification, System Interfaces and Headers, Issue 4, Version 2, also known as XPG4 V2.
- X/Open CAE Specification, System Interfaces and Headers, Issue 4, also known as XPG4.
- Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) – Amendment 2: Threads Extension [C Language], also known as POSIX 1003.1c-1995 or IEEE 1003.1c-1995.
- ISO/IEC 9945-2:1993 - Information Technology - Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities, also known as ISO POSIX-2.
- ISO/IEC 9945-1:1990 - Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) (C Language), also known as ISO POSIX-1.
- ANSI/ISO/IEC 9899:1999 - The C99 standard, published by ISO in December, 1999 and adopted as an ANSI standard in April, 2000.
- ISO/IEC 9899:1990-1994 - Programming Languages - C, Amendment 1: Integrity, also known as ISO C, Amendment 1.
- ISO/IEC 9899:1990 - Programming Languages - C, also known as ISO C. The normative part is the same as X3.159-1989, American National Standard for Information Systems - Programming Language C, also known as ANSI C.

1.4.2. Selecting a Standard

You can define a feature-test macro to select each standard. You can do this either with a `#define` preprocessor directive in your C source before the inclusion of any header file, or with the /DEFINE qualifier on the CC command line.

Table 1.3 lists and describes the C RTL feature-test macros that control standards support.

Table 1.3. Feature Test Macros — Standards

Macro Name	Standard Selected	Other Standards Implied	Description
_XOPEN_SOURCE_ EXTENDED	XPG4 V2	XPG4, ISO POSIX-2, ISO POSIX-1, ANSI C	Makes visible XPG4-extended features, including traditional UNIX based interfaces not previously adopted by X/Open.
_XOPEN_SOURCE	XPG4 (X/Open Issue 4)	ISO POSIX-2, ISO POSIX-1, ANSI C	Makes visible XPG4 standard symbols and causes _POSIX_C_SOURCE to be set to 2 if it is not already defined with a value greater than 2.[1] [2]
_XOPEN_SOURCE=500	X/Open Issue 5	ISO POSIX-2, ISO POSIX-1, ANSI C	Makes visible X/Open Issue 5 standard symbols and causes _POSIX_C_SOURCE to be set to 2 if it is not already defined with a value greater than 2.[1] [2]
_XOPEN_SOURCE=600	X/Open Issue 6	ISO POSIX-2, ISO POSIX-1, ANSI C	Makes visible X/Open Issue 6 standard symbols and causes _POSIX_C_SOURCE to be set to 2 if it is not already defined with a value greater than 2.[1] [2]
_POSIX_C_SOURCE==199506	IEEE 1003.1c-1995	ISO POSIX-2, ISO POSIX-1, ANSI C	Header files defined by ANSI C make visible those symbols required by IEEE 1003.1c-1995.
_POSIX_C_SOURCE==2	ISO POSIX-2	ISO POSIX-1, ANSI C	Header files defined by ANSI C make visible those symbols required by ISO POSIX-2 plus those required by ISO POSIX-1.
_POSIX_C_SOURCE==1	ISO POSIX-1	ANSI C	Header files defined by ANSI C make visible those symbols required by ISO POSIX-1.
__STDC_VERSION__==199409	ISO C amdt 1	ANSI C	Makes ISO C Amendment 1 symbols visible.
_ANSI_C_SOURCE	ANSI C	–	Makes ANSI C standard symbols visible.
__HIDE_FORBIDDEN_NAMES			When defined to the value 1, causes the C RTL headers that are named in the C standard to be configured such that they

Macro Name	Standard Selected	Other Standards Implied	Description
			define only those identifiers that are specified as being defined by those headers under the version of the C standard in effect for the compilation, unless additional features are explicitly requested by other configuration macros (_XOPEN_SOURCE, for example). The C and C++ compilers will predefine this macro when certain language standard conformance features are selected, but the user can override any such predefinition by specifying /UNDEFINE=_ _HIDE_FORBIDDEN_ NAMES on the command line (or using `#undef` before including any headers). Conversely, the user can explicitly define the macro before including any headers, regardless of the language standard selected for the compiler.

[1]Where the ISO C Amendment 1 includes symbols not specified by XPG4, defining __STDC_VERSION__ == 199409 and _XOPEN_SOURCE (or _XOPEN_SOURCE_EXTENDED) selects both ISO C and XPG4 APIs. Conflicts that arise when compiling with both XPG4 and ISO C Amendment 1 resolve in favor of ISO C Amendment 1.

[2]Where XPG4 extends the ISO C Amendment 1, defining _XOPEN_SOURCE or _XOPEN_SOURCE_EXTENDED selects ISO C APIs as well as the XPG4 extensions available in the header file. This mode of compilation makes XPG4 extensions visible.

Features not defined by one of the previously named standards are considered VSI C extensions and are selected by not defining any standards-related, feature-test macros.

If you do not explicitly define feature test macros to control header file definitions, you implicitly include all defined symbols as well as VSI C extensions.

1.4.3. Interactions with the /STANDARD Qualifier

The /STANDARD qualifier selects the dialect of the C language supported.

With the exception of /STANDARD=ANSI89 and /STANDARD=ISOC94, the selection of C dialect and the selection of C RTL APIs to use are independent choices. All other values for /STANDARD cause the entire set of APIs to be available, including extensions.

Specifying /STANDARD=ANSI89 restricts the default API set to the ANSI C set. In this case, to select a broader set of APIs, you must also specify the appropriate feature-test macro. To select the ANSI C dialect and all APIs, including extensions, undefine __HIDE_FORBIDDEN_NAMES before including any header file.

Compiling with /STANDARD=ISOC94 sets __STDC_VERSION__ to 199409. Conflicts that arise when compiling with both XPG4 and ISO C Amendment 1 resolve in favor of ISO C Amendment 1. XPG4 extensions to ISO C Amendment 1 are selected by defining _XOPEN_SOURCE.

The following examples help clarify these rules:

- The `fdopen` function is an ISO POSIX-1 extension to `<stdio.h>`. Therefore, `<stdio.h>` defines `fdopen` only if one or more of the following is true:
  - The program including it is not compiled in strict ANSI C mode (/STANDARD=ANSI89).
  - _POSIX_C_SOURCE is defined as 1 or greater.
  - _XOPEN_SOURCE is defined.
  - _XOPEN_SOURCE_EXTENDED is defined.
- The `popen` function is an ISO POSIX-2 extension to `<stdio.h>`. Therefore, `<stdio.h>` defines `popen` only if one or more of the following is true:
  - The program including it is not compiled in strict ANSI C mode (/STANDARD=ANSI89).
  - _POSIX_C_SOURCE is defined as 2 or greater.
  - _XOPEN_SOURCE is defined.
  - _XOPEN_SOURCE_EXTENDED is defined.
- The `getw` function is an X/Open extension to `<stdio.h>`. Therefore, `<stdio.h>` defines `getw` only if one or more of the following is true:
  - The program is not compiled in strict ANSI C mode (/STANDARD=ANSI89).
  - _XOPEN_SOURCE is defined.
  - _XOPEN_SOURCE_EXTENDED is defined.
- The X/Open Extended symbolic constants _SC_PAGESIZE, _SC_PAGE_SIZE, _SC_ATEXIT_MAX, and _SC_IOV_MAX were added to `<unistd.h>` to support the `sysconf`function. However, these constants are not defined by _POSIX_C_SOURCE.

  The `<unistd.h>` header file defines these constants only if a program does not define _POSIX_C_SOURCE and does define _XOPEN_SOURCE_EXTENDED.

  If _POSIX_C_SOURCE is defined, these constants are not visible in `<unistd.h>`. Note that _POSIX_C_SOURCE is defined only for programs compiled in strict ANSI C mode.
- The `fgetname` function is a C RTL extension to `<stdio.h>`. Therefore, `<stdio.h>` defines `fgetname` only if the program is not compiled in strict ANSI C mode (/STANDARD=ANSI89).
- The macro _PTHREAD_KEYS_MAX is defined by POSIX 1003.1c-1995. This macro is made visible in `<limits.h>` when compiling for this standard with _POSIX_C_SOURCE == 199506 defined, or by default when compiling without any standards-defining, feature-test macros.
- The macro WCHAR_MAX defined in `<wchar.h>` is required by ISO C Amendment 1 but not by XPG4. Therefore:

- Compiling for ISO C Amendment 1 makes this symbol visible, but compiling for XPG4 compliance does not.
- Compiling for both ISO C Amendment 1 and XPG4 makes this symbol visible.

Similarly, the functions `wcsftime` and `wcstok` in `<wchar.h>` are defined slightly differently by the ISO C Amendment 1 and XPG4:

- Compiling for ISO C Amendment 1 makes the ISO C Amendment 1 prototypes visible.
- Compiling for XPG4 compliance makes the XPG4 prototypes visible.
- Compiling for both ISO C Amendment 1 and XPG4 selects the ISO C prototypes because conflicts resulting from this mode of compilation resolve in favor of ISO C.
- Compiling without any standard selecting feature test macros makes ISO C Amendment 1 features visible.

In this example, compiling with no standard-selecting feature-test macros makes WCHAR_MAX and the ISO C Amendment 1 prototypes for `wcsftime` and `wcstok` visible.

- The `wcswidth` and `wcwidth` functions areXPG4 extensions to ISO C Amendment 1. Their prototypes are in `<wchar.h>`.

  These symbols are visible if:

  - Compiling for XPG4 compliance by defining _XOPEN_SOURCE or _XOPEN_SOURCE_EXTENDED.
  - Compiling for DEC C Version 4.0 compatibility or on pre-OpenVMS Version 7.0 systems.
  - Compiling with no standard-selecting feature-test macros.
  - Compiling for both ISO C Amendment 1 and XPG4 compilance because these symbols are XPG4 extensions to ISO C Amendment 1.

  Compiling for strict ISO C Amendment 1 does not make them visible.

1.4.4. Multiple-Version-Support Macros

1.4.4.1. The __VMS_VER Macro

By default, the header files enable APIs in the C RTL provided by the version of the operating system on which the compilation occurs. This is accomplished by the predefined setting of the __VMS_VER macro, as described in the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/]. For example, compiling on OpenVMS Version 6.2 causes only C RTL APIs from Version 6.2 and earlier to be made available.

Another example of the use of the __VMS_VER macro is support for the 64-bit versions of C RTL functions available with OpenVMS Alpha Version 7.0 and higher. In all header files, functions that provide 64-bit support are conditionalized so that they are visible only if __VMS_VER indicates a version of OpenVMS that is greater than or equal to 7.0.

To target an older version of the operating system, do the following:

1. Define a logical DECC$SHR to point to the old version of DECC$SHR. The compiler uses a table from DECC$SHR to perform routine name prefixing.

2. Override the value of __VMS_VER by defining the macro __VMS_VER_OVERRIDE on the command line and setting its value to a number that represents the version of OpenVMS that you are compiling for. For example, to compile for OpenVMS Version 7.0, add the following switch to your compile command:

   ```
   /DEFINE=(__VMS_VER_0VERRIDE=70000000)
   ```

   This will set the value of __VMS_VER to 70000000.

   Defining __VMS_VER_OVERRIDE without a value sets __VMS_VER to the maximum value.

Targeting a newer version of the operating system might not always be possible. For some versions, you can expect that the new DECC$SHR.EXE will require new features of the operating system that are not present. For such versions, the defining of the logical DECC$SHR in Step 1 would cause the compilation to fail.

1.4.4.2. The __CRTL_VER Macro

The __CRTL_VER macro allows you to compile applications on an OpenVMS system with one version of the C RTL or C RTL ECO installed and execute them on OpenVMS systems with a different version of the C RTL or C RTL ECO.

The __CRTL_VER macro is defined by the compiler and is set to a value equal to the version of the C RTL on the current system. The current value of __CRTL_VER is set to 80500000.

If you compile an application that is intended to be executed on a system with a lower version of the C RTL, override the value of _CRTL_VER macro to enable functions and features that are only available on the system with the lower version of the C RTL.

You can override the value of __CRTL_VER by defining __CRTL_VER_OVERRIDE on the compile command line. For example, to set the value of __CRTL_VER to 80400000, use the following compiler switch:

```
/DEFINE=(__CRTL_VER_OVERRIDE=80400000)
```

Defining __CRTL_VER_OVERRIDE without a value sets __CRTL_VER to the maximum value.

1.4.5. Compatibility Modes

The following predefined macros are used to select header-file compatibility with previous versions of DEC C or the OpenVMS operating system:

- _DECC_V4_SOURCE
- _VMS_V6_SOURCE

There are two types of incompatibilities that can be controlled in the header files:

- To conform to standards, some changes are source-code incompatible but binary compatible. To select DEC C Version 4.0 source compatibility, use the _DECC_V4_SOURCE macro.
- Other changes to conform to standards introduce a binary or run-time incompatibility.

In general, programs that recompile get new behaviors. In these cases, use the _VMS_V6_SOURCE feature test macro to retain previous behaviors.

However, for the `exit`, `kill`, and `wait` functions, the OpenVMS Version 7.0 changes to make these routines ISO POSIX-1 compliant were considered too incompatible to become the default. Therefore, in these cases the default behavior is the same as on pre-OpenVMS Version 7.0 systems. To access the versions of these routines that comply with ISO POSIX-1, use the _POSIX_EXIT feature test macro.

The following examples help clarify the use of these macros:

- To conform to the ISO POSIX-1 standard, `typedefs` for the following have been added to `<types.h>`:

```
dev_t          off_t
gid_t          pid_t
ino_t          size_t
mode_t         ssize_t
nlink_t        uid_t
```

  Previous development environments using a version of DEC C earlier than Version 5.2 may have compensated for the lack of these `typedef`s in `<types.h>` by adding them to another module. If this is the case on your system, then compiling with the `<types.h>` provided with DEC C Version 5.2 might cause compilation errors.

  To maintain your current environment and include the DEC C Version 5.2 `<types.h>`, compile with _DECC_V4_SOURCE defined. This will omit incompatible references from the DEC C Version 5.2 headers. In `<types.h>`, for example, the previously listed `typedefs` will not be visible.

- As of OpenVMS Version 7.0, the C RTL `getuid` and `geteuid` functions are defined to return an OpenVMS UIC (user identification code) that contains both the group and member portions of the UIC. In previous versions of the DEC C RTL, these functions returned only the member number from the UIC code.

  Note that the prototypes for `getuid` and `geteuid` in `<unistd.h>` (as required by the ISO POSIX-1 standard) and in `<unixlib.h>` (for C RTL compatibility) have not changed. By default, newly compiled programs that call `getuid` and `geteuid` get the new definitions. That is, these functions will return an OpenVMS UIC.

  To let programs retain the pre-OpenVMS Version 7.0 behavior of `getuid` and `geteuid`, compile with the _VMS_V6_SOURCE feature-test macro defined.

- As of OpenVMS Version 7.0, the C RTL `exit` function is defined with ISO POSIX-1 semantics. As a result, the input status argument to `exit` takes a number between 0 and 255. (Prior to this, `exit` could take an OpenVMS condition code in its status parameter.)

  By default, the behavior for `exit` on OpenVMS systems is the same as before: `exit` accepts an OpenVMS condition code. To enable the ISO POSIX-1 compatible `exit` function, compile with the _POSIX_EXIT feature-test macro defined.

1.4.6. Curses and Socket Compatibility Macros

The following feature-test macros are used to control the Curses and Socket subsets of the C RTL library:

- _BSD44_CURSES

  This macro selects the Curses package from the 4.4BSD Berkeley Software Distribution.

- _VMS_CURSES

  This macro selects a Curses package based on the VAX C compiler. This is the default Curses package.

- _SOCKADDR_LEN

  This macro is used to select 4.4BSD-compatible and XPG4 V2-compatible socket interfaces. These interfaces require support in your underlying TCP/IP software. Contact your TCP/IP vendor to inquire if the version of TCP/IP software you run supports 4.4BSD sockets.

Strict XPG4 V2 compliance requires the 4.4BSD-compatible socket interface. Therefore, if _XOPEN_SOURCE_EXTENDED is defined on OpenVMS Version 7.0 or higher, _SOCKADDR_LEN is defined to be 1.

The following examples help clarify the use of these macros:

- Symbolic constants like AE, AL, AS, AM, BC, which represent pointers to termcap fields used by the BSD Curses package, are only visible in `<curses.h>` if _BSD44_CURSES is defined.
- The `<socket.h>` header file defines a 4.4BSD `sockaddr` structure only if _SOCKADDR_LEN or _XOPEN_SOURCE_EXTENDED is defined. Otherwise, `<socket.h>` defines a pre-4.4BSD `sockaddr` structure. If _SOCKADDR_LEN is defined and _XOPEN_SOURCE_EXTENDED is not defined.

  The `<socket.h>` header file also defines an `osockaddr` structure, which is a 4.3BSD `sockaddr` structure to be used for compatibility purposes. Since XPG4 V2 does not define an `osockaddr` structure, it is not visible in _XOPEN_SOURCE_EXTENDED mode.

1.4.7. 2 GB File Size Macro

The C RTL provides support for compiling applications to use file sizes and offsets that are 2 GB and larger. This is accomplished by allowing file offsets of 64-bit integers.

The `fseeko` and `ftello` functions, which have the same behavior as `fseek` and `ftell`, accept or return values of type `off_t`, which allows for a 64-bit variant of `off_t` to be used.

C RTL functions `lseek`, `mmap`, `ftuncate`, `truncate`, `stat`, `fstat`, and `ftw` can also accommodate a 64-bit file offset.

The new 64-bit interfaces can be selected at compile time by defining the _LARGEFILE feature macro.

1.4.8. 32-Bit UID and GID Macro

The C RTL supports 32-bit User Identification (UID) and Group Identification (GID). When an application is compiled to use 32-bit UID/GID, the UID and GID are derived from the UIC as in previous versions of the operating system.

To compile an application for 16-bit UID/GID support on systems that by default use 32-bit UIDs/GIDs, define the _DECC_SHORT_GID_T macro to 1.

Not specifying _DECC_SHORT_GID_T provides long (32-bit) UID/GID.

Compiling on older OpenVMS systems where long UID/GID is not supported, or compiling for legacy compatibility (_DECC_V4_SOURCE for VSI C Version 4 or _VMS_V6_SOURCE for OpenVMS Version 6), forces use of short (16-bit) UID/GID.

1.4.9. Standard-Compliant stat Structure

The C RTL supports an X/Open standard-compliant definition of the `stat` structure and associated definitions. To use these new definitions, applications must compile with the _USE_STD_STAT feature-test macro defined. Use of _USE_STD_STAT specifies long (32-bit) GIDs.

When compiled with _USE_STD_STAT, the `stat` structure includes these changes:

- Type `ino_t` is defined as an `unsigned quadword int`. Without _USE_STD_STAT, it is an `unsigned short`.
- Type `dev_t` is defined as a 64-bit integer. Without _USE_STD_STAT, it is a 32-bit character pointer.
- Type `off_t` is defined as a 64-bit integer, as if the _LARGEFILE macro has been defined. Without _USE_STD_STAT, `off_t` is a 32-bit integer.
- Fields `st_dev` and `st_rdev` will have unique values per device. Without _USE_STD_STAT, uniqueness is not assured.
- Fields `st_blksize` and `st_blocks` are added. Without _USE_STD_STAT, these fields do not exist.

1.4.10. Using Legacy `_toupper` and `_tolower` Behavior

As of OpenVMS Version 8.3, to comply with the C99 ANSI standard and X/Open Specification, the `_tolower` and `_toupper` macros by default do not evaluate their parameter more than once. They simply call their respective `tolower` or `toupper` function. This avoids side effects (such as i++ or function calls) where the user can tell how many times an expression is evaluated.

To retain the older, optimized behavior of the `_tolower` and `_toupper` macros, compile with /DEFINE=_FAST_TOUPPER. Then, as in previous releases, these macros optimize the call to avoid the overhead of a runtime call. However, the macro's parameter is evaluated more than once to determine how to calculate the result, possibly creating unwanted side effects.

1.4.11. Using Faster, Inlined Put and Get Functions

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that sets the following I/O functions to use faster, in lined functions:

```
fgetc
fputc
putc
putchar
fgetc_unlocked
fputc_unlocked
putc_unlocked
putchar_unlocked
```

1.4.12. POSIX Style Exit

The VSI C and C++ Version 7.1 and higher compilers have a /MAIN=POSIX_EXIT qualifier that defines the _POSIX_EXIT macro and causes the main program to call `__posix_exit` instead of `exit` when returning from the main program.

This qualifier should be used with programs ported from UNIX that do not explicitly call `exit` and do not use OpenVMS specific exit codes.

For older compilers, the following sample code can be used to force the existing main module to have a different name so that a simple `main` program will call it but force the exit status to be through the `__posix_exit` call.

The replacement `main` function can be in a different module, so that /DEFINE="main=real_main" is all that is needed for modifying the build of the existing `main` function.

```
#define _POSIX_EXIT 1

#include <stdlib.h>

int real_main(int argc, char **argv);

/* Make sure POSIXized exit is used */
int main(int argc, char **argv)
{
int ret_status;

   ret_status = real_main(argc, argv);

exit (ret_status);
}
#define main real_main
```

Unless your C program is intentionally using OpenVMS status codes for exit values, it is strongly recommended that both the _POSIX_EXIT macro be defined and, if needed, the /MAIN=POSIX_EXIT or the alternative `main` replacement be used so that DCL, BASH, and the accounting file get usable exit values.

1.5. Enabling C RTL Features Using Feature Logical Names

The C RTL provides an extensive list of feature switches that can be set using DECC$ logical names. These switches affect the behavior of a C application at run time.

The feature switches introduce new behaviors and also preserve old behaviors that have been deprecated.

You enable most features by setting a logical name to ENABLE and disable a feature by setting the logical name to DISABLE:

```
$ DEFINE DECC$feature ENABLE

DEFINE DECC$feature DISABLE
```

Some feature logical names can be set to a numeric value. For example:

```
$ DEFINE DECC$PIPE_BUFFER_SIZE 32768
```

Notes

- Do not set C RTL feature logical names for the system. Set them only for the applications that need them, because other applications including OpenVMS components depend on the default behavior of these logical names.
- Older feature logicals from earlier releases of the C RTL were documented as supplying "any equivalence string" to enable a feature. While this was true at one time, we now strongly recommend that you use ENABLE for setting these feature logicals and DISABLE for disabling them. Failure to do so may produce unexpected results.

  The reason for this is twofold:

  - In previous versions of the C RTL, *any* equivalence string, even DISABLE, may have enabled a feature logical.
  - In subsequent and current versions of the C RTL, the following equivalence strings will *disable* a feature logical. Do not use them to *enable* a feature logical.

    DISABLE
    0 (zero)
    F
    FALSE
    N
    NO

    Any other string not on this list will enable a feature logical. The unintentionally misspelled string "DSABLE", for example, will enable a feature logical.

The C RTL also provides several functions to manage feature logicals within your applications:

```
decc$feature_get
decc$feature_get_value
decc$feature_get_index
decc$feature_get_name
decc$feature_set
decc$feature_set_value
decc$feature_show
decc$feature_show_all
```

See the Reference Section for more information on these functions.

Table 1.4 lists the C RTL feature logical names, grouped by the type of features they control.

Table 1.4. C RTL Feature Logical Names

Feature Logical Name	Default
Performance Optimizations	
DECC$ENABLE_GETENV_CACHE	DISABLE
DECC$LOCALE_CACHE_SIZE	0
DECC$TZ_CACHE_SIZE	2
Legacy Behaviors	
DECC$ALLOW_UNPRIVILEGED_NICE	DISABLE

Feature Logical Name	Default
DECC$NO_ROOTED_SEARCH_LISTS	DISABLE
DECC$PRINTF_USES_VAX_ROUND	DISABLE
DECC$THREAD_DATA_AST_SAFE	DISABLE
DECC$V62_RECORD_GENERATION	DISABLE
DECC$WRITE_SHORT_RECORDS	DISABLE
DECC$XPG4_STRPTIME	DISABLE
File Attributes	
DECC$DEFAULT_LRL	32767
DECC$DEFAULT_UDF_RECORD	DISABLE
DECC$FIXED_LENGTH_SEEK_TO_EOF	DISABLE
DECC$ACL_ACCESS_CHECK	DISABLE
DECC$PRN_PRE_BYTE	DISABLE
Mailboxes	
DECC$MAILBOX_CTX_STM	DISABLE
Changes for UNIX Conformance	
DECC$SELECT_IGNORES_INVALID_FD	DISABLE
DECC$STRTOL_ERANGE	DISABLE
DECC$VALIDATE_SIGNAL_IN_KILL	DISABLE
General UNIX Enhancements	
DECC$UNIX_LEVEL	DISABLE
DECC$ARGV_PARSE_STYLE	DISABLE
DECC$PIPE_BUFFER_SIZE	512
DECC$PIPE_BUFFER_QUOTA	512
DECC$STREAM_PIPE	DISABLE
DECC$POPEN_NO_CRLF_REC_ATTR	DISABLE
DECC$STDIO_CTX_EOL	DISABLE
DECC$USE_RAB64	DISABLE
DECC$GLOB_UNIX_STYLE	DISABLE
Enhancements for UNIX Style Filenames	
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION	DISABLE
DECC$EFS_CHARSET	DISABLE
DECC$ENABLE_TO_VMS_LOGNAME_CACHE	ENABLE
DECC$FILENAME_ENCODING_UTF8	DISABLE
DECC$FILENAME_UNIX_NO_VERSION	DISABLE
DECC$FILENAME_UNIX_REPORT	DISABLE
DECC$READDIR_DROPDOTNOTYPE	DISABLE
DECC$RENAME_NO_INHERIT	DISABLE
DECC$RENAME_ALLOW_DIR	DISABLE

Feature Logical Name	Default
Enhancements for UNIX Style File Attributes	
DECC$EFS_FILE_TIMESTAMPS	DISABLE
DECC$EXEC_FILEATTR_INHERITANCE	DISABLE
DECC$FILE_OWNER_UNIX	DISABLE
DECC$FILE_PERMISSION_UNIX	DISABLE
DECC$FILE_SHARING	DISABLE
UNIX Compliance Mode	
DECC$DETACHED_CHILD_PROCESS	DISABLE
DECC$FILENAME_UNIX_ONLY	DISABLE
DECC$POSIX_STYLE_UID	DISABLE
DECCUSE_JPI_CREATOR	DISABLE
New Behaviors for POSIX Conformance	
DECC$ALLOW_REMOVE_OPEN_FILES	DISABLE
DECC$POSIX_SEEK_STREAM_FILE	DISABLE
DECC$UMASK	RMS default
File-Name Handling	
DECC$POSIX_COMPLIANT_PATHNAMES	DISABLE
DECC$DISABLE_POSIX_ROOT	ENABLE
DECC$EFS_CASE_PRESERVE	DISABLE
DECC$EFS_CASE_SPECIAL	DISABLE
DECC$EFS_NO_DOTS_IN_DIRNAME	DISABLE
DECC$READDIR_KEEPDOTDIR	DISABLE
DECC$UNIX_PATH_BEFORE_LOGNAME	DISABLE

An alphabetic listing and description of the C RTL feature logical names follows. Unless otherwise stated, the feature logicals are enabled with ENABLE and disabled with DISABLE.

DECC$ACL_ACCESS_CHECK

The DECC$ACL_ACCESS_CHECK feature logical controls the behavior of the `access` function.

With DECC$ACL_ACCESS_CHECK enabled, the `access` function checks both UIC protection and OpenVMS Access Control Lists (ACLs).

With DECC$ACL_ACCESS_CHECK disabled, the `access` function checks only UIC protection.

DECC$ALLOW_REMOVE_OPEN_FILES

The DECC$ALLOW_REMOVE_OPEN_FILES feature logical controls the behavior of the `remove` function on open files. Ordinarily, the operation fails. However, POSIX conformance dictates that the operation succeed.

With DECC$ALLOW_REMOVE_OPEN_FILES enabled, this POSIX conformant behavior is achieved.

DECC$ALLOW_UNPRIVILEGED_NICE

With DECC$ALLOW_UNPRIVILEGED_NICE enabled, the `nice` function exhibits its legacy behavior of not checking the privilege of the calling process (that is, any user may lower the `nice` value to increase process priorities). Also, when the caller sets a priority above MAX_PRIORITY, the `nice` value is set to the base priority.

With DECC$ALLOW_UNPRIVILEGED_NICE disabled, the `nice` function conforms to the X/Open standard of checking the privilege of the calling process (only users with ALTPRI privilege can lower the `nice` value to increase process priorities), and when the caller sets a priority above MAX_PRIORITY, the `nice` value is set to MAX_PRIORITY.

DECC$ARGV_PARSE_STYLE

With DECC$ARGV_PARSE_STYLE enabled, case is preserved in command-line arguments when the process has been set up for extended DCL parsing using SETPROCESS/PARSE_STYLE=EXTENDED.

DECC$ARGV_PARSE_STYLE must be defined externally as a logical name or set in a function called using the LIB$INITIALIZE mechanism because it is evaluated before function `main` is called.

DECC$DEFAULT_LRL

DECC$DEFAULT_LRL specifies the default value for the RMS attribute for the longest record length. The default value 32767 is the largest record size supported by RMS.

Default: 32767

Maximum: 32767

DECC$DEFAULT_UDF_RECORD

With DECC$DEFAULT_UDF_RECORD enabled, file access mode defaults to RECORD instead of STREAM mode for all files except STREAMLF.

DECC$DETACHED_CHILD_PROCESS

With DECC$DETACHED_CHILD_PROCESS enabled, child processes created using `vfork` and `exec` are created as detached processes instead of subprocesses.

This feature has only limited support. In some cases the console cannot be shared between the parent process and the detached process, which can cause `exec` to fail.

DECC$DISABLE_POSIX_ROOT

With DECC$DISABLE_POSIX_ROOT enabled, support for the POSIX root directory defined by SYS$POSIX_ROOT is disabled.

With DECC$DISABLE_POSIX_ROOT disabled, the SYS$POSIX_ROOT logical name is interpreted as the equivalent of the file path "/". If a UNIX path starting with a slash (/) is given and the value after the leading slash cannot be translated as a logical name, SYS$POSIX_ROOT is used as the parent directory for the specified UNIX file path.

The C RTL supports a UNIX style root that behaves like a real directory. This allows such actions as:

```
% cd /
% mkdir /dirname
```

```
% tar -xvf tarfile.tar /dirname
% ls /
```

Previously, the C RTL did not recognize "/" as a directory name. The normal processing for a file path starting with "/" was to interpret the first element as a logical name or device name. If this failed, there was special processing for the name `/dev/null` and names starting with `/bin` and `/tmp`:

```
/dev/null       NLA0:
/bin            SYS$SYSTEM:
/tmp            SYS$SCRATCH:
```

These behaviors are retained for compatibility purposes. In addition, support has been added to the C RTL for the logical name SYS$POSIX_ROOT as an equivalent to "/".

To enable this feature for use by the C RTL, define SYS$POSIX_ROOT as a concealed logical name. For example:

```
$ DEFINE/TRANSLATION=(CONCEALED,TERMINAL) SYS$POSIX_ROOT "$1$DKA0:
[SYS0.abc.]"
```

To disable this feature:

```
$ DEFINE DECC$DISABLE_POSIX_ROOT DISABLE
```

Enabling SYS$POSIX_ROOT results in the following behavior:

- If the existing translation of a UNIX path starting with "/" fails and SYS$POSIX_ROOT is defined, the name is interpreted as if it starts with `/sys$posix_root`.
- When converting from an OpenVMS to a UNIX style filename, and the OpenVMS name starts with "SYS$POSIX_ROOT:", then the "SYS$POSIX_ROOT:" is removed. For example, SYS$POSIX_ROOT:[dirname] becomes `/dirname`. If the resulting name could be interpreted as a logical name or one of the special cases previously listed, the result is `/./dirname` instead of `/dirname`.

DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION

With DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION enabled, the conversion routine `decc$to_vms` will only treat the first element of a UNIX style name as a logical name if there is a leading slash (/).

DECC$EFS_CASE_PRESERVE

With DECC$EFS_CASE_PRESERVE enabled, case is preserved for filenames on ODS-5 disks.

With DECC$EFS_CASE_PRESERVE disabled, UNIX style filenames are always reported in lowercase.

However, note that enabling DECC$EFS_CASE_SPECIAL overrides the setting for DECC$EFS_CASE_PRESERVE.

DECC$EFS_CASE_SPECIAL

With DECC$EFS_CASE_SPECIAL enabled, case is preserved only for filenames containing lowercase. If an element of a filename contains all uppercase letters, it is reported in all lowercase in UNIX style.

When enabled, DECC$EFS_CASE_SPECIAL overrides the value of DECC$EFS_CASE_PRESERVE.

DECC$EFS_CHARSET

With DECC$EFS_CHARSET enabled, UNIX names can contain ODS-5 extended characters. Support includes multiple dots and all ASCII characters in the range 0 to 255, except the following:

```
<NUL>       /           *"          ?
```

Unless DECC$FILENAME_UNIX_ONLY is enabled, some characters can be interpreted as OpenVMS characters depending on context. They are:

```
:           ^[          ;<
```

DECC$EFS_CHARSET might be necessary for existing applications that make assumptions about filenames based on the presence of certain characters, because the following nonstandard and undocumented C RTL extensions do not work when EFS extended character-set support is enabled:

- $HOME is interpreted as the user's login directory

  With DECC$EFS_CHARSET enabled, $HOME is treated literally and may be in an OpenVMS or UNIX style filename.

- `~name` is interpreted as the login directory for user `name`

  With DECC$EFS_CHARSET enabled, `~name` is treated literally and can be in an OpenVMS or UNIX style filename.

- Wild card regular expressions in the form `[a-z]`

  With DECC$EFS_CHARSET enabled, square brackets are acceptable in OpenVMS and UNIX style filenames. For instance, in a function such as `open`, `abc[a-z]ef.txt` is interpreted as a UNIX style name equivalent to the OpenVMS style name `abc^[a-z^]ef.txt`, and `[a-z]bc` is interpreted asan OpenVMS style name equivalent to the UNIX style name `/sys$disk/a-z/bc`.

With DECC$EFS_CHARSET enabled, the following encoding for EFS extended characters is supported when converting from an OpenVMS style filename to a UNIX style filename:

- All ODS-2 compatible names
- All encoding for 8-bit characters, either as single byte or using two-digit hexadecimal form `^ab`. In a UNIX path these are always represented as a single byte.
- Encoding for DEL (^7F)
- The following characters when preceded by a caret:

  ```
  space ! , _ & ' ( ) + @ { } ; # [ ] % ^ = $ - ~ .
  ```

- The following characters when not preceded by a caret:

  ```
  $  -  ~  .
  ```

- The implementation supports the conversion from OpenVMS to UNIX needed for functions `readdir`, `ftw`, `getname`, `fgetname`, `getcwd`, and others.

Note

There are some special cases in C RTL filename processing. For example:

- Pathnames ending in `^.dir` are treated as directories, and when translated, these characters are truncated from the string.
- Pathnames beginning with `^/` treat the next token as a logical name or a directory from the root.

The following sample program shows these nuances:

```
#include <stdio.h>
#include <dirent.h>
#include <unixlib.h>
#include <string.h>
main()
{
  char adir[80];
  DIR *dir;
  struct dirent *dp;
  int decc_feature_efs_charset_index = 0;
  int decc_feature_efs_charset_default_val = 0;

  if (
      ( (decc_feature_efs_charset_index =
        decc$feature_get_index("DECC$EFS_CHARSET")) == -1 )
      ||
      ( (decc_feature_efs_charset_default_val =
        decc$feature_get_value(decc_feature_efs_charset_index, 0)) ==
 -1 )
      ||
      ( (decc$feature_set_value(decc_feature_efs_charset_index, 1, TRUE)
 == -1))
     )
    {
      printf("Error setting up DECC$EFS_CHARSET macro\n");
    }

  strcpy(adir, "SYS$SYSDEVICE:[SSHTEST.TEST.a^,test^.dir^;22]");
  printf("\n\nFor %s\n", adir);
mrb:  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  closedir(dir);

  strcpy(adir, "SYS$SYSDEVICE:[SSHTEST.TEST.a^,test^.dir]");
  printf("\n\nFor %s\n", adir);
  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
```

```
    }

  closedir(dir);

  strcpy(adir, "SYS$SYSDEVICE:[SSHTEST.TEST.a^\\test]");
  printf("\n\nFor %s\n", adir);
  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  strcpy(adir, "SYS$SYSDEVICE:[SSHTEST.TEST.copies]");
  printf("\n\nFor %s\n", adir);
  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  closedir(dir);

  strcpy(adir, "/SYS$SYSDEVICE/SSHTEST/TEST/copies");
  printf("\n\nFor %s\n", adir);
  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  closedir(dir);

}
```

DECC$EFS_FILE_TIMESTAMPS

With DECC$EFS_FILE_TIMESTAMPS enabled, `stat` and `fstat` report new ODS-5 access time (`st_atime`), attribute revision time (`st_ctime`) and modification time (`st_mtime`) for files on ODS-5 volumes that have the extended file times enabled using SET VOLUME/VOLUME=ACCESS_DATES.

If DECC$EFS_FILE_TIMESTAMPS is disabled, or the volume is not ODS-5, or the volume does not have support for these additional times enabled, `st_ctime` continues to be the file creation time and `st_atime` the same as the `st_mtime`.

The `utime` and `utimes` functions support these ODS-5 times in the same way as `stat`.

DECC$EFS_NO_DOTS_IN_DIRNAME

With support for extended characters in filenames for ODS-5, a name such as NAME.EXT can be interpreted as NAME.EXT.DIR. Determining if directory [.name^.ext] exists adds overhead to UNIX name translation when support for extended character support in UNIX filenames is enabled.

Enabling the DECC$EFS_NO_DOTS_IN_DIRNAME feature logical suppresses the interpretation of a filename containing dots as a directory name. With this logical enabled, NAME.EXT is assumed to be a filename; no check is made for directory [.name^.ext].

DECC$ENABLE_GETENV_CACHE

The C RTL supplements the list of environment variables in the `environ` table with all logical names and DCL symbols available to the process.

By default, whenever `getenv` is called for a name not in the `environ` table, an attempt is made to resolve this as a logical name and, if this fails, as a DCL symbol.

With DECC$ENABLE_GETENV_CACHE enabled, once a logical name or DCL name has been successfully translated, its value is stored in a cache. When the same name is requested in a future call to `getenv`, the value is returned from the cache instead of reevaluating the logical name or DCL symbol.

DECC$ENABLE_TO_VMS_LOGNAME_CACHE

Use the DECC$ENABLE_TO_VMS_LOGNAME_CACHE to improve the performance of UNIX name translation. The value is the life of each cache entry in seconds. The equivalence string ENABLE is evaluated as 1 second.

Define DECC$ENABLE_TO_VMS_LOGNAME_CACHE to 1 to enable the cache with a 1-second life for each entry.

Define DECC$ENABLE_TO_VMS_LOGNAME_CACHE to 2 to enable the cache with a 2-second life for each entry.

Define DECC$ENABLE_TO_VMS_LOGNAME_CACHE to -1 to enable the cache without a cache entry expiration.

DECC$EXEC_FILEATTR_INHERITANCE

The DECC$EXEC_FILEATTR_INHERITANCE feature logical affects child processes that are C programs.

For versions of OpenVMS before Version 7.3-2, DECC$EXEC_FILEATTR_INHERITANCE is either enabled or disabled:

- With DECC$EXEC_FILEATTR_INHERITANCE enabled, the current file pointer and the file open mode is passed to the child process in `exec` calls.
- With this logical name disabled, the child process does not inherit append mode or the file position.

For OpenVMS Version 7.3-2 and higher, DECC$EXEC_FILEATTR_INHERITANCE can be defined to 1 or 2, or be disabled:

- With DECC$EXEC_FILEATTR_INHERITANCE defined to 1, a child process inherits file positioning for all file access modes except append.
- With DECC$EXEC_FILEATTR_INHERITANCE defined to 2, a child process inherits file positioning for all file access modes including append.
- With DECC$EXEC_FILEATTR_INHERITANCE disabled, a child process does not inherit the file position for any access modes.

DECC$FILENAME_ENCODING_UTF8

C RTL routines that deal with filenames now support filenames in UTF-8 encoding when given in UNIX style.

For example, on an ODS-5 disk the OpenVMS DIRECTORY command supports a filename with the following characters:

```
disk:[mydir]^U65E5^U672C^U8A9E.txt
```

This filename contains three UCS-2 characters (call them xxx, yyy, and zzz for typographical purposes) meaning "day", "origin", and "language", respectively.

With UTF-8 support enabled, a C program can now read the filename from the VMS directory and use that filename as an UTF-8 encoded string.

For example, `opendir("/disk/mydir")` followed by a `readdir` will place the following into the *d_name* field of the supplied `dirent` structure:

```
"\xE6\x97\xA5\xE6\x9C\xAC\xE8\xAA\x9E.txt"
```

One of the following calls can then open this file:

```
open("/disk/mydir/\xE6\x97\xA5\xE6\x9C\xAC\xE8\xAA\x9E.txt",O_RDWR,0)
open("/disk/mydir/xxxyyyzzz.txt", O_RDWR,0)
```

The "\xE6\x97\xA5" above is the byte stream E697A5, which represents the xxx character in UTF-8 encoding. See the following example, where the actual characters comprising the filename are shown:

Figure 1.2. Unicode Example

```
$ DIR $1$DKA100:[ENCODE].TXT
Directory $1$DKA100:[ENCODE]

UTF8.TXT;1   ^U65E5^U672C^U8A9A.TXT;1

Total of 2 files.
$ MCR JSYS$CONTROL SET RMS/FILE=SDEC
$ DIR $1$DKA100:[ENCODE].TXT

Directory $1$DKA100:[ENCODE]

UTF8.TXT;1        日本語.TXT;1

Total of 2 files
$
```

This feature enhances the UNIX portability of international software that uses UTF-8 encoded filenames.

The DECC$FILENAME_ENCODING_UTF8 feature logical controls whether or not the C RTL allows and correctly interprets Unicode UTF-8 encoding for filenames given in UNIX style.

This logical is undefined by default, and the C RTL behavior is to accept filenames as ASCII and Latin-1 format.

This feature works only on ODS-5 disks. Therefore, to enable Unicode UTF-8 encoding, you must define both the DECC$FILENAME_ENCODING_UTF8 and DECC$EFS_CHARSET logicals to ENABLE.

DECC$FILENAME_UNIX_ONLY

With DECC$FILENAME_UNIX_ONLY enabled, filenames are never interpreted as OpenVMS style names. This prevents any interpretation of the following as OpenVMS special characters:

```
:    [    ^
```

DECC$FILENAME_UNIX_NO_VERSION

With DECC$FILENAME_UNIX_NO_VERSION enabled, OpenVMS version numbers are not supported in UNIX style filenames.

With DECC$FILENAME_UNIX_NO_VERSION disabled, in UNIX style names, version numbers are reported preceded by a period (.).

DECC$FILENAME_UNIX_REPORT

With DECC$FILENAME_UNIX_REPORT enabled, all filenames are reported in UNIX style unless the caller specifically selects OpenVMS style. This applies to `getpwnam`, `getpwuid`, `argv[0]`, `getname`, `fgetname`, and `tempnam`.

With DECC$FILENAME_UNIX_REPORT disabled, unless specified in the function call, filenames are reported in OpenVMS style.

DECC$FILE_OWNER_UNIX

When DECC$FILE_OWNER_UNIX is enabled, the owner for a new file or directory is always based on the effective UIC. When an earlier version of the file exists, the owner for the new file is inherited from this.

When disabled, the owner for a new file is set following VMS rules and may inherit the owner from the parent directory.

DECC$FILE_PERMISSION_UNIX

With DECC$FILE_PERMISSION_UNIX enabled, the file permissions for new files and directories are set according to the file creation mode and `umask`. This includes mode 0777. When an earlier version of the file exists, the file permissions for the new file are inherited from the earlier version. This mode sets DELETE permission for a new directory when WRITE permission is enabled.

With DECC$FILE_PERMISSION_UNIX disabled, modes 0 and 0777 indicate using RMS default protection or protection from the previous version of the file. Permissions for new directories also follow OpenVMS rules, including disabling DELETE permissions.

DECC$FILE_SHARING

With DECC$FILE_SHARING enabled, all files are opened with full sharing enabled (FAB$M_DEL | FAB$M_GET | FAB$M_PUT | FAB$M_UPD). This is set as a logical OR with any sharing mode specified by the caller.

DECC$FIXED_LENGTH_SEEK_TO_EOF

With DECC$FIXED_LENGTH_SEEK_TO_EOF enabled, `lseek`, `fseeko`, and `fseek` with the *direction* parameter set to SEEK_END will position relative to the last byte in the file for files with fixed-length records.

With DECC$FIXED_LENGTH_SEEK_TO_EOF disabled, `lseek`, `fseek`, and `fseeko` when called with SEEK_EOF on files with fixed-length records, will position relative to the end of the last record in the file.

DECC$GLOB_UNIX_STYLE

Enabling DECC$GLOB_UNIX_STYLE selects the UNIX mode of the `glob` function, which uses UNIX style filenames and wildcards instead of OpenVMS style filenames and wildcards.

DECC$LOCALE_CACHE_SIZE

DECC$LOCALE_CACHE_SIZE defines how much memory, in categories, to allocate for caching locale data. The default value is 0, which disables the locale cache.

Default: 0

Maximum: 2147483647

DECC$MAILBOX_CTX_STM

By default, an `open` on a local mailbox that is not a pipe treats mailbox records as having a record attribute of FAB$M_CR.

With DECC$MAILBOX_CTX_STM enabled, the record attribute FAB$M_CR is not set.

DECC$NO_ROOTED_SEARCH_LISTS

When the `decc$to_vms` function evaluates a UNIX style path string, if it determines the first element to be a logical name, then:

- For rooted logicals or devices, it appends ":[000000]" to the logical name.

  For example, if `log1` is a rooted logical ($DEFINE LOG1[DIR_NAME.]) then `/log1/filename.ext` translates to LOG1:[000000]FILENAME.EXT.

- For nonrooted logicals, it appends just a colon (:) to the logical name.

  For example, if `log2` is a nonrooted logical ($ DEFINE LOG2[DIR_NAME]), then `/log2/filename.ext` translates to LOG2:FILENAME.EXT.

- If the first element is a search-list logical, the translation proceeds by evaluating the first element in the search list, and translating the path as previously described.

The preceding three cases lead to predictable, expected results.

In the case where the first element is a search list that consists of a mixture of rooted and nonrooted logicals, translating paths as described previously can lead to different behavior from that of older versions of OpenVMS (before OpenVMS Version 7.3-1):

- Before OpenVMS Version 7.3-1, regardless of the contents of the logical, the `decc$to_vms` function appended only a colon (:). For search lists that consisted of a mixture of rooted and nonrooted logicals, this resulted in certain expected behaviors.

- For OpenVMS Version 7.3-1 and later, if the first element of the mixed search list is a rooted logical, then `decc$to_vms` appends ":[000000]" to the logical name, resulting in different behavior from that of OpenVMS releases prior to Version 7.3-1.

DECC$NO_ROOTED_SEARCH_LISTS controls how the `decc$to_vms` function resolves search-list logicals and provides a means to restore the OpenVMS behavior prior to Version 7.3-1.

With DECC$NO_ROOTED_SEARCH_LISTS enabled:

- If a logical is detected in a file specification, and it is a search list, then a colon (:) is appended when forming the OpenVMS file specification.
- If it is not a search list, the behavior is the same as with DECC$NO_ROOTED_SEARCH_LISTS disabled.

Enabling this feature logical provides the pre-Version 7.3-1 behavior for search list logicals.

With DECC$NO_ROOTED_SEARCH_LISTS disabled:

- If a logical is detected in a file specification, and it is a rooted logical (or a search list whose first element is a rooted logical), then ":[000000]" is appended when forming the OpenVMS file specification.
- If it is a nonrooted logical (or a search list whose first element is a nonrooted logical), then just a colon (:) is appended.

Disabling this feature logical provides the behavior for OpenVMS Version 7.3-1 and later.

DECC$PIPE_BUFFER_QUOTA

OpenVMS Version 7.3-2 adds an optional fourth argument of type `int` to the `pipe` function to specify the buffer quota of the pipe's mailbox. In previous OpenVMS versions, the buffer quota was equal to the buffer size.

DECC$PIPE_BUFFER_QUOTA lets you specify a buffer quota to use for the `pipe` function if the optional fourth argument of that function is omitted.

If the optional `pipe` fourth argument is omitted and DECC$PIPE_BUFFER_QUOTA is not defined, then the buffer quota defaults to the buffer size, as before.

Default: 512

Minimum: 512

Maximum: 2147483647

DECC$PIPE_BUFFER_SIZE

The system default buffer size of 512 bytes for pipe write operations can limit performance and generate extra line feeds when handling messages longer than 512 bytes.

DECC$PIPE_BUFFER_SIZE allows a larger buffer size to be used for pipe functions such as `pipe` and `popen`. A value of 512 to 65535 bytes can be specified.

If DECC$PIPE_BUFFER_SIZE is not specified, the default buffer size 512 is used.

Default: 512

Minimum: 512

Maximum: 65535

DECC$POPEN_NO_CRLF_REC_ATTR

With DECC$POPEN_NO_CRLF_REC_ATTR disabled, a pipe opened with the `popen` function has its record attributes set to CR/LF carriage control (fab$b_rat |= FAB$M_CR). This is the default behavior.

With DECC$POPEN_NO_CRLF_REC_ATTR enabled, CR/LF carriage control is prevented from being added to the pipe records. This is compatible with UNIX behavior, but be aware that enabling this feature might result in undesired behavior from other functions, such as `gets`, that rely on the carriage-return character.

DECC$POSIX_COMPLIANT_PATHNAMES

With DECC$POSIX_COMPLIANT_PATHNAMES enabled, an application is allowed to present POSIX-compliant pathnames to any C RTL function that accepts a pathname.

By default DECC$POSIX_COMPLIANT_PATHNAMES is disabled, and the usual C RTL behavior prevails. This disabled mode includes interpretation of pathnames as UNIX style specifications and uses rules that are different and unrelated to POSIX-compliant pathname processing.

To enable DECC$POSIX_COMPLIANT_PATHNAMES, set it to one of the following values:

1	All pathnames are designated as POSIX style.
2	Pathnames that end in "`:`" or contain any of the bracket characters " `[]<>`", and that can be successfully parsed by the SYS$FILESCAN service, are designated as OpenVMS style. Otherwise, they are designated as POSIX style.
3	The pathnames "`.`" and "`..`", or pathnames that contain "`/`"are designated as POSIX style. Otherwise, they are designated as OpenVMS style.
4	All pathnames are designated as OpenVMS style.

See Section 13.3.1 for more information

DECC$POSIX_SEEK_STREAM_FILE

With DECC$POSIX_SEEK_STREAM_FILE enabled, positioning beyond end-of-file on STREAM files does not write to the file until the next write. If the write is beyond the current end-of-file, this positions beyond the old end-of-file, and the start position for the write is filled with zeros.

With DECC$POSIX_SEEK_STREAM_FILE disabled, positioning beyond end-of-file will immediately write zeros to the file from the current end-of-file to the new position.

DECC$POSIX_STYLE_UID

With DECC$POSIX_STYLE_UID enabled, 32-bit UIDs and GIDs are interpreted as POSIX style identifiers.

With this logical name disabled, UIDs and GIDs are derived from the process UIC.

This feature is only available on OpenVMS systems providing POSIX style UID and GID support.

DECC$PRINTF_USES_VAX_ROUND

With DECC$PRINTF_USES_VAX_ROUND enabled, the F and E format specifiers of `printf` use VAX rounding rules for programs compiled with IEEE float.

DECC$PRN_PRE_BYTE

A change introduced by Hewlett Packard Enterprise (HPE) during OpenVMS V8.4 maintenance allowed systems that used the CIFS product (SAMBA) to display files in the appropriate format. However, that change affected files with PRINT carriage control (also known as FORTRAN carriage control). For some environments, the print codes that are removed when transferring files between systems cause incorrect printing behavior resulting in form feeds being lost.

When enabled, DECC$PRN_PRE_BYTE converts the print codes in files that have PRINT file carriage control to their ASCII control code equivalents. CIFS then sends them to the client.

Enabling DECC$PRN_PRE_BYTE, in addition to enabling the logical DECC$TERM_REC_CRLF, which is used by CIFS, correctly includes the print codes on transferred files.

To enable the DECC$PRN_PRE_BYTE logical feature, set it to ENABLE.

```
$ DEFINE/SYSTEM DECC$PRN_PRE_BYTE ENABLE
```

DECC$READDIR_DROPDOTNOTYPE

With DECC$READDIR_DROPDOTNOTYPE enabled, `readdir` when reporting files in UNIX style only reports the trailing period (.) for files with no file type when the filename contains a period.

With this logical name disabled, all files without a file type are reported with a trailing period.

DECC$READDIR_KEEPDOTDIR

The default behavior when reporting files in UNIX style from `readdir` is to report directories without a file type.

With DECC$READDIR_KEEPDOTDIR enabled, directories are reported in UNIX style with a file type of ".DIR".

DECC$RENAME_NO_INHERIT

DECC$RENAME_NO_INHERIT provides more UNIX compliant behavior in the `rename` function. With DECC$RENAME_NO_INHERIT enabled, the following behaviors are enforced:

- If the *old* argument points to the pathname of a file that is not a directory, the *new* argument will not point to the pathname of a directory.
- The new argument cannot point to a directory that exists.
- If the *old* argument points to the pathname of a directory, the *new* argument will not point to the pathname of a file that is not a directory.
- The new name for the file does not inherit anything from the old name. The new name must be specified completely. For example:

  Renaming "A.A" to "B" yields "B"

With this logical name disabled, you get the expected OpenVMS behavior. For example:

Renaming "A.A" to "B" yields "B.A"

DECC$RENAME_ALLOW_DIR

Enabling DECC$RENAME_ALLOW_DIR restores the prior OpenVMS behavior of the `rename` function by allowing conversion to a directory specification when the second argument

is an ambiguous file specification passed as a logical name. The ambiguity is whether the logical name is a UNIX or OpenVMS file specification. Consider the following example with DECC$RENAME_ALLOW_DIR enabled:

```
rename("file.ext", "logical_name") /* where logical_name = dev:
[dir.subdir] */
                                   /* and :[dir.subdir] exists.
    */
```

This results in:

```
dev:[dir.subdir]file.ext
```

This example renames a file from one directory into another directory, which is the same behavior as in legacy versions of OpenVMS (versions before 7.3-1). Also in this example, if `dev:[dir.subdir]` does not exist, `rename` returns an error.

Disabling DECC$RENAME_ALLOW_DIR provides a more UNIX compliant conversion of the `"logical_name"` argument of `rename`. Consider the following example with DECC$RENAME_ALLOW_DIR disabled:

```
rename("file.ext", "logical_name") /* where logical_name = dev:
[dir.subdir] */
```

This results in:

```
dev:[dir]subdir.ext
```

This example renames the file using the `subdir` part of the `"logical_name"` argument as the new filename because on UNIX systems, renaming a file to a directory is not allowed. So `rename` internally converts the `"logical_name"` to a filename, and `dev:[dir]subdir` is the most reasonable conversion it can perform.

This new feature switch has a side effect of causing rename to a directory to take precedence over rename to a file. Consider this example:

```
rename ( "file1.ext", "dir2" )      /* dir2 is not a logical */
```

With DECC$RENAME_ALLOW_DIR disabled, this example results in `dir2.ext`, regardless of whether or not subdirectory `[.dir2]` exists.

With DECC$RENAME_ALLOW_DIR enabled, this example results in `dir2.ext` only if subdirectory `[.dir2]` does not exist. If subdirectory `[.dir2]` does exist, the result is `[.dir2]file1.ext`.

Note

If DECC$RENAME_NO_INHERIT is enabled, UNIX compliant behavior is expected, so DECC$RENAME_ALLOW_DIR is ignored, and renaming a file to a directory is not allowed.

DECC$SELECT_IGNORES_INVALID_FD

With DECC$SELECT_IGNORES_INVALID_FD disabled, `select` and `poll` fail with `errno` set to EBADF when an invalid file descriptor is specified in one of the descriptor sets.

With DECC$SELECT_IGNORES_INVALID_FD enabled, `select` and `poll` ignore invalid file descriptors.

DECC$STDIO_CTX_EOL

With DECC$STDIO_CTX_EOL enabled, writing to `stdout` and `stderr` for stream access is deferred until a terminator is seen or the buffer is full.

With DECC$STDIO_CTX_EOL disabled, each `fwrite` generates a separate write, which for mailbox and record files generates a separate record.

DECC$STREAM_PIPE

With DECC$STREAM_PIPE enabled, the C RTL `pipe` function uses the more UNIX compatible stream I/O.

With DECC$STREAM_PIPE disabled, `pipe` uses the OpenVMS legacy record I/O. This is the default.

DECC$STRTOL_ERANGE

With DECC$STRTOL_ERANGE enabled, the `strtol` behavior for an ERANGE error is corrected to consume all remaining digits in the string.

With DECC$STRTOL_ERANGE disabled, the legacy behavior of leaving the pointer at the failing digit is preserved.

DECC$THREAD_DATA_AST_SAFE

The C RTL has a mode that allocates storage for thread-specific data allocated by threads at non-AST level separate for data allocated for ASTs. In this mode, each access to thread-specific data requires a call to LIB$AST_IN_PROG, which can add significant overhead when accessing thread-specific data in the C RTL.

The alternate mode protects thread-specific data only if another function has it locked. This protects data that is in use within the C RTL, but does not protect the caller from an AST changing the data pointed to.

This latter mode is now the C RTL default for the `strtok`, `ecvt`, and `fcvt` functions.

You can select the legacy AST safe mode by enabling DECC$THREAD_DATA_AST_SAFE.

DECC$TZ_CACHE_SIZE

DECC$TZ_CACHE_SIZE specifies the number of time zones that can be held in memory.

Default: 2

Maximum: 2147483647

DECC$UMASK

DECC$UMASK specifies the default value for the permission mask `umask`. By default, a parent C program sets the `umask` from the RMS default permissions for the process. A child process inherits the parent's value for `umask`.

To enter the value as an octal value, add the leading zero; otherwise, it is translated as a decimal value. For example:

```
$ DEFINE DECC$UMASK 026
```

Maximum: 0777

DECC$UNIX_LEVEL

With the DECC$UNIX_LEVEL logical name, you can manage multiple C RTL feature logical names at once. By setting a value for DECC$UNIX_LEVEL from 1 to 100, you determine the default value for groups of feature logical names. The value you set has a cumulative effect: the higher the value, the more groups that are affected. Setting a value of 20, for example, enables all the feature logicals associated with a DECC$UNIX_LEVEL of 20, 10, and 1.

The principal logical names affecting UNIX like behavior are grouped as follows:

1 General corrections
10 Enhancements
20 UNIX style filenames
30 UNIX style file attributes
90 Full UNIX behavior - No concessions to OpenVMS

Level 30 is appropriate for UNIX like programs such as BASH and GNV.

The DECC$UNIX_LEVEL values and associated groups of affected feature logical names are:

```
General Corrections          (DECC$UNIX_LEVEL 1)

   DECC$FIXED_LENGTH_SEEK_TO_EOF   1
   DECC$POSIX_SEEK_STREAM_FILE     1
   DECC$SELECT_IGNORES_INVALID_FD  1
   DECC$STRTOL_ERANGE              1
   DECC$VALIDATE_SIGNAL_IN_KILL    1

General Enhancements        (DECC$UNIX_LEVEL 10)

   DECC$ARGV_PARSE_STYLE           1
   DECC$EFS_CASE_PRESERVE          1
   DECC$STDIO_CTX_EOL              1
   DECC$PIPE_BUFFER_SIZE           4096
   DECC$USE_RAB64                  1

UNIX style filenames       (DECC$UNIX_LEVEL 20)

   DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION 1
   DECC$EFS_CHARSET                1
   DECC$FILENAME_UNIX_NO_VERSION   1
   DECC$FILENAME_UNIX_REPORT       1
   DECC$READDIR_DROPDOTNOTYPE      1
   DECC$RENAME_NO_INHERIT          1
   DECC$GLOB_UNIX_STYLE

UNIX like file attributes     (DECC$UNIX_LEVEL 30)

   DECC$EFS_FILE_TIMESTAMPS        1
   DECC$EXEC_FILEATTR_INHERITANCE  1
   DECC$FILE_OWNER_UNIX            1
   DECC$FILE_PERMISSION_UNIX       1
   DECC$FILE_SHARING               1

UNIX compliant behavior       (DECC$UNIX_LEVEL 90)

   DECC$FILENAME_UNIX_ONLY         1
   DECC$POSIX_STYLE_UID            1
```

```
DECC$USE_JPI$_CREATOR           1
DECC$DETACHED_CHILD_PROCESS     1
```

Notes

- Defining a logical name for an individual feature logical supersedes the default value established by DECC$UNIX_LEVEL for that feature.
- Future revisions of the C RTL may add new feature logicals to a given DECC$UNIX_LEVEL. For applications that specify that UNIX level, the effect is to enable those new feature logicals by default.

DECC$UNIX_PATH_BEFORE_LOGNAME

With DECC$UNIX_PATH_BEFORE_LOGNAME enabled, when translating a UNIX filename not starting with a leading slash (/), an attempt is made to match this to a file or directory in the current directory. If this is not found and the name is valid as a logical name in an OpenVMS filename, an attempt is made to translate the logical name and, if found, is used as part of the resulting filename.

Enabling DECC$UNIX_PATH_BEFORE_LOGNAME overrides the setting for DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION.

DECCUSE_JPI_CREATOR

When enabled, DECCUSE_JPI_CREATOR determines the parent process ID in `getppid` by calling $GETJPI using item JPI$_CREATOR instead of JPI$_OWNER.

This feature is only available on systems supporting POSIX style session identifiers.

DECC$USE_RAB64

With DECC$USE_RAB64 enabled, open functions allocate a RAB64 structure instead of the traditional RAB structure.

This provides latent support for file buffers in 64-bit memory.

DECC$VALIDATE_SIGNAL_IN_KILL

With DECC$VALIDATE_SIGNAL_IN_KILL enabled, a signal value that is in the range 0 to _SIG_MAX but is not supported by the C RTL generates an error with `errno` set to EINVAL, which makes the behavior the same as for `raise`.

With this logical name disabled, validation of signals is restricted to checking that the signal value is in the range 0 to _SIG_MAX. If `sys$sigprc` fails, `errno` is set based on `sys$sigprc` exit status.

DECC$V62_RECORD_GENERATION

OpenVMS Versions 6.2 and higher can output record files using different rules.

With DECC$V62_RECORD_GENERATION enabled, the output mechanism follows the rules used for OpenVMS Version 6.2.

DECC$WRITE_SHORT_RECORDS

The DECC$WRITE_SHORT_RECORDS feature logical supports a previous change to the `fwrite` function (to accommodate writing records with size less than the maximum record size), while retaining the legacy way of writing records to a fixed-length file as the default behavior:

With DECC$WRITE_SHORT_RECORDS enabled, short-sized records (records with size less than the maximum record size) written at EOF are padded with zeros to align records on record boundaries. This is the behavior seen in OpenVMS Version 7.3-1 and some ACRTL ECOs of that time period.

With DECC$WRITE_SHORT_RECORDS disabled, the legacy behavior of writing records with no padding is implemented. This is the recommended and default behavior.

DECC$XPG4_STRPTIME

XPG5 support for `strptime` introduces pivoting year support so that years in the range 0 to 68 are in the 21st century, and years in the range 69-99 are in the 20th century.

With DECC$XPG4_STRPTIME enabled, XPG5 support for the pivoting year is disabled and all years in the range 0 to 99 are in the current century.

1.6. 32-Bit UIDs/GIDs and POSIX Style Identifiers

Where supported in versions of the OpenVMS operating system, POSIX style identifiers refers to the User Identifier (UID), Group Identifier (GID), and Process Group. The scope includes real and effective identifiers.

The support for POSIX style identifiers in the C RTL requires 32-bit user and group ID support and also depends on features in the base version of OpenVMS. POSIX style IDs are supported by OpenVMS Version 7.3-2 and higher.

To use POSIX style identifiers on OpenVMS versions that support them requires applications to be compiled for 32-bit UID/GID. On OpenVMS versions where 32-bit UID/GID is the default, the user or application must still enable POSIX style IDs by defining the DECC$POSIX_STYLE_UID feature logical name:

```
$ DEFINE DECC$POSIX_STYLE_UID ENABLE
```

With POSIX style IDs enabled, at compile time you can selectively invoke the traditional (UIC-based) definition for an individual function by explicitly calling it by its `decc$`-prefixed entry point (as opposed to the `decc$__long_gid_`-prefixed entry point, which provides the POSIX style behavior).

To disable POSIX style IDs:

```
$ DEFINE DECC$POSIX_STYLE_UID DISABLE
```

OpenVMS Version 7.3-2 and higher supports POSIX style IDs as well as 32-bit UID/GIDs. When an application is compiled to use 32-bit UID/GIDs, the UID and GID are derived from the UIC as in previous versions of the operating system. In some cases, such as with the `getgroups` function, more information may be returned when the application supports 32-bit GIDs.

To compile an application for 16-bit UID/GID support on systems that by default use 32-bit UIDs/GIDs, define the _DECC_SHORT_GID_T macro to 1.

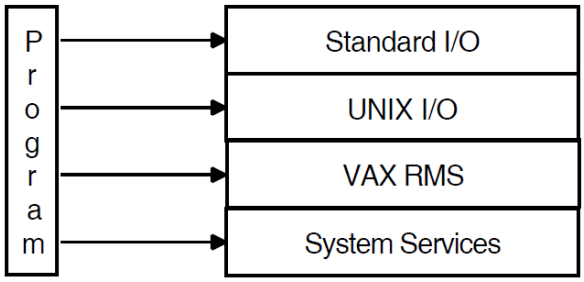
1.7. Input and Output on OpenVMS Systems

After you learn how to link with the C RTL and call VSI C functions and macros, you can use the C RTL for its primary purpose: input/output (I/O).

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

Figure 1.3 shows the I/O methods available with the C RTL. The OpenVMS system services communicate directly with the OpenVMS operating system, so they are closest to the operating system. The OpenVMS Record Management Services (RMS) functions use the system services, which manipulate the operating system. The VSI C Standard I/O and UNIX I/O functions and macros use the RMS functions. Since the C RTL Standard I/O and UNIX I/O functions and macros must go through several layers of function calls before the system is manipulated, they are furthest from the operating system.

Figure 1.3. I/O Interface from C Programs



The C programming language was developed on the UNIX operating system, and the Standard I/O functions were designed to provide a convenient method of I/O that would be powerful enough to be efficient for most applications, and also be portable so that the functions could be used on any system running C language compilers.

The C RTL adds functionality to this original specification. Since, as implemented in the C RTL, the Standard I/O functions recognize line terminators, the C RTL Standard I/O functions are particularly useful for text manipulation. The C RTL also implements some of the Standard I/O functions as preprocessor-defined macros.

In a similar manner, the UNIX I/O functions originally were designed to provide a more direct access to the UNIX operating systems. These functions were meant to use a numeric file descriptor to represent a file. A UNIX system represents all peripheral devices as files to provide a uniform method of access.

The C RTL adds functionality to the original specification. The UNIX I/O functions, as implemented in VSI C, are particularly useful for manipulating binary data. The C RTL also implements some of the UNIX I/O functions as preprocessor-defined macros.

The C RTL includes the Standard I/O functions that should exist on all C compilers, and also the UNIX I/O functions to maintain compatibility with as many other implementations of C as possible. However, both Standard I/O and UNIX I/O use RMS to access files. To understand how the Standard I/O and UNIX I/O functions manipulate RMS formatted files, learn the fundamentals of RMS. See Section 1.7.1 for more information about Standard I/O and UNIX I/O in relationship to RMS files. For an introduction to RMS, see the *VSI OpenVMS Guide to OpenVMS File Applications*.

Before deciding which method is appropriate for you, first ask this question: Are you concerned with UNIX compatibility or with developing code that will run solely under the OpenVMS operating system?

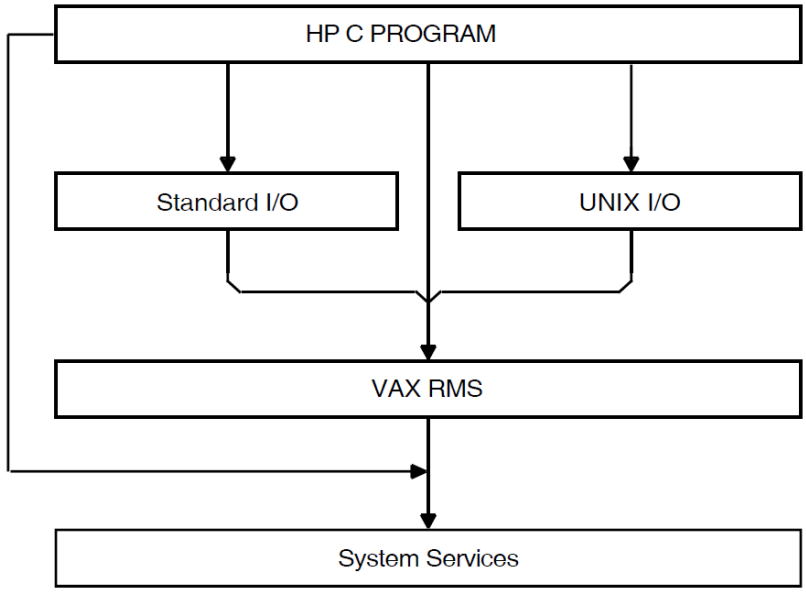
- If UNIX compatibility is important, you probably want to use the highest levels of I/O—Standard I/O and UNIX I/O—because that level is largely independent of the operating system. Also, the highest level is easier to learn quickly, an important consideration if you are a new programmer.

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

If you are writing system-level software, you may need to access the OpenVMS operating system directly through calls to system services. For example, you may need to access a user-written device driver directly through the Queue I/O Request System Service ($QIO). To do this, use the OpenVMS level of I/O; this level is recommended if you are an experienced OpenVMS programmer. For examples of programs that call OpenVMS system services, see the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/].

You may never use the RMS or the OpenVMS system services. The Standard I/O and UNIX I/O functions are efficient enough for a large number of applications. Figure 1.4 shows the dependency of the Standard I/O and the UNIX I/O functions on RMS, and the various methods of I/O available to you.

Figure 1.4. Mapping Standard I/O and UNIX I/O to RMS



1.7.1. RMS Record and File Formats

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

RMS supports the following file organizations:

- Sequential
- Relative
- Indexed

Sequential files have consecutive records with no empty records in between; relative files have fixed-length cells that may or may not contain a record; and indexed files have records that contain data, carriage-control information, and keys that permit various orders of access.

The C RTL functions can access only sequential files. If you wish to use the other file organizations, you must use the RMS functions. For more information about the RMS functions, see the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/].

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

RMS supports the following record formats:

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

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

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

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

By default, files inherit the RMS record format, maximum record size and record attributes, from the previous version of the file, if one exists; to an OpenVMS system programmer, the inherited attributes are known as FABB_RFM, FABW_MRS and FAB$B_RAT. If no previous versions exist, the newly created file defaults to stream format with line-feed record separator and implied carriage-return attributes. (This manual refers to this type of file as a *stream file*.) You can manipulate stream files using the Standard I/O and the UNIX I/O functions of the C RTL. When using these files and fixed-record files with no carriage control, there is no restriction on the ability to seek to any random byte of the file using the `fseek` or the `lseek` functions. However, if the file has one of the other RMS record formats, such as variable-length record format, then these functions, due to RMS restrictions, can seek only to record boundaries. Use the default VAX stream format unless you need to create or access files to be used with other VAX languages or utilities.

1.7.2. Access to RMS Files

RMS sequential files can be opened in record mode or stream mode. By default, STREAM_LF files are opened in stream mode; all other file types are opened in record mode. When opening a file, you can override these defaults by specifying the optional argument "ctx=rec" to force record mode, or "ctx=stm" to force stream mode. RMS relative and indexed files are always opened in record mode. The access mode determines the behavior of various I/O functions in the C RTL.

One of the file types defined by RMS is an RMS-11 stream format file, corresponding to a value of FAB$C_STM for the record format. The definition of this format is such that the RMS record operation SYS$GET removes leading null bytes from each record. Because this file type is processed in record mode by the C RTL, it is unsuitable as a file format for binary data unless it is explicitly opened with "ctx=stm", in which case the raw bytes of data from the file are returned.

Note

In OpenVMS Version 7.0 the default LRL value on stream files was changed from 0 to 32767. This change caused significant performance degradation on certain file operations such as sort.

This is no longer a problem. The C RTL now lets you define the logical DECC$DEFAULT_LRL to change the default record-length value on stream files.

The C RTL first looks for this logical. If it is found and it translates to a numeric value between 0 and 32767, that value is used for the default LRL.

To restore the behavior prior to OpenVMS Version 7.0, enter the following command:

```
$ DEFINE DECC$DEFAULT_LRL 0
```

1.7.2.1. Accessing RMS Files in Stream Mode

Stream access to RMS files is done with the block I/O facilities of RMS. Stream input is performed from RMS files by passing each byte of the on-disk representation of the file to your program. Stream output to RMS files is done by passing each byte from your program to the file. The C RTL performs no special processing on the data.

When opening a file in stream mode, the C RTL allocates a large internal buffer area. Data is read from the file using a single read into the buffer area and then passing the data to your program as needed. Data is written to the file when the internal buffer is full or when the `fflush` function is called.

1.7.2.2. Accessing RMS Record Files in Record Mode

Record access to record files is done with the record I/O facilities of RMS. The C RTL emulates a byte stream by translating carriage-control characters during the process of reading and writing records. Random access is allowed to all record files, but positioning (with `fseek` and `lseek`) must be on a record boundary for VFC files, variable record files, or files with non-null carriage control. Positioning a record file causes all buffered input to be discarded and buffered output to be written to the file.

Record input from RMS record files is emulated by the C RTL in two steps:

1. The C RTL reads a logical record from the file.

   If the record format is variable length with fixed control (RFM = VFC), and the record attributes are not print carriage control (RAT is not PRN), then the C RTL concatenates the fixed-control area to the beginning of the record.

2. The C RTL expands the record to simulate a stream of bytes by translating the record's carriage-control information (if any).

In RMS terms, the C RTL translates the record's carriage-control information using one of the following methods:

- If the record attribute is implied carriage control (RAT = CR), then the C RTL appends a new-line character to the record.

  This new-line character is considered an integral part of the record, which means for example, that it can be obtained by the `fgetc` function and is considered a line terminator by the `fgets` function. Since `fgets` reads the file up to the new-line character, for RAT=CR files this function cannot retrieve a string that crosses the record boundaries.

- If the record attributes are print carriage control (RAT = PRN), then the C RTL expands and concatenates the prefix and postfix carriage controls before and after the record.

  This translation is done according to rules specified by RMS, with one exception: if the prefix character is x01 and the postfix character is x8D, then nothing is attached to the beginning of the record and a single new-line character is attached to the end of it. This is done because this prefix/postfix combination is normally used to represent a line.

- If the record attributes are Fortran carriage control (RAT = FTN), then the C RTL removes the initial control byte and attaches the appropriate carriage-control characters before and after the data as defined by RMS, with the exception of the space and default carriage-control characters. In these cases, which are used to represent a line, the C RTL appends a single new-line character to the data.

  The mapping of Fortran carriage-control can be disabled by using "ctx=nocvt".

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

  If the input is coming from a nonterminal file, then the C RTL passes the record unchanged to your program with no additional prefix or postfix characters.

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

The C RTL performs record output to RMS record files in two steps.

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

- For all files, if the number of output bytes is greater than the internal buffer allocated by the C RTL, a record is output.

- For files with fixed record length (RFM = FIX) or for files opened with "ctx=bin" or "ctx=xplct", a record is output only when the internal buffer is filled or when the `flush` function is called.

- For files with STREAM_CR record format (RFM = STMCR), the C RTL outputs a record when it encounters a carriage-return character (\r).

- For files with STREAM record format (RFM = STM) the C RTL outputs a record when it encounters a new-line (\n), form feed (\f), or vertical tab (\v) character.

- For all other file types, the C RTL outputs a record when it encounters a new-line (\n) character.

The second part of record output emulation is to write the logical record formed during the first step. The C RTL forms the output record as follows:

- If the record attribute is carriage control (R AT = CR), and if the logical record ends with a new-line character (\n), the C RTL drops the new-line character and writes the logical record with implied carriage control.

- If the record attribute is print carriage control (RAT = PRN), then the C RTL writes the record with print carriage control according to the rules specified by RMS. If the logical record ends with a single new-line character (\n), the C RTL maps the new-line character to an x01 prefix and x8D postfix

character. This is the reverse of the translation for record input files with print carriage-control attributes.

- If the record attributes are Fortran carriage control (RAT = FTN), then the C RTL removes any prefix and/or postfix carriage-control characters and concatenates a single carriage-control byte to the beginning of the record as defined by RMS, with one exception: If the output record ends in a new-line character (\n), the C RTL will remove the new-line character and use the space carriage-control byte. This is the reverse of the translation for record input files with Fortran carriage-control attributes.

  The mapping of Fortran carriage-control can be disabled by using "ctx=nocvt".

- If the logical record is to be written to a terminal device and the last character of the record is a new-line character (\n) the C RTL replaces the new-line character with a carriage-return (\r), and attaches a line-feed character (\n) to the front of the record. The C RTL then writes out the record with no carriage control.

- If the output file record format is variable length with fixed control (RFM = VFC), and the record attributes do not include print carriage control (RAT is not PRN), then the C RTL takes the beginning of the logical record to be the fixed-control header, and reduces the number of bytes written out by the length of the header. These bytes are then used to construct the fixed-control header. If there are too few bytes in the logical record, an error is signaled.

1.7.2.2.1. Accessing Variable-Length or VFC Record Files in Record Mode

When you access a variable-length or VFC record file in record mode, many I/O functions behave differently than they would if they were being used with stream mode. This section describes these differences.

In general, the new-line character (\n) is the record separator for all record modes. On output, when a new-line character is encountered, a record is generated unless you specify an optional argument (such as "ctx=bin"or "ctx=xplct") that affects the interpretation of new lines.

The `read` and `decc$record_read` functions always read at most one record. The `write` and `decc$record_write` functions always generate at least one record.

`decc$record_read` and `decc$record_write` are equivalent, respectively, to `read` and `write`, except that they work with file pointers rather than file descriptors.

Unlike the `read` function, which reads at most one record, the `fread` function can span records. Rather than read *number_items* records (where *number_items* is the third parameter to `fread`), `fread` tries to read the number of bytes equal to *number_items* × *size_of_item* (where *size_of_item* is the second parameter to `fread`). The value returned by `fread` is equal to the number of bytes read divided by *size_of_item*.

However, the `fwrite` function always generates at least *number_items* records.

The `fgets` and `gets` functions read to either a new-line character or a record boundary.

The `fflush` function always generates a record if there is unwritten data in the buffer. The same is true of `close`, `fclose`, `fseek`, `lseek`, `rewind`, and `fsetpos`, all of which perform implicit `fflush` functions.

A record is also generated whenever an attempt is made to write more characters than allowed by the maximum record size.

For more information on these functions, see the Reference Section.

1.7.2.2.2. Accessing Fixed-Length Record Files in Record Mode

When accessing a fixed-length record file in record mode, the I/O functions generally behave as described in Section 1.7.2.2.1.

The `write`, `fwrite`, and `decc$record_write` functions will fail if given a record size that is not an integral multiple of the maximum record size, unless the file was opened with the "ctx=xplct" optional argument specified. All other output functions will generate records at every *n*th byte, where *n* is the maximum record size.

If a new record is forced by `fflush`, the data in the buffer is padded to the maximum record size with null characters.

Note

This padding can cause problems for programs that seek to the end-of-file. For example, if a program were to append data to a file, then seek backwards in the file (causing an `fflush` to occur), and then seek to the end-of-file again, a zero-filled "hole" will have been created between the previous end-of-file and the new end-of-file if the previous end-of-file was not on a record boundary.

1.7.2.3. Example – Difference Between Stream Mode and Record Mode

Example 1.1 demonstrates the difference between stream mode and record mode access.

Example 1.1. Differences Between Stream Mode and Record Mode Access

```
/*      CHAP_1_STREAM_RECORD.C                          */

/* This program demonstrates the difference between  */
/* record mode and stream mode input/output.         */

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

void process_records(const char *fspec, FILE * fp);

main()
{
    FILE *fp;

    fp = fopen("example-fixed.dat", "w", "rfm=fix", "mrs=40", "rat=none");
    if (fp == NULL) {
        perror("example-fixed");
        exit(EXIT_FAILURE);
    }
    printf("Record mode\n");
    process_records("example-fixed.dat", fp);
    fclose(fp);

    printf("\nStream mode\n");
    fp = fopen("example-streamlf.dat", "w");
    if (fp == NULL) {
        perror("example-streamlf");
```

```
        exit(EXIT_FAILURE);
    }
    process_records("example-streamlf.dat", fp);
    fclose(fp);
}

void process_records(const char *fspec, FILE * fp)
{
    int i,
        sts;

    char buffer[40];

    /* Write records of all 1's, all 2's and all 3's */
    for (i = 0; i < 3; i++) {
        memset(buffer, '1' + i, 40);
        sts = fwrite(buffer, 40, 1, fp);
        if (sts != 1) {
            perror("fwrite");
            exit(EXIT_FAILURE);
        }
    }

    /* Rewind the file and write 10 characters of A's, then 10 B's, */
    /* then 10 C's.                                                 */
    /*                                                              */
    /* For stream mode, each fwrite call outputs 10 characters      */
    /* and advances the file position 10 characters                 */
    /* characters.                                                  */
    /*                                                              */
    /* For record mode, each fwrite merges the 10 characters  into  */
    /* the existing 40-character record, updates the record and     */
    /* advances the file position 40 characters to the next record. */
    rewind(fp);
    for (i = 0; i < 3; i++) {
        memset(buffer, 'A' + i, 10);
        sts = fwrite(buffer, 10, 1, fp);
        if (sts != 1) {
            perror("fwrite2");
            exit(EXIT_FAILURE);
        }
    }

    /* Now reopen the file and output the records. */

    fclose(fp);
    fp = fopen(fspec, "r");
    for (i = 0; i < 3; i++) {
        sts = fread(buffer, 40, 1, fp);
        if (sts != 1)
            perror("fread");
            printf("%.40s\n", buffer);
    }

    return;
}
```

Running this program produces the following output:

```
Record Mode
AAAAAAAAAA11111111111111111111111111111
BBBBBBBBBB22222222222222222222222222222
CCCCCCCCCC33333333333333333333333333333

Stream mode
AAAAAAAAAABBBBBBBBBBCCCCCCCCCC1111111111
22222222222222222222222222222222222222222
33333333333333333333333333333333333333333
```

1.8. Specific Portability Concerns

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

The C RTL provides ANSI C defined library functions as well as many commonly available APIs and a few OpenVMS extensions. See Section 1.4 for specific standards, portions of which are implemented by the C RTL. Attempts have been made to maintain complete portability in functionality whenever possible. Many of the Standard I/O and UNIX I/O functions and macros contained in the C RTL are functionally equivalent to those of other implementations.

The RTL function and macro descriptions elaborate on issues presented in this section and describe concerns not documented here.

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

- VSI C for OpenVMS Systems does not implement the global symbols `end`, `edata`, and `etext`.

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

  For example, if a program tries to reference location 0 on an OpenVMS system, a hardware error (ACCVIO) is returned and the program terminates abnormally. OpenVMS systems reserve the first page of address space to catch incorrect pointer references, such as a reference to a location pointed to by a null pointer. For this reason, some existing programs that run on some UNIX systems may fail and you should modify them, as necessary. (Tru64 UNIX and OpenVMS, however, are compatible in this regard.)

- Some C programmers code all external declarations in `#include` files. Then, specific declarations that require initialization are redeclared in the relevant module. This practice causes the VSI C compiler to issue a warning message about multiply declared variables in the same compilation. One way to avoid this warning is to make the redeclared symbols `extern` variables in the `#include` files.

- VSI C does not support `asm` calls on OpenVMS VAX and Integrity server systems. They are supported on OpenVMS Alpha systems. See the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/] for more information on intrinsic functions.

- Some C programs call the counted string functions `strcmpn` and `strcpyn`. These names are not used by VSI C for OpenVMS systems. Instead, you can define macros that expand the `strcmpn` and `strcpyn` names into the equivalent, ANSI-compliant names `strncmp` and `strncpy`.
- The VSI C for OpenVMS compiler does not support the following initialization form:

  ```
  int  foo  123;
  ```

  Programs using this form of initialization must be changed.
- VSI C for OpenVMS systems predefines several compile-time macros such as `_ _vax`, `_ _alpha`, `_ _ia64` `_ _32BITS`, `_ _vms`, `_ _vaxc`, `_ _VMS_VER`, `_ _DECC_VER`, `_ _D_FLOAT`, `_ _G_FLOAT`, `_ _IEEE_FLOAT`, `_ _X_FLOAT`, and others. These predefined macros are useful for programs that must be compatible on other machines and operating systems. For more information, see the predefined macro chapter of the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/].
- The ANSI C language does not guarantee any memory order for the variables in a declaration. For example:

  ```
  int  a, b, c;
  ```
- Depending on the type of external linkage requested, `extern` variables in a program may be treated differently using VSI C on OpenVMS systems than they would on UNIX systems. See the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/] for more information.
- The dollar sign ($) is a legal character in VSI C for OpenVMS identifiers, and can be used as the first character.
- The ANSI C language does not define any order for evaluating expressions in function parameter lists or for many kinds of expressions. The way in which different C compilers evaluate an expression is only important when the expression has side effects. Consider the following examples:

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

  x = func_y() + func_z();

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

  Neither VSI C nor any other C compiler can guarantee that such expressions evaluate in the same order on all C compilers.
- The size of a VSI C variable of type `int` is 32bits on OpenVMS systems. You will have to modify programs that are written for other machines and that assume a different size for a variable of type `int`. A variable of type `long` is the same size (32 bits) as a variable of type `int`.
- The C language defines structure alignment to be dependent on the machine for which the compiler is designed. On OpenVMS Alpha systems, VSI C aligns structure members on natural boundaries, unless `#pragma nomember_alignment` is specified. Other implementations may align structure members differently.
- References to structure members in VSI C cannot be vague. For more information, see the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/].
- Registers are allocated based upon how often a variable is used, but the `register` keyword gives the compiler a strong hint that you want to place a particular variable into a register. Whenever

possible, the variable is placed into a register. Any scalar variable with the storage class `auto` or `register` can be allocated to a register as long as the variable's address is not taken with the ampersand operator (&) and it is not a member of a structure or union.

1.8.1. Reentrancy

The C RTL supports an improved and enhanced reentrancy. The following types of reentrancy are supported:

- AST reentrancy uses the _BBSSI built-in function to perform simple locking around critical sections of RTL code, but it may also disable asynchronous system traps (ASTs) in locked regions of code. This type of locking should be used when AST code contains calls to C RTL I/O routines.

  Failure to specify AST reentrancy might cause I/O routines to fail, setting `errno` to EALREADY.

- MULTITHREAD reentrancy is designed to be used in threaded programs such as those that use the POSIX threads library. It performs threads locking and never disables ASTs. The POSIX Threads library must be available on your system to use this form of reentrancy.

- TOLERANT reentrancy uses the _BBSSI built-in function to perform simple locking around critical sections of RTL code, but ASTs are not disabled. This type of locking should be used when ASTs are used and must be delivered immediately.

- NONE gives optimal performance in the C RTL, but does absolutely no locking around critical sections of RTL code. It should only be used in a single-threaded environment when there is no chance that the thread of execution will be interrupted by an AST that would call the C RTL.

For non-threaded processes, the default reentrancy type is TOLERANT. When the threads library is loaded, the reentrancy level is implicitly set to C$C_MULTITHREAD, and the application cannot change it after that.

You can set the reentrancy type by compiling with the /REENTRANCY command-line qualifier or by calling the `decc$set_reentrancy` function. This function must be called exclusively from non-AST level.

When programming an application using multiple threads or ASTs, consider three classes of functions:

- Functions with no internal data
- Functions with thread-local internal data
- Functions with processwide internal data

Most functions have no internal data at all. For these functions, synchronization is necessary only if the parameter is used by the application in multiple threads or in both AST and non-AST contexts. For example, although the `strcat` function is ordinarily safe, the following is an example of unsafe usage:

```
extern char buffer[100];
void routine1(char *data) {
    strcat( buffer, data );
}
```

If `routine1` executed concurrently in multiple threads, or if `routine1` is interrupted by an AST routine that calls it,the results of the `strcat` call are unpredictable.

The second class of functions are those that have thread-local static data. Typically, these are routines in the library that return a string where the application is not permitted to free the storage for the string.

These routines are thread-safe but not AST-reentrant. This means they can safely be called concurrently, and each thread will have its own copy of the data. They cannot be called from AST routines if it is possible that the same routine was executing in non-AST context. The routines in this class are:

```
asctime         stat
ctermid         strerror
ctime           strtok
cuserid         VAXC$ESTABLISH
gmtime          the errno variable
localtime       wcstok
perror
```

All the socket functions are also included in this list if the TCP/IP product in use is thread-safe.

The third class of functions are those that affect processwide data. These functions are neither thread-safe nor AST-reentrant. For example, `sigsetmask` establishes the processwide signal mask. Consider a routine like the following:

```
void update_data
base()
{
    int old_mask;

    old_mask = sigsetmask( 1 << (SIGINT - 1));
        /* Do work here that should not be aborted. */
    sigsetmask( old_mask );
}
```

If `update_database` was called concurrently in multiple threads, thread 1 might unblock SIGINT while thread 2 was still performing work that should not be aborted.

The routines in this class are:

- All the `signal` routines
- All the `exec` routines
- The `exit`, `_exit`, `nice`, `system`, `wait`, `getitimer`, `setitimer`, and `setlocale` routines.

Note

Generally, UTC-based time functions can affect in-memory time-zone information, which is processwide data. However, if the system time zone remains the same during the execution of the application (which is the common case) and the cache of time-zone files is enabled (which is the default), then the `_r` variant of the time functions `asctime_r`, `ctime_r`, `gmtime_r` and `localtime_r` is both thread-safe and AST-reentrant.

If, however, the system time zone can change during the execution of the application or the cache of time-zone files is not enabled, then both variants of the UTC-based time functions belong to the third class of functions, which are neither thread-safe nor AST-reentrant.

Some functions remain inherently nonthread-safe regardless of the reentrancy type. They are:

```
execl       exit
execle      _exit
```

```
execlp      nice
execv       system
execve      vfork
execvp
```

1.8.2. Multithread Restrictions

Mixing the multithread programming model and the OpenVMS AST programming model in the same application is not recommended. The application has no mechanism to control which thread gets interrupted by an AST. This can result in are source deadlock if the thread holds a resource that is also needed by the AST routine. The following functions use mutexes. To avoid a potential resource deadlock, do not call them from AST functions in a multithreaded application.

- All the I/O functions
- All the socket functions
- All the signal functions
- `vfork`, `exec`, `wait`, `system`
- `catgets`
- `set_new_handler` (C++ only)
- `getenv`
- `rand` and `srand`
- `exit` and `_exit`
- `clock`
- `nice`
- `times`
- `ctime`, `localtime`, `asctime`, `mktime`

1.9. 64-Bit Pointer Support

This section is for application developers who need to use 64-bit virtual memory addressing on OpenVMS Alpha Version 7.0 or higher.

OpenVMS Alpha 64-bit virtual addressing support makes the 64-bit virtual address space defined by the Alpha architecture available to both the OpenVMS operating system and its users. It also allows per-process virtual addressing for accessing dynamically mapped data beyond traditional 32-bit limits.

The VSI C Run-Time Library on OpenVMS Alpha Version 7.0 systems and higher includes the following features in support of 64-bit pointers:

- Guaranteed binary and source compatibility of existing programs
- No impact on applications that are not modified to exploit 64-bit support

- Enhanced memory allocation routines that allocate 64-bit memory
- Widened function parameters to accommodate 64-bit pointers
- Dual implementations of functions that need to know the pointer size used by the caller
- New information available to the DEC C Version 5.2 compiler or higher to seamlessly call the correct implementation
- Ability to explicitly call either the 32-bit or 64-bit form of functions for applications that mix pointer sizes
- A single shareable image for use by 32-bit and 64-bit applications

1.9.1. Using the VSI C Run-Time Library

The VSI C Run-Time library on OpenVMS Alpha Version 7.0 systems and higher can generate and accept 64-bit pointers. Functions that require a second interface to be used with 64-bit pointers reside in the same object libraries and shareable images as their 32-bit counterparts. No new object libraries or shareable images are introduced. Using 64-bit pointers does not require changes to your link command or link options files.

The VSI C 64-bit environment allows an application to use both 32-bit and 64-bit addresses. For more information about how to manipulate pointer sizes, see the /POINTER_SIZE qualifier and `#pragma pointer_size` and `#pragma required_pointer_size` preprocessor directives in the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/].

The /POINTER_SIZE qualifier requires you to specify a value of 32 or 64. This value is used as the default pointer size within the compilation unit. You can compile one set of modules using 32-bit pointers and another set using 64-bit pointers. Care must be taken when these two separate groups of modules call each other.

Use of the /POINTER_SIZE qualifier also influences the processing of C RTL header files. For those functions that have a 32-bit and 64-bit implementation, specifying /POINTER_SIZE enables function prototypes to access both functions, regardless of the actual value supplied to the qualifier. In addition, the value specified to the qualifier determines the default implementation to call during that compilation unit.

The `#pragma pointer_size` and `#pragma required_pointer_size` preprocessor directives can be used to change the pointer size in effect within a compilation unit. You can default pointers to 32-bit pointers and then declare specific pointers within the module as 64-bit pointers. You would also need to specifically call the `_malloc64` form of `malloc` to obtain memory from the 64-bit memory area.

1.9.2. Obtaining 64-Bit Pointers to Memory

The C RTL has many functions that return pointers to newly allocated memory. In each of these functions, the application owns the memory pointed to and is responsible for freeing that memory.

Functions that allocate memory are:

```
malloc
calloc
realloc
```

`strdup`

Each of these functions have a 32-bit and a 64-bit implementation. When the /POINTER_SIZE qualifier is used, the following functions can also be called:

`_malloc32`, `_malloc64`
`_calloc32`, `_calloc64`
`_realloc32`, `_realloc64`
`_strdup32`, `_strdup64`

When /POINTER_SIZE=32 is specified, all `malloc` calls default to `_malloc32`.

When /POINTER_SIZE=64 is specified, all `malloc` calls default to `_malloc64`.

Regardless of whether the application calls a 32-bit or 64-bit memory allocation routine, there is still a single `free` function. This function accepts either pointer size.

Be aware that the memory allocation functions are the only ones that return pointers to 64-bit memory. All C RTL structure pointers returned to the calling application (such as a FILE, WINDOW, or DIR) are always 32-bit pointers. This allows both 32-bit and 64-bit callers to pass these structure pointers within the application.

1.9.3. VSI C Header Files

The header files distributed with OpenVMS support 64-bit pointers. Each function prototype whose signature contains a pointer is constructed to indicate the size of the pointer accepted.

A 32-bit pointer can be passed as an argument to functions that accept either a 32-bit or 64-bit pointer for that argument.

A 64-bit pointer, however, cannot be passed as an argument to a function that accepts a 32-bit pointer. Attempts to do this are diagnosed by the compiler with a MAYLOSEDATA message. The diagnostic message IMPLICITFUNC means the compiler can do no additional pointer-size validation for calls to that function. If this function is a C RTL function, refer to the reference section of this manual for the name of the header file that defines that function.

You might find the following pointer-size compiler diagnostics useful:

- %CC-IMPLICITFUNC

  A function prototype was not found before using the specified function. The compiler and run-time system rely on prototype definitions to detect incorrect pointer-size usage. Failure to include the proper header files can lead to incorrect results and/or pointer truncation.

- %CC-MAYLOSEDATA

  A truncation is necessary to do this operation. The operation could be passing a 64-bit pointer to a function that does not support a 64-bit pointer in the given context. It could also be a function returning a 64-bit pointer to a calling application that is trying to store that return value in a 32-bit pointer.

- %CC-MAYHIDELOSS

  This message (when enabled) helps expose real MAYLOSEDATA messages that are being suppressed because of a cast operation. To enable this warning, compile with the qualifier /WARNINGS=ENABLE=MAYHIDELOSS.

1.9.4. Functions Affected

The C RTL accommodates applications that use only 32-bit pointers, only 64-bit pointers, or combinations of both. To use 64-bit memory, you must, at a minimum, recompile and relink an application. The amount of source code change required depends on the application itself, calls to other run-time libraries, and the combinations of pointer sizes used.

With respect to 64-bit pointer support, the functions in the C RTL fall into four categories:

- Functions not impacted by choice of pointer size
- Functions enhanced to accept either pointer size
- Functions having a 32-bit and 64-bit implementation
- Functions that accept only 32-bit pointers

From an application developer's perspective, the first two types of functions are the easiest to use in either a single- or mixed-pointer mode.

The third type requires no modifications when used in a single-pointer compilation, but might require source code changes when used in a mixed-pointer mode.

The fourth type requires careful attention whenever 64-bit pointers are used.

1.9.4.1. No Pointer-Size Impact

The choice of pointer size has no impact on a function if its prototype contains no pointer-related parameters or return values. The mathematical functions are good examples of this.

Even some functions in this category that do have pointers in their prototype are not impacted by pointer size. For example, `strerror` has the prototype:

```
char * strerror (int error_number);
```

This function returns a pointer to a character string, but this string is allocated by the C RTL. As a result, to support both 32-bit and 64-bit applications, these types of pointers are guaranteed to fit in a 32-bit pointer.

1.9.4.2. Functions Accepting Both Pointer Sizes

The Alpha architecture supports 64-bit pointers. The OpenVMS Alpha calling standard specifies that all arguments are actually passed as 64-bit values. Before OpenVMS Alpha Version 7.0, all 32-bit addresses passed to procedures were sign-extended into this 64-bit parameter. The called function declared the parameters as 32-bit addresses, which caused the compiler to generate 32-bit instructions (such as LDL) to manipulate these parameters.

Many functions in the C RTL are enhanced to receive the full 64-bit address. For example, consider `strlen`:

```
size_t strlen (const char *string);
```

The only pointer in this function is the character-string pointer. If the caller passes a 32-bit pointer, the function works with the sign-extended 64-bit address. If the caller passes a 64-bit address, the function works directly with that address.

The C RTL continues to have only a single entry point for functions in this category. There are no source-code changes required to add any of the four pointer-size options for functions of this type. The OpenVMS documentation refers to these functions as 64-bit friendly.

1.9.4.3. Functions with Two Implementations

There are many reasons why a function might need one implementation for 32-bit pointers and another for 64-bit pointers. Some of these reasons include:

- The pointer size of the return value is the same size as the pointer size of one of the arguments. If the argument is 32 bits, the return value is 32 bits. If the argument is 64 bits, the return value is 64 bits.
- One of the arguments is a pointer to an object whose size is pointer-size sensitive. To know how many bytes are being pointed to, the function must know if the code was compiled in 32-bit or 64-bit pointer-size mode.
- The function returns the address of dynamically allocated memory. The memory is allocated in 32-bit space when compiled for 32-bit pointers, and is allocated in 64-bit space when compiled for 64-bit pointers.

From the application developer's point of view, there are three function prototypes for each of these functions. The `<string.h>` header file contains many functions whose return value is dependent upon the pointer size used as the first argument to the function call. For example, consider the `memset` function. The header file defines three entry points for this function:

```
void * memset   (void *memory_pointer, int character, size_t size);
void *_memset32 (void *memory_pointer, int character, size_t size);
void *_memset64 (void *memory_pointer, int character, size_t size);
```

The first prototype is the function that your application would currently call if using this function. The compiler changes a call to `memset` into a call to either `_memset32` when compiled with /POINTER_SIZE=32, or `_memset64` when compiled with /POINTER_SIZE=64.

You can override this default behavior by directly calling either the 32-bit or the 64-bit form of the function. This accommodates applications using mixed-pointer sizes, regardless of the default pointer size specified with the /POINTER_SIZE qualifier.

If the application is compiled *without* specifying the /POINTER_SIZE qualifier, *neither* the 32-bit specific *nor* the 64-bit specific function prototypes are defined. In this case, the compiler automatically calls the 32-bit interface for all interfaces having dual implementations.

Table 1.5 shows the C RTL functions that have dual implementations to support 64-bit pointer size. When compiling with the /POINTER_SIZE qualifier, calls to the unmodified function names are changed to calls to the function interface that matches the pointer size specified on the qualifier.

Table 1.5. Functions with Dual Implementations

`basename`	`bsearch`	`calloc`	`catgets`
`ctermid`	`cuserid`	`dirname`	`fgetname`
`fgets`	`fgetws`	`gcvt`	`getcwd`
`getname`	`getpwent`	`getpwnam`	`getpwnam_r`
`getpwuid`	`getpwuid_r`	`gets`	`index`
`longname`	`malloc`	`mbsrtowcs`	`memccpy`

memchr	memcpy	memmove	memset
mktemp	mmap	qsort	readv
realloc	rindex	strcat	strchr
strcpy	strdup	strncat	strncpy
strpbrk	strptime	strrchr	strsep
strstr	strtod	strtok	strtok_r
strtol	strtoll	strtoq	strtoul
strtoull	strtouq	tmpnam	wcscat
wcschr	wcscpy	wcsncat	wcsncpy
wcspbrk	wcsrchr	wcsrtombs	wcsstr
wcstok	wcstol	wcstoul	wcswcs
wmemchr	wmemcpy	wmemmove	wmemset
writev	glob	globfree	

Table 1.6 shows the TCP/IP socket routines that have dual implementations to support 64-bit pointer size.

Table 1.6. Socket Routines with Dual Implementations

freeaddrinfo	getaddrinfo
recvmsg	sendmsg

1.9.4.4. Socket Transfers Greater than 64 KB

Starting with OpenVMS Version 8.3, support is added for socket transfers greater than 64 KB for the following socket routines:

send	recv	read
sendto	recvfrom	write
sendmsg	recvmsg	

1.9.4.5. Functions Requiring Explicit use of 64-Bit Structures

A few functions require explicit use of 64-bit structures when compiling /POINTER_SIZE=LONG. This is necessary for functions that have recently had 64-bit support added to avoid unexpected run-time errors by inadvertently mixing 32-bit and 64-bit versions of structures.

Consider the following two functions:

```
sendmsg recvmsg
```

These functions previously offered 32-bit support only, even when compiled with /POINTER_SIZE=LONG. In order to preserve the previous behavior of 32-bit pointer support in those functions even when compiled with /POINTER_SIZE=LONG, these two functions do not follow the normal convention for 32-bit and 64-bit support as documented in the previous section.

The following variants of these functions, and the corresponding structures they use, have been added to the C RTL to provide 64-bit support:

```
Function       Structure
--------       ---------
__recvmsg32    __msghdr32
__recvmsg64    __msghdr64
__sendmsg32    __msghdr32
__sendmsg64    __msghdr64
```

When compiling the standard versions of these functions, the following behavior occurs:

- With /POINTER_SIZE=32 specified, the compiler converts the call to the 32-bit version of the function. For example, `recvmsg` is converted to `__recvmsg32`.
- With /POINTER_SIZE=64 specified, the compiler converts the call to the 64-bit version of the function. For example, `recvmsg` is converted to `__recvmsg64`.
- When the /POINTER_SIZE qualifier is not specified, neither the 32-bit-specific nor the 64-bit-specific function prototypes are defined.

However, a similar conversion of the corresponding structures does *not* occur for these functions. This behavior is necessary because these structures existed before OpenVMS Version 7.3-2 as 32-bit versions only, even when compiled with /POINTER_SIZE=LONG. Implicitly changing the size of the structure could result in unexpected run-time errors.

When compiling programs that use the standard version of these functions for 64-bit support, you must use the 64-bit-specific definition of the related structure. With /POINTER_SIZE=64 specified, compiling a program with the standard function name and standard structure definition will result in compiler PTRMISMATCH warning messages.

For example, you insert the `recvmsg` routine into your program along with the standard definition of the `msghdr` structure. Compiling the program with /POINTER=64 or /POINTER=LONG results in the following warning messages being displayed:

```
struct msghdr msgin;
int retval = recvmsg(sock, &msgin, flag);

recvmsg(sock, &msgin, flag);
....^
%CC-W-PTRMISMATCH, In this statement, the referenced type of the pointer
value "&msgin" is "long pointer to struct msghdr", which is not compatible
with "long pointer to struct __msghdr64".
```

When compiling for 64 bits, you need to use the 64-bit-specific version of the related structure. In the previous example, the declaration of the `msgin` structure could be changed to the following:

```
struct __msghdr64 *msgin;
```

Or, to provide flexibility between 32-bit and 64-bit compilations, the `msgin` structure could be declared as follows:

```
#if __INITIAL_POINTER_SIZE == 64
struct __msghdr64 *msgin;
#else
struct __msghdr32 *msgin;
#endif
```

1.9.4.6. Functions Restricted to 32-Bit Pointers

A few functions in the C RTL do not support 64-bit pointers. If you try to pass a 64-bit pointer to one of these functions, the compiler generates a %CC-W-MAYLOSEDATA warning. Applications compiled with /POINTER_SIZE=64 might need to be modified to avoid passing 64-bit pointers to these functions.

Table 1.7 shows the functions restricted to using 32-bit pointers. The C RTL offers no 64-bit support for these functions. You must ensure that only 32-bit pointers are used with these functions.

Table 1.7. Functions Restricted to 32-Bit Pointers

atexit	initstate
iconv	

Table 1.8 shows functions that make callbacks to user-supplied functions as part of processing that function call. The callback procedures are not passed 64-bit pointers.

Table 1.8. Callbacks that Pass Only 32-Bit Pointers

decc$from_vms	decc$to_vms
ftw	tputs

1.9.5. Reading Header Files

This section introduces the pointer-size manipulations used in the C RTL header files. Use the following examples to become more proficient in reading these header files and to help modify your own header files.

```
:#if __INITIAL_POINTER_SIZE ❶
#   if (__VMS_VER < 70000000) || !defined ALPHA ❷
#      error " Pointer size usage not permitted before OpenVMS Alpha V7.0"
#   endif
#   pragma __pointer_size __save ❸
#   pragma __pointer_size 32 ❹
#endif
:
:
#if __INITIAL_POINTER_SIZE ❺
#   pragma __pointer_size 64
#endif
:
:
#if __INITIAL_POINTER_SIZE ❻
#   pragma __pointer_size __restore
#endif
:
```

All VSI C compilers that support the /POINTER_SIZE qualifier predefine the __INITIAL_POINTER_SIZE macro. The C RTL header files take advantage of the ANSI rule that if a macro is not defined, it has an implicit value of 0.

The macro is defined as 32 or 64 when the /POINTER_SIZE qualifier is used. It is defined as 0 if the qualifier is not used. The statement at ❶ can be read as "if the user has specified either /POINTER_SIZE=32 or /POINTER_SIZE=64 on the command line".

The C compiler is supported on many OpenVMS versions. The lines at ❷ generate an error message if the target of the compilation is one that does not support 64-bit pointers.

A header file cannot assume anything about the actual pointer-size context in effect at the time the header file is included. Furthermore, the VSI C compiler offers only the _ _INITIAL_POINTER_SIZE macro and a mechanism to change the pointer size, but not a way to determine the current pointer size.

All header files that have a dependency on pointer sizes are responsible for saving ❸, initializing ❹, altering ❺, and restoring ❻ the pointer-size context.

```
:#ifndef __CHAR_PTR32 ❶
#   define __CHAR_PTR32 1
    typedef char * __char_ptr32;
    typedef const char * __const_char_ptr32;
#endif
:
:
#if __INITIAL_POINTER_SIZE
#   pragma __pointer_size 64
#endif
:
:
#ifndef __CHAR_PTR64 ❷
#   define __CHAR_PTR64 1
    typedef char * __char_ptr64;
    typedef const char * __const_char_ptr64;
#endif
:
```

Some function prototypes need to refer to a 32-bit pointer when in a 64-bit pointer-size context. Other function prototypes need to refer to a 64-bit pointer when in a 32-bit pointer-size context.

VSI C binds the pointer size used in a `typedef` at the time the `typedef` is made. Assuming this header file is compiled with no /POINTER_SIZE qualifier or with /POINTER_SIZE=SHORT, the `typedef` declaration of `__char_ptr32` ❶ is made in a 32-bit context. The `typedef` declaration of `__char_ptr64` ❷ is made in a 64-bit context.

```
:
#if __INITIAL_POINTER_SIZE
#   if (__VMS_VER < 70000000) || !defined __ALPHA
#      error " Pointer size usage not permitted before OpenVMS Alpha V7.0"
#   endif
#   pragma __pointer_size __save
#   pragma __pointer_size 32
#endif
:
❶
:
#if __INITIAL_POINTER_SIZE ❷
#   pragma __pointer_size 64
#endif
:
❸
:
int abs (int __j); ❹
:
__char_ptr32 strerror (int __errnum); ❺
:
```

Before declaring function prototypes that support 64-bit pointers, the pointer context is changed ❷ from 32-bit pointers to 64-bit pointers.

Functions restricted to 32-bit pointers are placed in the 32-bit pointer context section ❶ of the header file. All other functions are placed in the 64-bit context section ❸ of the header file.

Functions that have no pointer-size impact (❹ and ❺) are located in the 64-bit section. Functions that have no pointer-size impact except for a 32-bit address return value ❺ are also in the 64-bit section, and use the 32-bit specific `typedef`s previously discussed.

```
:
#if __INITIAL_POINTER_SIZE
#   pragma __pointer_size 64
#endif
:
:
#if __INITIAL_POINTER_SIZE == 32 ❶
#   pragma __pointer_size 32
#endif
:
char *strcat (char *__s1, __const_char_ptr64 __s2); ❷
:
#if __INITIAL_POINTER_SIZE
#   pragma __pointer_size 32
    :
    char *_strcat32  (char *__s1, __const_char_ptr64 __s2); ❸
    :
#   pragma __pointer_size 64
    :
    char *_strcat64  (char *__s1, const char *__s2); ❹
    :
#endif
:
```

This example shows declarations of functions that have both a 32-bit and 64-bit implementation. These declarations are located in the 64-bit section of the header file.

The normal interface to the function ❷ is declared using the pointer size specified on the /POINTER_SIZE qualifier. Because the header file is in 64-bit pointer context and because of the statements at ❶, the declaration at ❷ is made using the same pointer-size context as the /POINTER_SIZE qualifier.

The 32-bit specific interface ❸ and the 64-bit specific interface ❹ are declared in 32-bit and 64-bit pointer-size context, respectively.

Chapter 2. Understanding Input and Output

There are three types of input and output (I/O) in the VSI C Run-Time Library (C RTL): UNIX, Standard, and Terminal. Table 2.1 lists all the I/O functions and macros found in the C RTL. For more detailed information on each function and macro, see the Reference Section.

Table 2.1. I/O Functions and Macros

Function or Macro	Description
UNIX I/O – Opening and Closing Files	
`close`	Closes the file associated with a file descriptor.
`creat`	Creates a new file.
`dup` `dup2`	Allocate a new descriptor that refers to a file specified by a file descriptor returned by `open`, `creat`, or `pipe`.
`open`	Opens a file and positions it at its beginning.
UNIX I/O – Reading from Files	
`read`	Reads bytes from a file and places them in a buffer.
UNIX I/O – Writing to Files	
`write`	Writes a specified number of bytes from a buffer to a file.
UNIX I/O – Maneuvering in Files	
`lseek`	Positions a stream file to an arbitrary byte position and returns the new position as an `int`.
UNIX I/O – Additional X/Open I/O Functions and Macros	
`fstat` `stat`	Access information about the file descriptor or the file specification.

`flockfile`
`ftrylockfile`
`funlockfile`
	File-pointer-locking functions.

`fsync`	Writes to disk any buffered information for the specified file.
`getname`	Returns the file specification associated with a file descriptor.
`isapipe`	Returns 1 if the file descriptor is associated with a pipe and 0 if it is not.
`isatty`	Returns 1 if the specified file descriptor is associated with a terminal and 0 if it is not.
`lwait`	Waits for completion of pending asynchronous I/O.
`ttyname`	Returns a pointer to the null-terminated name of the terminal device associated with file descriptor 0, the default input device.
Standard I/O – Opening and Closing Files	
`fclose`	Closes a function by flushing any buffers associated with the file control block, and freeing the file control block and buffers previously associated with the file pointer.
`fdopen`	Associates a file pointer with a file descriptor returned by an `open`, `creat`, `dup`, `dup2`, or `pipe` function.

Function or Macro	Description
`fopen`	Opens a file by returning the address of a FILE structure.
`freopen`	Substitutes the file, named by a file specification, for the open file addressed by a file pointer.
Standard I/O – Reading from Files	

`fgetc`
`getc`
`fgetwc`
`getw`
`getwc`
	Return characters from a specified file.

`fgets` `fgetws`	Read a line from a specified file and stores the string in an argument.
`fread`	Reads a specified number of items from a file.

`getline`
`getwline`
`getdelim`
`getwdelim`
	Read characters from an input stream.

`fscanf`
`fwscanf`
`vfscanf`
`vfwscanf`
	Perform formatted input from a specified file.

`sscanf`
`swscanf`
`vsscanf`
`vswscanf`
	Perform formatted input from a character string in memory.

`ungetc` `ungetwc`	Push back a character into the input stream and leave the stream positioned before the character.
Standard I/O – Writing to Files	

`fprintf`
`fwprintf`
`vfprintf`
`vfwprintf`
	Perform formatted output to a specified file.

`fputc`
`putc`
`putw`
`putwc`
`fputwc`
	Write characters to a specified file.

`fputs` `fputws`	Write a character string to a file without copying the string's null terminator.
`fwrite`	Writes a specified number of items to a file.

`sprintf`
`swprintf`
`vsprintf`
`vswprintf`
	Perform formatted output to a string in memory.

Standard I/O – Maneuvering in Files	
`fflush`	Sends any buffered information for the specified file to RMS.

Function or Macro	Description
`fgetpos`	Stores the current value of the file position indicator for the stream.
`fsetpos`	Sets the file position indicator for the stream according to the value of the object pointed to.
`fseek` `fseeko`	Position the file to the specified byte offset in the file.
`ftell` `ftello`	Return the current byte offset to the specified stream file.
`rewind`	Sets the file to its beginning.
Standard I/O – Additional Standard I/O Functions and Macros	
`access`	Checks a file to see whether a specified access mode is allowed.
`clearerr`	Resets the error and end-of-file indications for a file.
`feof`	Tests a file to see if the end-of-file has been reached.
`ferror`	Returns a nonzero integer if an error has occurred while reading or writing a file.
`fgetname`	Returns the file specification associated with a file pointer.
`fileno`	Returns an integer file descriptor that identifies the specified file.
`ftruncate`	Truncates a file at the specified position.
`fwait`	Waits for completion of pending asynchronous I/O.
`fwide`	Sets the orientation a stream.
`mktemp` `mkostemp`	Create a unique filename from a template.
`remove` `delete`	Delete a file.
`rename`	Gives a new name to an existing file.
`setbuf` `setvbuf`	Associate a buffer with an input or output file.
`tmpfile`	Creates a temporary file that is opened for update.
`tmpnam`	Creates a character string that can be used in place of the file-name argument in other function calls.
Terminal I/O – Reading from Files	
`getchar` `getwchar`	Read a single character from the standard input (`stdin`).
`gets`	Reads a line from the standard input (`stdin`).

`scanf`
`wscanf`
`vscanf`
`vwscanf`
	Perform formatted input from the standard input.

Terminal I/O – Writing to Files	

`printf`
`wprintf`
`vprintf`
`vwprintf`
	Perform formatted output to the standard output (`stdout`).

Function or Macro	Description
`putchar` `putwchar`	Write a single character to the standard output and return the character.
`puts`	Writes a character string to the standard output followed by a new-line character.

2.1. Using RMS from RTL Routines

When you create a file using the C RTL I/O functions and macros, you can supply values for many RMS file attributes, including:

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

See the description of the `creat` function in the Reference Section for information on these values.

Other functions that allow you to set these values include `open`, `fopen`, and `freopen`.

For more information about RMS, see the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-users-guide-for-openvms-systems/].

2.2. UNIX I/O and Standard I/O

UNIX I/O functions are UNIX system services, now standardized by ISO POSIX-1 (the ISO Portable Operating System Interface).

UNIX I/O functions use file descriptors to access files. A *file descriptor* is an integer that identifies the file. A file descriptor is declared in the following way, where *file_desc* is the name of the file descriptor:

```
int  file_desc;
```

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

```
file_desc1 = creat("INFILE.DAT", 0, "rat=cr", "rfm=var");
```

This statement creates the file, INFILE.DAT, with file access mode 0, carriage-return control, variable-length records, and it associates the variable `file_desc1` with the file. When the file is accessed for other operations, such as reading or writing, the file descriptor is used to refer to the file. For example:

```
write(file_desc1, buffer, sizeof(buffer));
```

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

There may be circumstances when you should use UNIX I/O functions and macros instead of the Standard I/O functions and macros. For a detailed discussion of both forms of I/O and how they manipulate the RMS file formats, see Chapter 1.

Standard I/O functions are specified by the ANSI C Standard.

Standard I/O functions add buffering to the features of UNIX I/O and use file pointers to access files. A *file pointer* is an object of type `FILE *`, which is a `typedef` defined in the `<stdio.h>` header file as follows:

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

The `_iobuf` identifier is also defined in `<stdio.h>`.

To declare a file pointer, use the following:

```
FILE  *file_ptr;
```

Use the Standard I/O `fopen` function to create or open an existing file. For example:

```
#include <stdio.h>

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

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

The C RTL provides the following functions for converting between file descriptors and file pointers:

- `fileno` – returns the file descriptor associated with the specified file pointer.
- `fdopen` – associates a file pointer with a file descriptor returned by an `open`, `creat`, `dup`, `dup2`, or `pipe` function.

2.3. Wide-Character Versus Byte I/O Functions

The wide-character I/O functions provide operations similar to most of the byte I/O functions, except that the fundamental units internal to the wide-character functions are wide characters.

However, the external representation (in files) is a sequence of multibyte characters, not wide characters. For the wide-character formatted input and output functions:

- The wide-character formatted *input* functions (such as `fwscanf`) always read a sequence of multibyte characters from files, regardless of the specified directive and, before any further processing, convert this sequence to a sequence of wide characters.
- The wide-character formatted *output* functions (such as `fwprintf`) write wide characters to output files by first converting wide-character argument types to a sequence of multibyte characters, then calling the underlying operating system output primitives.

Byte I/O functions cannot handle state-dependent encodings. Wide-character I/O functions can. They accomplish this by associating each wide-character stream with a conversion-state object of type `mbstate_t`.

The wide-character I/O functions are:

```
fgetwc        fputwc        fwscanf       fwprintf       ungetwc
fgetws        fputws        wscanf        wprintf
getwc         putwc                       vfwprintf
getwchar      putwchar                    vwprintf
```

The byte I/O functions are:

```
fgetc         fputc         fscanf        fprintf         ungetc
fgets         fputs         scanf         printf          fread
getc          putc                        vfprinf         fwrite
gets          puts                        vprintf
getchar       putchar
```

The wide-character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the `fgetwc` function. Each conversion occurs as if a call were made to the `mbrtowc` function with the conversion state described by the stream's own `mbstate_t` object.

The wide-character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the `fputwc` function. Each conversion occurs as if a call were made to the `wcrtomb` function, with the conversion state described by the I/O stream's own `mbstate_t` object.

If a wide-character I/O function encounters an invalid multibyte character, the function sets `errno` to the value EILSEQ.

2.4. Conversion Specifications

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

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

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

Each conversion specification begins with a percent sign (%) and ends with a *conversion specifier*, which is a character that specifies the type of conversion to be performed. Optional characters can appear between the percent sign and the conversion specifier.

For the wide-character formatted I/O functions, the conversion specification is a string of wide characters. For the byte I/O equivalent functions, it is a string of bytes.

Sections 2.4.1 and 2.4.2 describe these optional characters and conversion specifiers.

2.4.1. Converting Input Information

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

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

Each input pointer is an address expression indicating an object whose type matches that of a corresponding conversion specification. Conversion specifications form part of the format string. 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.

A conversion specification consists of the following characters, in the order listed:

- A percent character (%) or the sequence %*n*$ (where *n* is an integer),

  The sequence %*n*$ denotes that the conversion is applied to the *n*th input pointer listed, where *n* is a decimal integer between [1, NL_ARGMAX] (see the `<limits.h>` header file). For example, a conversion specification beginning with %5$ means that the conversion will be applied to the fifth input pointer listed after the format specification. The sequence %$ is invalid.

  If the conversion specification does not begin with the sequence %*n*$, the conversion specification is matched to its input pointer in left-to-right order. You should only use one type of conversion specification (% or %n$) in a format specification.

- One or more optional characters (see Table 2.2).
- A conversion specifier (see Table 2.3).

Table 2.2 shows the characters you can use between the percent sign (%) (or the sequence %*n*$), and the conversion specifier. These characters are optional but, if specified, must occur in the order shown in Table 2.2.

Table 2.2. Optional Characters Between % (or %*n*$) and the Input Conversion Specifier

Character	Meaning
*	An assignment-suppressing character.
field width	A nonzero decimal integer that specifies the maximum field width. For the wide-character input functions, the field width is measured in wide characters. For the byte input functions, the field width is measured in bytes, unless the directive is one of the following: `%lc, %ls, %C, %S, %[` In these cases, the field width is measured in multibyte character units. For programs compiled with /L_DOUBLE=64 (that is, compiled without the default /L_DOUBLE=128), the maximum field width is 1024.
h, l, or L (or ll)	Precede a conversion specifier of `d`, `i`, or `n` with an h if the corresponding argument is a pointer to `shortint` rather than a pointer to `int`; with an l (lowercase ell) if it is a pointer to `long int`; or, for OpenVMS Alpha systems only, with an L or ll (two lowercase ells) if it is a pointer to `__int64`. Precede a conversion specifier of `o`, `u`, or `x` with an h if the corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`; with an l if it is a pointer to `unsigned long int`; or, for OpenVMS Alpha systems only, with an L or ll if it is a pointer to `unsigned __int64`. Precede a conversion specifier of `c`, `s`, or `[` with an l (lowercase ell) if the corresponding argument is a pointer to a `wchar_t`. Finally, precede a conversion specifier of `e`, `f`, or `g` with an l (lowercase ell) if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or with an L if it is a pointer to `long double`. If an h, l, L, or ll appears with any other conversion specifier, then the behavior is undefined.

Table 2.3 describes the conversion specifiers for formatted input.

Table 2.3. Conversion Specifiers for Formatted Input

Specifier	Input Type[1]	Description
d		Expects a decimal integer in the input whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the `base` argument. The corresponding argument must be a pointer to `int`.
i		Expects an integer whose type is determined by the leading input characters. A leading 0 is equated to octal, a leading 0X or 0x is equated to hexadecimal, and all other forms are equated to decimal. The corresponding argument must be a pointer to `int`.
o		Expects an octal integer in the input (with or without a leading 0). The corresponding argument must be a pointer to `int`.

Specifier	Input Type[1]	Description
u		Expects a decimal integer in the input whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the `base` argument.
x		Expects a hexadecimal integer in the input (with or without a leading 0x). The corresponding argument must be a pointer to `unsignedint`.
c	Byte	Expects a single byte in the input. The corresponding argument must be a pointer to `char`. If a field width precedes the c conversion specifier, then the number of characters specified by the field width is read. In this case, the corresponding argument must be a pointer to an array of `char`. If the optional character l (lowercase ell) precedes this conversion specifier, then the specifier expects a multibyte character in the input which is converted into a wide-character code. The corresponding argument must be a pointer to type `wchar_t`. If a field width also precedes the c conversion specifier, then the number of characters specified by the field width is read. In this case, the corresponding argument must be a pointer to an array of `wchar_t`.
	Wide-character	Expects a sequence of the number of characters specified in the optional field width; this is 1 if not specified. If no l (lowercase ell) precedes the `c` specifier, then the corresponding argument must be a pointer to an array of `char`. If an l (lowercase ell) precedes the `c` specifier, then the corresponding argument must be a pointer to an array of `wchar_t`.
C	Byte	The specifier expects a multibyte character in the input, which is converted into a wide-character code. The corresponding argument must be a pointer to type `wchar_t`. If a field width also precedes the `C` conversion specifier, then the number of characters specified by the field width is read. In this case, the corresponding argument must be a pointer to an array of `wchar_t`.
	Wide-character	Expects a sequence of the number of characters specified in the optional field width; this is 1 if not specified. The corresponding argument must be a pointer to an array of `wchar_t`.
s	Byte	Expects a sequences of bytes in the input. The corresponding argument must be a pointer to an array of characters that is large enough to contain the sequence and a terminating null character (\0) that is automatically added. The input field is terminated by a space, tab, or new-line character. If the optional character l (ell) precedes this conversion specifier, then the specifier expects a sequence of multibyte characters in the input, which are converted to wide-character codes. The corresponding argument must be a pointer to an array of wide characters (type `wchar_t`) that is large enough to contain the sequence plus the

Specifier	Input Type[1]	Description
		terminating null wide-character code that is automatically added. The input field is terminated by a space, tab, or new-line character.
	Wide-character	Expects (conceptually) a sequence of nonwhite-space characters in the input. If no l (lowercase ell) precedes the s specifier, then the corresponding argument must be a pointer to an array of `char` large enough to contain the sequence plus the terminating null byte that is automatically added. If an l (lowercase ell) precedes the s specifier, then the corresponding argument must be a pointer to an array of `wchar_t` large enough to contain the sequence plus the terminating null wide character that is automatically added.
S	Byte	The specifier expects a sequence of multibyte characters in the input, which are converted to wide-character codes. The corresponding argument must be a pointer to an array of wide characters (type `wchar_t`) that is large enough to contain the sequence plus a terminating null wide-character code that is added automatically. The input field is terminated by a space, tab, or new-line character.
	Wide-character	Expects a sequence of nonwhite-space characters in the input. The corresponding argument must be a pointer to an array of `wchar_t` large enough to contain the sequence plus the terminating null wide character that is automatically added.
e, f, g		Expects a floating-point number in the input. The corresponding argument must be a pointer to `float`. The input format for floating-point numbers is: [±]nnn[radix][ddd][{E |e}[±]nn]. The n's and d's are decimal digits (as many as indicated by the field width minus the signs and the letter E). The radix character is defined in the current locale.
[...]		Expects a nonempty sequence of characters that is not delimited by a white-space character. The brackets enclose a set of characters (the *scanset*) expected in the input sequence. Any character in the input sequence that does not match a character in the scanset terminates the character sequence. All characters between the brackets comprise the scanset, unless the first character after the left bracket is a circumflex (^). In this case, the scanset contains all characters other than those that appear between the circumflex and the right bracket. Any character that *does* appear between the circumflex and the right bracket will terminate the input character sequence. If the conversion specifier begins with [] or [^], then the right bracket character is in the scanset and the next right bracket character is the matching right bracket that ends the specification; otherwise, the first right bracket character ends the specification.
	Byte	If an l (lowercase ell) does not precede the `[` specifier, then the characters in the scanset must be single-byte characters only. In this case, the corresponding argument must be a pointer to an array of `char` large enough to accept the sequence and the terminating null byte that is automatically added.

Specifier	Input Type[1]	Description
		If an l (lowercase ell) does precede the [specifier, then the characters in the input sequence are considered to be multibyte characters, which are then converted to a wide-character sequence for further processing. If character ranges are specified in the scanset, then the processing is done according to the LC_COLLATE category of the current program's locale. In this case, the corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide character that is automatically added.
	Wide-character	If no l (lowercase ell) precedes the [conversion specifier, then processing is the same as described for the byte-input type of the %l[specifier, except that the corresponding argument must be an array of `char` large enough to accept the multibyte sequence plus the terminating null byte that is automatically added. If an l (lowercase ell) precedes the [conversion specifier, then processing is the same as in the preceding paragraph except that the corresponding argument must be an array of `wchar_t` large enough to accept the wide-character sequence plus the terminating null wide character that is automatically added.
p		Requires an argument that is a pointer to `void`. The input value is interpreted as a hexadecimal value.
n		No input is consumed. The corresponding argument is a pointer to an integer. The integer is assigned the number of characters read from the input stream so far by this call to the formatted input function. Execution of a %n directive does not increment the assignment count returned when the formatted input function completes execution.
%		Matches a single percent symbol. No conversion or assignment takes place. The complete conversion specification would be %%.

[1]Either *byte* or *wide-character*. Where neither is shown for a given specifier, the specifier description applies to both.

Remarks

- You can change the delimiters of the input field with the bracket ([ ]) conversion specification. Otherwise, an input field is defined as a string of nonwhite-space characters. It extends either to the next white-space character or until the field width, if specified, is exhausted. The function reads across line and record boundaries, since the new-line 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 VSI C permits only calls by value. To read a number in decimal format and assign its value to n, you must use the following form:

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

  You cannot use the following form:

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

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

  ```
  field = %x
  ```

  This format specification matches the following forms:

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

  These forms do not match the following example:

  ```
  fiel d=5218
  ```

2.4.2. Converting Output Information

The format specification string for the output of information can contain:

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

A conversion specification consists of the following, in the order listed:

- A percent character (%) or the sequence *%n*$.

  The sequence *%n*$ denotes that the conversion is applied to the *n*th output source listed, where *n* is a decimal integer between [1, NL_ARGMAX] (see the `<limits.h>` header file). For example, a conversion specification beginning with %5$ means that the conversion will be applied to the fifth output source listed after the format specification.

  If the conversion specification does not begin with the sequence *%n*$, the conversion specification is matched to its output source in left-to-right order. You should only use one type of conversion specification (% or %n$) in a format specification.

- One or more optional characters (see Table 2.4).
- A conversion specifier (see Table 2.5) concludes the conversion specification.

For examples of conversion specifications, see the sample programs in Section 2.6.

Table 2.4 shows the characters you can use between the percent sign (%) (or the sequence *%n*$) and the conversion specifier. These characters are optional, but if specified, they must occur in the order shown in Table 2.4.

Table 2.4. Optional Characters Between % (or *%n*$) and the Output Conversion Specifier

Character	Meaning	
flags	You can use the following flag characters, alone or in any combined order, to modify the conversion specification:	
	' (single quote)	Requests that a numeric conversion is formatted with the thousands separator character. Only the numbers to the left of the radix

Character	Meaning	
		character are formatted with the separator character. The character used as a separator and the positioning of the separators are defined in the program's current locale.
	– (hyphen)	Left-justifies the converted output source in its field.
	+	Requests that an explicit sign be present on a signed conversion. If this flag is not specified, the result of a signed conversion begins with a sign only when a negative value is converted.
	space	Prefixes a space to the result of a signed conversion, if the first character of the conversion is not a sign, or if the conversion results in no characters. If you specify both the *space* and the + flag, the *space* flag is ignored.
	#	Requests an alternate conversion format. Depending on the conversion specified, different actions will occur. For the `o` (octal) conversion, the precision is increased to force the first digit to be a zero. For the `x` (or `X`) conversion, a nonzero result is prefixed with 0x (or 0X). For `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result contains a decimal point even at the end of an integer value. For `g` and `G` conversions, trailing zeros are not trimmed. For other conversions, the effect of # is undefined.
	0	Uses zeros rather than spaces to pad the field width for `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `F`, `g`, and `G` conversions. If both the 0 and the – flags are specified, then the 0 flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior of the 0 flag is undefined.
field width	The minimum field width can be designated by a decimal integer constant, or by an output source. To specify an output source, use an asterisk (*) or the sequence *n\$, where *n* refers to the *n*th output source listed after the format specification. The minimum field width is considered after the conversion is done according to all the other components of the format directive. This component affects the padding of the conversion result as follows: If the result of the conversion is wider than the minimum field, write it out. If the result of the conversion is narrower than the minimum width, pad it to make up the field width. Pad with spaces by default. Pad with zeros if the 0 flag is specified; this does not mean that the width is an octal number. Padding is on the left by default, and on the right if a minus sign is specified.	

Character	Meaning
	For the wide-character output functions, the field width is measured in wide characters; for the byte output functions, it is measured in bytes. For programs compiled with /L_DOUBLE=64 (that is, compiled without the default /L_DOUBLE=128), the maximum field width is 1024.
. (period)	Separates the field width from the precision.
precision	The precision defines any of the following: • Minimum number of digits to appear for `d`, `i`, `o`, `u`, `x`, and `X` conversions • Number of digits to appear after the decimal-point character for `e`, `E`, and `f` conversions • Maximum number of significant digits for `g` and `G` conversions • Maximum number of characters to be written from a string in an `s` or `S` conversion If a precision appears with any other conversion specifier, the behavior is undefined. Precision can be designated by a decimal integer constant, or by an output source. To specify an output source, use an asterisk (*) or the sequence *n\$, where n refers to the nth output source listed after the format specification. If only the period is specified, the precision is taken as 0.
h	An h specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short int` or an `unsigned short int` argument. An h specifies that a following `n` conversion specifier applies to a pointer to a `short int` argument.
hh	An hh specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or an `unsigned char` argument.
j	A j specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `intmax_t` argument.
l	An l (lowercase ell) specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long int` or an `unsigned long int` argument. An l specifies that a following `n` conversion specifier applies to a pointer to a `long int` argument. An l specifies that a following `c` or `s` conversion specifier applies to a `wchar_t` argument.
L (or ll)	An L or ll (two lowercase ells) specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `__int64` or `unsigned __int64` argument. An L specifies that a following `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.
t	A t specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` argument.
z	A z specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` argument.

Remarks

- If an h, hh, j, l, L (or ll), t, or z appears with any other conversion specifier, the behavior is undefined.
- On OpenVMS Alpha systems, VSI C `int` values are equivalent to `long` values.

Table 2.5 describes the conversion specifiers for formatted output.

Table 2.5. Conversion Specifiers for Formatted Output

Specifier	Output Type[1]	Description
d, i		Converts an `int` argument to signed decimal format.
o		Converts an `unsigned int` argument to unsigned octal format.
u		Converts an `unsigned int` argument to unsigned decimal format (giving a number in the range 0 to 4,294,967,295).
x, X		Converts an `unsigned int` argument to unsigned hexadecimal format (with or without a leading 0x). The letters `abcdef` are used for `x` conversion, and the letters `ABCDEF` are used for `X` conversion.
f, F		Converts a `float` or `double` argument to the format [–]mmm.nnnnnn. The number of n's is equal to the precision specification as follows: • If no precision is specified, the default is 6. • If the precision is 0 and the # flag is specified, the decimal point appears but no n's appear. • If the precision is 0 and the # flag is not specified, the decimal point also does not appear. • If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.
e, E		Converts a `float` or `double` argument to the format [–]m.nnnnnn E ±xx. The number of n's is specified by the precision. If no precision is specified, the default is 6. If the precision is explicitly 0 and the # flag is specified, the decimal point appears but no n's appear. If the precision is explicitly 0 and the # flag is not specified, the decimal point also does not appear. An 'e' is printed for `e` conversion; an 'E' is printed for `E` conversion. The exponent always contains at least two digits. If the value is 0, the exponent is 0.
g, G		Converts a `float` or `double` argument to format `f` or `e` (or `E` if the `G` conversion specifier is used), with the precision specifying the number of significant digits. If the precision is 0, it is taken as 1. The format used depends on the value of the argument: format `e` (or `E`) is used only if the exponent resulting from such a conversion is less than –4, or is greater than or equal to the precision; otherwise, format f is used. Trailing zeros are suppressed in the fractional portion of the result. A decimal point appears only if it is followed by a digit.

Specifier	Output Type[1]	Description
c	Byte	Converts an `int` argument to an `unsigned char`, and writes the resulting byte. If the optional character l (lowercase ell) precedes this conversion specifier, then the specifier converts a `wchar_t` argument to an array of bytes representing the character, and writes the resulting character. If the field width is specified and the resulting character occupies fewer bytes than the field width, then it will be padded to the given width with space characters. If the precision is specified, then the behavior is undefined.
	Wide-character	If an l (lowercase ell) does not precede the `c` specifier, then the `int` argument is converted to a wide character as if by calling `btowc`, and the resulting character is written. If an l (lowercase ell) precedes the `c` specifier, then the specifier converts a `wchar_t` argument to an array of bytes representing the character, and writes the resulting character. If the field width is specified and the resulting character occupies fewer characters than the field width, it will be padded to the given width with space characters. If the precision is specified, the behavior is undefined.
C	Byte	Converts a `wchar_t` argument to an array of bytes representing the character, and writes the resulting character. If the field width is specified and the resulting character occupies fewer bytes than the field width, then it will be padded to the given width with space characters. If the precision is specified, then the behavior is undefined.
	Wide-character	Converts a `wchar_t` argument to an array of bytes representing the character, and writes the resulting character. If the field width is specified and the resulting character occupies fewer wide characters than the field width, then it will be padded to the given width with space characters. If the precision is specified, then the behavior is undefined.
s	Byte	Requires an argument that is a pointer to an array of characters of type `char`. The argument is used to write characters until a null character is encountered or until the number of characters indicated by the precision specification is exhausted. If the precision specification is 0 or omitted, then all characters up to a null are output. If the optional character l (lowercase ell) precedes this conversion specifier, then the specifier converts an array of wide-character codes to multibyte characters, and writes the multibyte characters. Requires an argument that is a pointer to an array of wide characters of type `wchar_t`. Characters are written until a null wide character is encountered or until the number of bytes indicated by the precision specification is exhausted. If the precision specification is omitted or is greater than the size of the array of converted bytes, then the array of wide characters must be terminated by a null wide character.
	Wide-character	If an l (lowercase ell) does not precede the `s` specifier, then the specifier converts an array of multibyte characters, as if by calling `mbrtowc` for each multibyte character, and writes the resulting characters until a null wide character is encountered or the number of wide characters

Specifier	Output Type[1]	Description
		indicated by the precision specification is exhausted. If the precision specification is omitted or is greater than the size of the array of converted characters, then the converted array must be terminated by a null wide character. If an l precedes this conversion specifier, then the argument is a pointer to an array of `wchar_t`. Characters from this array are written until a null wide character is encountered or the number of wide characters indicated by the precision specification is exhausted. If the precision specification is omitted or is greater than the size of the array, then the array must be terminated by a null wide character.
S	Byte	Converts an array of wide-character codes to multibyte characters, and writes the multibyte characters. Requires an argument that is a pointer to an array of wide characters of type `wchar_t`. Characters are written until a null wide character is encountered or until the number of bytes indicated by the precision specification is exhausted. If the precision specification is omitted or is greater than the size of the array of converted bytes, then the array of wide characters must be terminated by a null wide character.
	Wide-character	The argument is a pointer to an array of `wchar_t`. Characters from this array are written until a null wide character is encountered or the number of wide characters indicated by the precision specification is exhausted. If the precision specification is omitted or is greater than the size of the array, then the array must be terminated by a null wide character.
p		Requires an argument that is a pointer to `void`. The value of the pointer is output as a hexadecimal number.
n		Requires an argument that is a pointer to an integer. The integer is assigned the number of characters written to the output stream so far by this call to the formatted output function. No argument is converted.
%		Writes out the percent symbol. No conversion is performed. The complete conversion specification would be %%.

[1]Either *byte* or *wide-character*. Where neither is shown for a given specifier, the specifier description applies to both.

2.5. Terminal I/O

VSI C defines three file pointers that allow you to perform I/O to and from the logical devices usually associated with your terminal (for interactive jobs) or a batch stream (for batch jobs). In the OpenVMS environment, the three permanent process files SYS$INPUT, SYS$OUTPUT, and SYS$ERROR perform the same functions for both interactive and batch jobs. Terminal I/O refers to both terminal and batch stream I/O. The file pointers `stdin`, `stdout`, and `stderr` are defined when you include the `<stdio.h>` header file using the `#include` preprocessor directive.

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

There are three file descriptors that refer to the terminal. The file descriptor 0 is equivalent to SYS$INPUT, 1 is equivalent to SYS$OUTPUT, and 2 is equivalent to SYS$ERROR.

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

2.6. Program Examples

This section gives some program examples that show how the I/O functions can be used in applications.

Example 2.1 shows the `printf` function.

Example 2.1. Output of the Conversion Specifications

```
/*      CHAP_2_OUT_CONV.C                                   */

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

/* Include the proper header files in case printf has   */
/* to return EOF.                                       */

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

#define WIDE_STR_SIZE 20

main()
{
    double val = 123345.5;
    char c = 'C';
    int i = -1500000000;
    char *s = "thomasina";
    wchar_t wc;
    wchar_t ws[WIDE_STR_SIZE];

    /* Produce a wide character and a wide character string */

    if (mbtowc(&wc, "W", 1) == -1) {
        perror("mbtowc");
        exit(EXIT_FAILURE);
    }

    if (mbstowcs(ws, "THOMASINA", WIDE_STR_SIZE) == -1) {
        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

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

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

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

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

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

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

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

Running Example 2.1 produces the following output:

```
$ RUN  EXAMPLE
%9.4f:          <123345.5000>
%9f:            <123345.500000>
%9.0f:          <   123346>
%-9.0f:         <123346   >
%11.6e:         <1.233455e+05>
%11e:           <1.233455e+05>
%11.0e:         <      1e+05>
%-11.0e:        <1e+05      >
%11g:           <     123346>
%9g:            <   123346>
%d:             <67>
%c:             <C>
%o:             <103>
%x:             <43>
%d:             <-1500000000>
%u:             <2794967296>
%x:             <a697d100>
%s:             <thomasina>
%-9.6s:         <thomas   >
%-*.*s:         <thoma    >
%6.0s:          <      >
```

```
%C:             <W>
%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

```
/*      CHAP_2_STDIO.C  */

/* 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>

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);
    }

    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);
    }

    /*  Read 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.     */
}
```

Running Example 2.2 produces the following result:

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

The 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
```

Example 2.3. Using Wide Character I/O Functions

```
/*      CHAP_2_WC_IO.C                                          */

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

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

main()
{
    char flat_buffer[32];
    wchar_t wide_buffer[32];
    wchar_t format[32];
    int i,
        pos;
    FILE *fptr;

    /*  Set file pointer.            */
    fptr = fopen("data.dat", "w+");
    if (fptr == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < 5; i++) {
        if (i == 2)     /*  Get position of record 2. */
            pos = ftell(fptr);
        /*  Print a line to the buffer.  */
        sprintf(flat_buffer, "test data line %d\n", i);
        if (mbstowcs(wide_buffer, flat_buffer, 32) == -1) {
            perror("mbstowcs");
            exit(EXIT_FAILURE);
        }

        /*  Print buffer to the record.  */
        fputws(wide_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 (fgetws(wide_buffer, 32, fptr) == NULL) {
        perror("fgetws");       /*  Exit on fgets error. */
        exit(EXIT_FAILURE);
    }
    /*  Print the buffer. */
    printf("Data in record 2 is: %S", wide_buffer);
    fclose(fptr);               /*  Close the file.  */
}
```

Running Example 2.3 produces the following result:

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

The output to DATA.DAT from Example 2.3 is:

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

Example 2.4 shows the use of both a file pointer and a file descriptor to access a single file.

Example 2.4. I/O Using File Descriptors and Pointers

```
/*      CHAP_2_FILE_DIS_AND_POINTER.C                          */

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

#include <unixio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define ERROR 0
#define ERROR1 -1
#define BUFFSIZE 132

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

    if ((fildes = creat("data.dat", 0, "rat=cr",
```

```
                        "rfm=var")) == ERROR1) {
        perror("DATA.DAT: creat() failed\n");
        exit(EXIT_FAILURE);
    }

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

    if (fclose(fp) == EOF) {
        perror("DATA.DAT: fclose() failed\n");
        exit(EXIT_FAILURE);
    }
}
```

Chapter 3. Character, String, and Argument-List Functions

Table 3.1 describes the character, string, and argument-list functions in the VSI C Run-Time Library (C RTL). Although further discussion follows in this chapter, see the Reference Section for more detailed information on each function.

Table 3.1. Character, String, and Argument-List Functions

Function	Description
Character Classification	
`isalnum` `iswalnum`	Return a nonzero integer if its argument is one of the alphanumeric characters in the current locale.
`isalpha` `iswalpha`	Return a nonzero integer if its argument is one of the alphabetic characters in the current locale.
`isascii`	Returns a nonzero integer if its argument is any ASCII character.
`iscntrl` `iswcntrl`	Return a nonzero integer if its argument is a control character in the current locale.
`isdigit` `iswdigit`	Return a nonzero integer if its argument is a digit character in the current locale.
`isgraph` `iswgraph`	Return a nonzero integer if its argument is a graphic character in the current locale.
`islower` `iswlower`	Return a nonzero integer if its argument is a lowercase character in the current locale.
`isprint` `iswprint`	Return a nonzero integer if its argument is a printing character in the current locale.
`ispunct` `iswpunct`	Return a nonzero integer if its argument is a punctuation character in the current locale.
`isspace` `iswspace`	Return a nonzero integer if its argument is a white-space character in the current locale.
`isupper` `iswupper`	Return a nonzero integer if its argument is an uppercase character in the current locale.
`iswctype`	Returns a nonzero integer if its argument has the specified property.
`isxdigit` `iswxdigit`	Return a nonzero integer if its argument is a hexadecimal digit (0 to 9, A to F, or a to f).
Character Conversion	
`btowc`	Converts a one-byte multibyte character to a wide character in the initial shift state.

`ecvt`
`fcvt`
`gcvt`
	Convert an argument to a null-terminated string of ASCII digits and return the address of the string.

`index` `rindex`	Search for a character in a string.
`mblen`	Determine the number of bytes in a multibyte character.

Function	Description
mbrlen	
mbsinit	Determines whether an mbstate_t object describes an initial conversion state.
mbstowcs	Converts a sequence of multibyte characters into a sequence of corresponding codes.
toascii	Converts its argument, an 8-bit ASCII character, to a 7-bit ASCII character.
tolower _tolower towlower	Convert its argument, an uppercase character, to lowercase.
toupper _toupper towupper	Convert its argument, a lowercase character, to uppercase.
towctrans	Maps one wide character to another according to a specified mapping descriptor.
wcstombs	Converts a sequence of wide-character codes corresponding to multibyte characters to a sequence of multibyte characters.
wctob	Determines if a wide character corresponds to a single-byte multibyte character and returns its multibyte character representation.
wctomb	Converts a wide character to its multibyte character representation.
wctrans	Returns the description of a mapping, corresponding to specified property, that can be later used in a call to towctrans.
wctype	Converts a valid character class defined for the current locale to an object of type wctype_t.
String Manipulation	
atof	Converts a given string to a double-precision number.
atoi atol	Convert a given string of ASCII characters to the appropriate numeric values.
atoll atoq	Convert a given string of ASCII characters to an __int64.
basename	Return the last component of a path name.
dirname	Report the parent directory name of a file path name.
strcat strncat wcscat wcsncat	Append one string to the end of another string.
strchr strrchr wcschr wcsrchr	Return the address of the first or last occurrence of a given character in a null-terminated string.
strcmp strncmp strcoll wcscmp	Compare 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.

Function	Description
`wcsncmp` `wcscoll`	

`strcpy`
`strncpy`
`wcscpy`
`wcsncpy`
	Copies all or part of one string into another.

`strxfrm` `wcsxfrm`	Transform a multibyte string to another string ready for comparisons using the `strcmp` or `wcscmp` function.
`strcspn` `wcscspn`	Search a string for a character that is in a specified set of characters.
`strlen` `wcslen`	Return the length of a string of characters. The returned length does not include the terminating null character (\0).
`strpbrk` `wcspbrk`	Search a string for the occurrence of one of a specified set of characters.
`strspn` `wcsspn`	Search a string for the occurrence of a character that is not in a specified set of characters.
`strstr` `wcswcs`	Search a string for the first occurrence of a specified set of characters.
`strtod` `wcstod`	Convert a given string to a double-precision number.
`strtok` `wcstok`	Locate text tokens in a given string.
`strtol` `wcstol`	Convert the initial portion of a string to a signed long integer.
`strtoll` `strtoq`	Convert the initial portion of a string to signed `__int64`.
`strtoul` `wcstoul`	Convert the initial portion of a string to an unsigned long integer.
`strtoull` `strtouq`	Convert the initial portion of the string pointed to by the pointer to the character string to an unsigned `__int64`.
String Handling—Accessing Binary Data	
`bcmp`	Compares byte strings.
`bcopy`	Copies byte strings.
`bzero`	Copies nulls into byte strings.
`memchr` `wmemchr`	Locate the first occurrence of the specified byte within the initial length of the object to be searched.
`memcmp` `wmemcmp`	Compare two objects byte by byte.

`memcpy`
`memmove`
`mempcpy`
`wmemcpy`
`wmemmove`
	Copy a specified number of bytes from one object to another.

`memset`	Set a specified number of bytes in a given object to a given value.

Function	Description
wmemset	
Argument-List Handling—Accessing a Variable-Length Argument List	
va_arg	Returns the next item in the argument list.
va_copy	Copies one argument list to another.
va_count	Returns the number of quadwords *(Alpha only)* in the argument list.
va_end	Finishes the va_start session.
va_start va_start_1	Initialize a variable to the beginning of the argument list.
vfprintf vprintf vsprintf	Print formatted output based on an argument list.

3.1. Character-Classification Functions

The character-classification functions take a single argument on which they perform a logical operation. The argument can have any value; it does not have to be an ASCII character. The isascii function determines if the argument is an ASCII character (0 through 177 octal). The other functions determine whether the argument is a particular type of ASCII character, such as a graphic character or digit. The isw* functions test wide characters. Character-classification information is in the LC_CTYPE category of the program's current locale.

For all functions, a positive return value indicates TRUE. A return value of0 indicates FALSE.

To briefly reference the character-classification functions in a subsequent table, each function is assigned a number, as shown in Table 3.2.

Table 3.2. Character-Classification Functions

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

Table 3.3 lists the numbers of the functions (as assigned in Table 3.2) that return the value TRUE for each of the given ASCII characters. The numeric code represents the octal value of each of the ASCII characters.

Table 3.3. ASCII Characters and the Character-Classification Functions

ASCII Values	Function Numbers	ASCII Values	Function Numbers
NUL 00	3,4	@ 100	3,6,8,9
SOH 01	3,4	A 101	1,2,3,6,8,11,12
STX 02	3,4	B 102	1,2,3,6,8,11,12
ETX 03	3,4	C 103	1,2,3,6,8,11,12

ASCII Values	Function Numbers	ASCII Values	Function Numbers
EOT 04	3,4	D 104	1,2,3,6,8,11,12
ENQ 05	3,4	E 105	1,2,3,6,8,11,12
ACK 06	3,4	F 106	1,2,3,6,8,11,12
BEL 07	3,4	G 107	1,2,3,6,8,11
BS 10	3,4	H 110	1,2,3,6,8,11
HT 11	3,4,10	I 111	1,2,3,6,8,11
LF 12	3,4,10	J 112	1,2,3,6,8,11
VT 13	3,4,10	K 113	1,2,3,6,8,11
FF 14	3,4,10	L 114	1,2,3,6,8,11
CR 15	3,4,10	M 115	1,2,3,6,8,11
SO 16	3,4	N 116	1,2,3,6,8,11
SI 17	3,4	O 117	1,2,3,6,8,11
DLE 20	3,4	P 120	1,2,3,6,8,11
DC1 21	3,4	Q 121	1,2,3,6,8,11
DC2 22	3,4	R 122	1,2,3,6,8,11
DC3 23	3,4	S 123	1,2,3,6,8,11
DC4 24	3,4	T 124	1,2,3,6,8,11
NAK 25	3,4	U 125	1,2,3,6,8,11
SYN 26	3,4	V 126	1,2,3,6,8,11
ETB 27	3,4	W 127	1,2,3,6,8,11
CAN 30	3,4	X 130	1,2,3,6,8,11
EM 31	3,4	Y 131	1,2,3,6,8,11
SUB 32	3,4	Z 132	1,2,3,6,8,11
ESC 33	3,4	[133	3,6,8,9
FS 34	3,4	\ 134	3,6,8,9
GS 35	3,4	] 135	3,6,8,9
RS 36	3,4	^ 136	3,6,8,9
US 37	3,4	– 137	3,6,8,9
SP 40	3,8,10	` 140	3,6,8,9
! 41	3,6,8,9	a 141	1,2,3,6,7,8,12
" 42	3,6,8,9	b 142	1,2,3,6,7,8,12
# 43	3,6,8,9	c 143	1,2,3,6,7,8,12
$ 44	3,6,8,9	d 144	1,2,3,6,7,8,12
% 45	3,6,8,9	e 145	1,2,3,6,7,8,12
& 46	3,6,8,9	f 146	1,2,3,6,7,8,12
’ 47	3,6,8,9	g 147	1,2,3,6,7,8
(50	3,6,8,9	h 150	1,2,3,6,7,8
) 51	3,6,8,9	i 151	1,2,3,6,7,8

ASCII Values	Function Numbers	ASCII Values	Function Numbers
* 52	3,6,8,9	j 152	1,2,3,6,7,8
+ 53	3,6,8,9	k 153	1,2,3,6,7,8
' 54	3,6,8,9	l 154	1,2,3,6,7,8
- 55	3,6,8,9	m 155	1,2,3,6,7,8
. 56	3,6,8,9	n 156	1,2,3,6,7,8
/ 57	3,6,8,9	o 157	1,2,3,6,7,8
0 60	1,3,5,6,8,12	p 160	1,2,3,6,7,8
1 61	1,3,5,6,8,12	q 161	1,2,3,6,7,8
2 62	1,3,5,6,8,12	r 162	1,2,3,6,7,8
3 63	1,3,5,6,8,12	s 163	1,2,3,6,7,8
4 64	1,3,5,6,8,12	t 164	1,2,3,6,7,8
5 65	1,3,5,6,8,12	u 165	1,2,3,6,7,8
6 66	1,3,5,6,8,12	v 166	1,2,3,6,7,8
7 67	1,3,5,6,8,12	w 167	1,2,3,6,7,8
8 70	1,3,5,6,8,12	x 170	1,2,3,5,6,8
9 71	1,3,5,6,8,12	y 171	1,2,3,5,6,8
: 72	3,6,8,9	z 172	1,2,3,5,6,8
; 73	3,6,8,9	{ 173	3,6,8,9
< 74	3,6,8,9	| 174	3,6,8,9
= 75	3,6,8,9	} 175	3,6,8,9
> 76	3,6,8,9	~ 176	3,6,8,9
? 77	3,6,8,9	DEL 177	3,4

Example 3.1 shows how the character-classification functions are used.

Example 3.1. Character-Classification Functions

```
/*       CHAP_3_CHARCLASS.C                                          */

/* This example uses the isalpha, isdigit, and isspace              */
/* functions 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>

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

    while ((c = getchar()) != EOF) {
```

```
        if (isalpha(c))
            i++;
        if (isdigit(c))
            j++;
        if (isspace(c))
            k++;
    }

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

The sample input and output from Example 3.1 follows:

```
$ RUN  EXAMPLE1
I saw 35 people riding bicycles on Main Street.Return
Ctrl/Z
Number of letters: 36
Number of digits:  2
Number of spaces:  8
$
```

3.2. Character-Conversion Functions

The character-conversion functions convert one type of character to another type. These functions include:

```
ecvt          _tolower
fcvt          toupper
gcvt          _toupper
mbtowc        towctrans
mbrtowc       wctrans
mbsrtowcs     wcrtomb
toascii       wcsrtombs
tolower
```

For more information on each of these functions, see the Reference Section.

Example 3.2 shows how to use the `ecvt` function.

Example 3.2. Converting Double Values to an ASCII String

```
/*     CHAP_3_CHARCONV.C                                          */

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

#include <stdio.h>
#include <stdlib.h>
#include <unixlib.h>
#include <string.h>

main()
{

    double val;         /* Value to be converted */
```

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

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

    val = -3.1297830e-10;

    printf("original value: %e\n", val);
    if (sign)
        printf("value is negative\n");
    else
        printf("value is positive\n");
    printf("decimal point at %d\n", point);
}
```

The output from Example 3.2 is as follows:

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

Example 3.3 shows how to use the `toupper` and `tolower` functions.

Example 3.3. Changing Characters to and from Uppercase Letters

```
/*     CHAP_3_CONV_UPPERLOWER.C                                  */

/* 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 */
#include <stdlib.h>

main()
{
    char c,
         ch;

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

Sample input and output from Example 3.3 are as follows:

```
$ RUN  EXAMPLE3
LET'S GO TO THE welcome INN.Ctrl/Z
```

```
let's go to the WELCOME inn.
$
```

3.3. String and Argument-List Functions

The C RTL contains a group of functions that manipulate strings. Some of these functions concatenate strings; others search a string for specific characters or perform some other comparison, such as determining the equality of two strings.

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

The set of functions defined and declared in the `<varargs.h>` and the `<stdarg.h>` header files provide a method of accessing variable-length argument lists. The `<stdarg.h>` functions are defined by the ANSI C Standard and are more portable than those defined in `<varargs.h>`.

The C RTL functions such as `printf` and `execl`, for example, use variable-length argument lists. User-defined functions with variable-length argument lists that do not use `<varargs.h>` or `<stdarg.h>` are not portable due to the different argument-passing conventions of various machines.

The `<stdarg.h>` header file does not contain `va_alist` and `va_dcl`. The following shows a syntax example when using `<stdarg.h>`:

```
function_name(int arg1, ...)
{
va_list ap;
. . .
```

When using `<varargs.h>`:

- The identifier `va_alist` is a parameter in the function definition.
- `va_dcl` declares the parameter `va_alist`, a declaration that is not terminated with a semicolon (;).
- The type `va_list` is used in the declaration of the variable used to traverse the list. You must declare at least one variable of type `va_list` when using `<varargs.h>`.

These names and declarations have the following syntax:

```
function_name(int arg1, ...)
{
va_list ap;
.
.
.
```

3.4. Program Examples

Example 3.4 shows how to use the `strcat` and `strncat` functions.

Example 3.4. Concatenating Two Strings

```
/*        CHAP_3_CONCAT.C                                        */

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

```
/*  strings.                                                    */

#include <stdio.h>
#include <string.h>

main()
{
    static char string1[80] = "Concatenates ";
    static char string2[] = "two strings ";
    static char string3[] = "up to a maximum of characters.";
    static char string4[] = "imum number of characters";

    printf("strcat:\t%s\n", strcat(string1, string2));
    printf("strncat ( 0):\t%s\n", strncat(string1, string3, 0));
    printf("strncat (11):\t%s\n", strncat(string1, string3, 11));
    printf("strncat (40):\t%s\n", strncat(string1, string4, 40));
}
```

Example 3.4 produces the following output:

```
$ RUN  EXAMPLE1
strcat: Concatenates two strings
strncat ( 0): Concatenates two strings
strncat (11): Concatenates two strings up to a max
strncat (40): Concatenates two strings up to a maximum number of
 characters.
$
```

Example 3.5 shows how to use the `strcspn` function.

Example 3.5. Four Arguments to the strcspn Function

```
/*        CHAP_3_STRCSPN.C                                      */

/*  This example shows how strcspn interprets four              */
/*  different kinds of arguments.                               */

#include <stdio.h>

main()
{

    printf("strcspn with null charset: %d\n",
            strcspn("abcdef", ""));

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

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

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

The sample output, to the file `strcspn.out`, in Example 3.5 is as follows:

```
$ RUN  EXAMPLE2 strcspn with null charset:  6
```

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

Example 3.6 shows how to use the `<stdarg.h>` functions and definitions.

Example 3.6. Using the <stdarg.h> Functions and Definitions

```
/*        CHAP_3_STDARG.C                                          */

/* This routine accepts a variable number of string arguments,  */
/* preceded by a count of the number of such strings. It        */
/* allocates enough space in which to concatenate all of the    */
/* strings, concatenates them together, and returns the address */
/* of the new string. It returns NULL if there are no string    */
/* arguments, or if they are all null strings.                  */

#include <stdarg.h>       /* Include appropriate header files   */
#include <stdlib.h>       /* for the "example" call in main.    */
#include <string.h>
#include <stdio.h>

/* NSTRINGS is the maximum number of string arguments accepted  */
/* (arbitrary).                                                 */

#define NSTRINGS 10

char *concatenate(int n,...)
{
    va_list ap;         /* Declare the argument pointer. */

    char *list[NSTRINGS],
        *string;
    int index = 0,
        size = 0;

    /* Check that the number of arguments is within range.   */

    if (n <= 0)
        return NULL;
    if (n > NSTRINGS)
        n = NSTRINGS;

    va_start(ap, n);    /* Initialize the argument pointer.  */

    do {
        /* Extract the next argument and save it. */

        list[index] = va_arg(ap, char *);

        size += strlen(list[index]);
    } while (++index < n);

    va_end(ap); /* Terminate use of ap. */

    if (size == 0)
        return NULL;
```

```
    string = malloc(size + 1);
    string[0] = '\0';

    /* Append each argument to the end of the growing result    */
    /*  string.                                                 */

    for (index = 0; index < n; ++index)
        strcat(string, list[index]);

    return string;
}

/* An example of calling this routine is */

main() {
    char *ret_string ;

    ret_string = concatenate(7, "This ", "message ", "is ",
                               "built with ", "a", " variable arg",
                               " list.") ;

    puts(ret_string) ;
}
```

The call to Example 3.6 produces the following output:

```
This message is built with a variable arg list.
```

Chapter 4. Error and Signal Handling

Table 4.1 lists and describes all the error- and signal-handling functions found in the VSI C Run-Time Library (C RTL). For more detailed information on each function, see the Reference Section.

Table 4.1. Error- and Signal-Handling Functions

Function	Description
`abort`	Raises the signal SIGABRT that terminates the execution of the program.
`assert`	Puts diagnostics into programs.
`atexit`	Registers a function to be called at program termination.
`exit, _exit`	Terminates the current program.
`perror`	Writes a short error message to `stderr` describing the current `errno` value.
`strerror`	Maps the error code in `errno` to an error message string.
`alarm`	Sends the signal SIGALARM to the invoking process after the number of seconds indicated by its argument has elapsed.
`gsignal`	Generates a specified software signal.
`kill`	Sends a SIGKILL signal to the process specified by a process ID.
`longjmp`	Transfers control from a nested series of function invocations back to a predefined point without returning normally.
`pause`	Causes the process to wait until it receives a signal.
`raise`	Generates a specified signal.
`setjmp`	Establishes the context for a later transfer of control from a nested series of function invocations, without returning normally.
`sigaction`	Specifies the action to take upon delivery of a signal.
`sigaddset`	Adds the specified individual signal.
`sigblock`	Causes the signals in its argument to be added to the current set of signals being blocked from delivery.
`sigdelset`	Deletes a specified individual signal.
`sigemptyset`	Initializes the signal set to exclude all signals.
`sigfillset`	Initializes the signal set to include all signals.
`sighold`	Adds the specified signal to the calling process's signal mask.
`sigignore`	Sets the disposition of the specified signal to SIG_IGN.
`sigismember`	Tests whether a specified signal is a member of the signal set.
`siglongjmp`	Nonlocal go to with signal handling.
`sigmask`	Constructs the mask for a given signal number.
`signal`	Catches or ignores a signal.
`sigpause`	Blocks a specified set of signals and then waits for a signal that was not blocked.
`sigpending`	Examines pending signals.
`sigprocmask`	Sets the current signal mask.

Function	Description
sigrelse	Removes the specified signal from the calling process's signal mask.
sigsetjmp	Sets the jump point for a nonlocal go to.
sigsetmask	Establishes the signals that are blocked from delivery.
sigsuspend	Atomically changes the set of blocked signals and waits for a signal.
sigtimedwait	Suspends a calling thread and waits for queued signals to arrive.
sigvec	Permanently assigns a handler for a specific signal.
sigwait	Suspends a calling thread and waits for queued signals to arrive.
sigwaitinfo	Suspends a calling thread and waits for queued signals to arrive.
ssignal	Allows you to specify the action to be taken when a particular signal is raised.
VAXC$ESTABLISH	Establishes an application exception handler in a way that is compatible with C RTL exception handling.

4.1. Error Handling

When an error occurs during a call to any of the C RTL functions, the function returns an unsuccessful status. Many RTL routines also set the external variable `errno` to a value that indicates the reason for the failure. You should always check the return value for an error situation.

The `<errno.h>` header file declares `errno` and symbolically defines the possible error codes. By including the `<errno.h>` header file in your program, you can check for specific error codes after a C RTL function call.

At program startup, the value of `errno` is 0. The value of `errno` can be set to a nonzero value by many C RTL functions. It is not reset to 0 by any C RTL function, so it is only valid to use `errno` after a C RTL function call has been made and a failure status returned. Table 4.2 lists the symbolic values that may be assigned to `errno` by the C RTL.

Table 4.2. The Error Code Symbolic Values

Symbolic Constant	Description
E2BIG	Argument list too long
EACCES	Permission denied
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Can't assign requested address
EAFNOSUPPORT	Address family not supported
EAGAIN	No more processes
EALIGN	Alignment error
EALREADY	Operation already in progress
EBADF	Bad file number
EBADCAT	Bad message catalog format
EBADMSG	Corrupted message detected
EBUSY	Mount device busy
ECANCELED	Operation canceled
ECHILD	No children

Symbolic Constant	Description
ECONNABORTED	Software caused connection abort
ECONNREFUSED	Connection refused
ECONNRESET	Connection reset by peer
EDEADLK	Resource deadlock avoided
EDESTADDRREQ	Destination address required
EDOM	Math argument
EDQUOT	Disk quota exceeded
EEXIST	File exists
EFAIL	Cannot start operation
EFAULT	Bad address
EFBIG	File too large
EFTYPE	Inappropriate operation for file type
EHOSTDOWN	Host is down
EHOSTUNREACH	No route to host
EIDRM	Identifier removed
EILSEQ	Illegal byte sequence
EINPROGRESS	Operation now in progress
EINPROG	Asynchronous operation in progress
EINTR	Interrupted system call
EINVAL	Invalid argument
EIO	I/O error
EISCONN	Socket is already connected
EISDIR	Is a directory
ELOOP	Too many levels of symbolic links
EMFILE	Too many open files
EMLINK	Too many links
EMSGSIZE	Message too long
ENAMETOOLONG	Filename too long
ENETDOWN	Network is down
ENETRESET	Network dropped connection on reset
ENETUNREACH	Network is unreachable
ENFILE	File table overflow
ENOBUFS	No buffer space available
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOLCK	No locks available
ENOMEM	Not enough core

Symbolic Constant	Description
ENOMSG	No message of desired type
ENOPROTOOPT	Protocol not available
ENOSPC	No space left on device
ENOSYS	Function not implemented
ENOTBLK	Block device required
ENOTCONN	Socket is not connected
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTSOCK	Socket operation on nonsocket
ENOTSUP	Function not implemented
ENOTTY	Not a typewriter
ENWAIT	No waiting processes
ENXIO	No such device or address
EOPNOTSUPP	Operation not supported on socket
EPERM	Not owner
EPFNOSUPPORT	Protocol family not supported
EPIPE	Broken pipe
EPROCLIM	Too many processes
EPROTONOSUPPORT	Protocol not supported
EPROTOTYPE	Protocol wrong type for socket
ERANGE	Result too large
EREMOTE	Too many levels of remote in path
EROFS	Read-only file system
ESHUTDOWN	Can't send after socket shutdown
ESOCKTNOSUPPORT	Socket type not supported
ESPIPE	Illegal seek
ESRCH	No such process
ESTALE	Stale NFS file handle
ETIMEDOUT	Connection timed out
ETOOMANYREFS	Too many references: can't splice
ETXTBSY	Text file busy
EUSERS	Too many users
EVMSERR	OpenVMS specific non-translatable error code
EWOULDBLOCK	I/O operation would block channel
EXDEV	Cross-device link

You can translate the error codes to a message, similar to that found in UNIX systems, by using the `perror` or `strerror` function. If `errno` is set to EVMSERR, `perror` cannot translate the error

code and prints the following message, followed by the OpenVMS error message associated with the value:

```
%s:nontranslatable vms error code: xxxxxx vms message:
```

In the message, %s is the string you supply to `perror` ; xxxxxx is the OpenVMS condition value.

If `errno` is set to EVMSERR, then the OpenVMS condition value is available in the `vaxc$errno` variable declared in the `<errno.h>` header file. The `vaxc$errno` variable is guaranteed to have a valid value only if `errno` is set to EVMSERR; if `errno` is set to a value other than EVMSERR, the value of `vaxc$errno` is undefined.

See the `strerror` function in the Reference Section for another way to translate error codes.

4.2. Signal Handling

A signal is a form of software interrupt to the normal execution of a user process. Signals occur as a result of a variety of events, including any of the following:

- Typing Ctrl/C at a terminal
- Certain programming errors
- A call to the `gsignal` or `raise` function
- A wake-up action

4.2.1. OpenVMS Versus UNIX Terminology

Both OpenVMS and UNIX systems provide signal-handling mechanisms that behave differently but use similar terminology. With the C RTL, you can program using either signal-handling mechanism. Before describing the signal-handling routines, some terminology must be established.

The UNIX term for a software interrupt is *signal*. A routine called by the UNIX system to process a signal is termed a *signal handler*.

A software interrupt on an OpenVMS system is referred to as a *signal*, *condition*, or *exception*. A routine called by the OpenVMS system to process software interrupts is termed a *signal handler*, *condition handler*, or *exception handler*.

To prevent confusion, the terms *signal* and *signal handler* in this manual refer to UNIX interrupts and interrupt processing routines, while the terms *exception* and *exception handler* refer to OpenVMS interrupts and interrupt processing routines.

4.2.2. UNIX Signals and the C RTL

Signals are represented by mnemonics defined in the `<signal.h>` header file. Table 4.3 lists the supported signal mnemonics and the corresponding event that causes each signal to be generated on the OpenVMS operating system.

Table 4.3. C RTL Signals

Name	Description	Generated by
SIGABRT[1]	Abort	`abort`()

Name	Description	Generated by
SIGALRM	Alarm clock	Timer AST, `alarm` routine
SIGBUS	Bus error	Access violation or change mode user
SIGCHLD	Child process stopped	Child process terminated or stopped
SIGEMT	EMT instruction	Compatibility mode trap or opcode reserved to customer
SIGFPE	Floating-point exception	Floating-point overflow/underflow
SIGHUP	Hang up	Data set hang up
SIGILL[1]	Illegal instruction	Illegal instruction, reserved operand, or reserved address mode
SIGINT[4]	Interrupt	OpenVMS Ctrl/C interrupt
SIGIOT[1]	IOT instruction	Opcode reserved to customer
SIGKILL[2 3]	Kill	External signal only
SIGQUIT[5]	Quit	Not implemented.
SIGPIPE	Broken pipe	Write to a pipe with no readers.
SIGSEGV	Segment violation	Length violation or change mode user
SIGSYS	System call error	Bad argument to system call
SIGTERM	Software terminate	External signal only
SIGTRAP[1]	Trace trap	TBIT trace trap or breakpoint fault instruction
SIGUSR1	User-defined signal	Explicit program call to raise the signal
SIGUSR2	User-defined signal	Explicit program call to raise the signal
SIGWINCH[6]	Window size changed	Explicit program call to raise the signal

[1]Cannot be reset when intercepted.

[4]Setting SIGINT can affect processing of Ctrl/Y interrupts. For example, in response to a caller's request to block or ignore SIGINT, the C RTL disables the Ctrl/Y interrupt.

[2]Cannot be intercepted or ignored.

[3]Cannot be blocked.

[5]"Not implemented" for SIGQUIT means that there is no external event, including a Ctrl/Y interrupt, that would trigger a SIGQUIT signal, thereby causing a signal handler established for SIGQUIT to be invoked. This signal can be generated only through an appropriate C RTL function, such as raise.

[6]Supported on OpenVMS Version 7.3 and higher.

By default, when a signal (except for SIGCHLD) occurs, the process is terminated. However, you can choose to have the signal ignored by using one of the following functions:

```
sigaction
signal
sigvec
ssignal
```

You can have the signal blocked by using one of the following functions:

```
sigblock
sigsetmask
sigprocmask
sigsuspend
```

`sigpause`

Table 4.3 indicates those signals that cannot be ignored or blocked.

You can also establish a signal handler to catch and process a signal by using one of the following functions:

`sigaction`
`signal`
`sigvec`
`ssignal`

Unless noted in Table 4.3, each signal can be reset. A signal is reset if the signal handler function calls `signal` or `ssignal` to re-establish itself to catch the signal. Example 4.1 shows how to establish a signal handler and reset the signal.

The calling interface to a signal handler is:

```
void handler (int sigint);
```

Where *sigint* is the signal number of the raised signal that caused this handler to be called.

A signal handler installed with `sigvec` remains installed until it is changed.

A signal handler installed with `signal` or `signal` remains installed until the signal is generated.

A signal handler can be installed for more than one signal. Use the `sigaction` routine with the SA_RESETHAND flag to control this.

4.2.3. Signal-Handling Concepts

A signal is said to be *generated* for (or sent to) a process when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, and terminal activity, as well as the invocation of `kill`. In some circumstances, the same event generates signals for multiple processes.

Each process has an action to be taken in response to each signal defined by the system. A signal is said to be *delivered* to a process when the appropriate action for the process and signal is taken.

During the time between the generation of a signal and its delivery, the signal is said to be *pending*. Ordinarily, this interval cannot be detected by an application. However, a signal can be *blocked* from delivery to a process:

- If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the process, the signal remains pending until either it is unblocked or the action associated with it is set to ignore the signal.
- If the action associated with a blocked signal is to ignore the signal and if that signal is generated for the process, it is unspecified whether the signal is discarded immediately upon generation or remains pending.

Each process has a *signal mask* that defines the set of signals currently blocked from delivery to it. The signal mask for a process is initialized from that of its parent. The `sigaction`, `sigprocmask`, and `sigsuspend` functions control the manipulation of the signal mask.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of

the means by which the signal was originally generated. If a subsequent occurrence of a pending signal is generated, it is implementation-dependent as to whether the signal is delivered more than once. The C RTL delivers the signal only once. The order in which multiple, simultaneously pending signals are delivered to a process is unspecified.

4.2.4. Signal Actions

This section applies to the `sigaction`, `signal`, `sigvec`, and `ssignal` functions.

There are three types of action that can be associated with a signal:

SIG_DFL
SIG_IGN
pointer to a function

Initially, all signals are set to SIG_DFL or SIG_IGN prior to entry of the `main` routine (see the `exec` functions.) The actions prescribed by these values are:

- SIG_DFL — signal-specific default action
    - The default actions for the signals defined in this document are specified under `<signal.h>`.
    - If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCHLD signal is generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag. While a process is stopped, any additional signals that are sent to the process are not delivered until the process is continued, except SIGKILL which always terminates the receiving process. A process that is a member of an orphaned process group is not allowed to stop in response to the SIGSTOP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of these signals would stop such a process, the signal is discarded.
    - Setting a signal action to SIG_DFL for a signal that is pending and whose default action is to ignore the signal (for example, SIGCHLD), causes the pending signal to be discarded, whether or not it is blocked.
- SIG_IGN — ignore signal
    - Delivery of the signal has no effect on the process. The behavior of a process is undefined after it ignores a SIGFPE, SIGILL, or SIGSEGV signal that was not generated by `kill` or `raise`.
    - The system does not allow the action for the SIGKILL or SIGSTOP signals to be set to SIG_IGN.
    - Setting a signal action to SIG_IGN for a signal that is pending causes the pending signal to be discarded, whether or not it is blocked.
    - If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is unspecified.
- *pointer to a function* — catch signal
    - On delivery of the signal, the receiving process executes the signal-catching function at the specified address. After returning from the signal-catching function, the receiving process resumes execution at the point at which it was interrupted.
    - Specify the signal-catching function as:

```
void func(int signo);
```

Here, *func* is the specified signal-catching function and *signo* is the signal number of the signal being delivered.

- The behavior of a process is undefined after it returns normally from a signal-catching function for a SIGFPE, SIGKILL, or SIGSEGV signal that was not generated by `kill` or `raise`.
- The system does not allow a process to catch the signals SIGKILL and SIGSTOP.
- If a process establishes a signal-catching function for the SIGCHLD signal while it has a terminated child process for which it has not waited, it is unspecified whether a SIGCHLD signal is generated to indicate that child process.

4.2.5. Signal Handling and OpenVMS Exception Handling

This section discusses how C RTL signal handling is implemented with and interacts with OpenVMS exception handling. Information in this section allows you to write OpenVMS exception handlers that do not conflict with C RTL signal handling. For information on OpenVMS exception handling, see the *OpenVMS Procedure Calling and Condition Handling Standard*.

The C RTL implements signals with OpenVMS exceptions. When `gsignal` or `raise` is called, the signal number is translated to a particular OpenVMS exception, which is used in a call to LIB$SIGNAL. This mechanism is necessary to catch an OpenVMS exception resulting from a user error and translate it into a corresponding UNIX signal. For example, an ACCVIO resulting from a write to a NULL pointer is translated to a SIGBUS or SIGSEGV signal.

Tables 4.4 and 4.5 list the C RTL signal names, the corresponding OpenVMS Alpha and Integrity server system exceptions, the event that generates the signal, and the optional signal code for use with the `gsignal` and `raise` functions.

To call a signal handler that you have established with `signal` or `sigvec`, the C RTL intercepts the OpenVMS exceptions that correspond to signals by having an OpenVMS exception handler in the main routine of the program. If your program has a `main` function, then this exception handler is automatically established. If you do not have a `main` function, or if your main function is written in a language other than VSI C, then you must invoke the `VAXC$CRTL_INIT` routine to establish this handler.

The C RTL uses OpenVMS exceptions to implement the `setjmp` and `longjmp` functions. When the `longjmp` function is called, a C$_LONGJMP OpenVMS exception is signaled. To prevent the C$_LONGJMP exception from being interfered with by user exception handlers, use the `VAXC$ESTABLISH` routine to establish user OpenVMS exception handlers instead of calling LIB$ESTABLISH. The C$_LONGJMP mnemonic is defined in the `<errnodef.h>` header file.

If you want to use OpenVMS exception handlers and UNIX signals in your C program, your OpenVMS exception handler must be prepared to accept and resignal the OpenVMS exceptions listed in Table 4.4 *(Alpha only)*, as well as the C$_LONGJMP exception and any C$ facility exception that might be introduced in future versions of the C RTL. This is because UNIX signals are global in context, whereas OpenVMS exceptions are stack-frame based.

Consequently, an OpenVMS exception handler always receives the exception that corresponds to the UNIX signal before the C RTL exception handler in the main routine does. By resignalling the OpenVMS exception, you allow the C RTL exception handler to receive the exception. You can intercept

any of those OpenVMS exceptions yourself, but in doing so you will disable the corresponding UNIX signal.

Table 4.4. C RTL Signals and Corresponding OpenVMS Alpha Exceptions (Alpha Only)

Name	OpenVMS Exception	Generated By	Code
SIGABRT	SS$_OPCCUS	The `abort` function	–
SIGALRM	SS$_ASTFLT	The `alarm` function	–
SIGBUS	SS$_ACCVIO	Access violation	–
SIGBUS	SS$_CMODUSER	Change mode user	–
SIGCHLD	C$_SIGCHLD	Child process stopped	–
SIGEMT	SS$_COMPAT	Compatibility mode trap	–
SIGFPE	SS$_DECDIV	Decimal divide trap	FPE_DECDIV_TRAP
SIGFPE	SS$_DECINV	Decimal invalid operand trap	FPE_DECINV_TRAP
SIGFPE	SS$_DECOVF	Decimal overflow trap	FPE_DECOVF_TRAP
SIGFPE	SS$_HPARITH	Floating/decimal division by 0	FPE_FLTDIV_TRAP
SIGFPE	SS$_HPARITH	Floating overflow trap	FPE_FLTOVF_TRAP
SIGFPE	SS$_HPARITH	Floating underflow trap	FPE_FLTUND_TRAP
SIGFPE	SS$_HPARITH	Integer overflow	FPE_INTOVF_TRAP
SIGFPE	SS$_HPARITH	Invalid operand	FPE_INVOPR_TRAP
SIGFPE	SS$_HPARITH	Inexact result	FPE_INXRES_TRAP
SIGFPE	SS$_INTDIV	Integer div by zero	FPE_INTDIV_TRAP
SIGFPE	SS$_SUBRNG	Subscript out of range	FPE_SUBRNG_TRAP
SIGFPE	SS$_SUBRNG1	Subscript1 out of range	FPE_SUBRNG1_TRAP
SIGFPE	SS$_SUBRNG2	Subscript2 out of range	FPE_SUBRNG2_TRAP
SIGFPE	SS$_SUBRNG3	Subscript3 out of range	FPE_SUBRNG3_TRAP
SIGFPE	SS$_SUBRNG4	Subscript4 out of range	FPE_SUBRNG4_TRAP
SIGFPE	SS$_SUBRNG5	Subscript5 out of range	FPE_SUBRNG5_TRAP
SIGFPE	SS$_SUBRNG6	Subscript6 out of range	FPE_SUBRNG6_TRAP
SIGFPE	SS$_SUBRNG7	Subscript7 out of range	FPE_SUBRNG7_TRAP
SIGHUP	SS$_HANGUP	Data set hangup	–
SIGILL	SS$_OPCDEC	Reserved instruction	ILL_PRIVIN_FAULT
SIGILL	SS$_ROPRAND	Reserved operand	ILL_RESOP_FAULT
SIGINT	SS$_CONTROLC	OpenVMS Ctrl/C interrupt	–
SIGIOT	SS$_OPCCUS	Customer-reserved opcode	–
SIGKILL	SS$_ABORT	External signal only	–
SIGQUIT	SS$_CONTROLY	The `raise` function	–
SIGPIPE	SS$_NOMBX	No mailbox	–

Name	OpenVMS Exception	Generated By	Code
SIGPIPE	C$_SIGPIPE	Broken pipe	–
SIGSEGV	SS$_ACCVIO	Length violation	–
SIGSEGV	SS$_CMODSUPR	Change mode supervisor	–
SIGSYS	SS$_BADPARAM	Bad argument to system call	–
SIGTERM	Not implemented	–	–
SIGTRAP	SS$_BREAK	Breakpoint fault instruction	–
SIGUSR1	C$_SIGUSR1	The `raise` function	–
SIGUSR2	C$_SIGUSR2	The `raise` function	–
SIGWINCH [1]	C$_SIGWINCH [2]	The `raise` function	–

[1]Supported on OpenVMS Version 7.3 and higher.

[2]SS$_BADWINCNT when C$_SIGWINCH not defined (OpenVMS versions before 7.3).

OpenVMS Alpha Signal-Handling Notes

- While all signals that exist on OpenVMS VAX systems also exist on OpenVMS Alpha systems, the corresponding OpenVMS exceptions and code is different in a number of cases because on Alpha processors there are two new OpenVMS exceptions and several others that are obsolete.
- All floating-point exceptions on OpenVMS Alpha systems are signaled by the OpenVMS exception SS$_HPARITH (high-performance arithmetic trap). The particular type of trap that occurred is translated by the C RTL through use of the exception summary longword, which is set when a high-performance arithmetic trap is signaled.

Table 4.5. C RTL Signals and Corresponding OpenVMS Integrity server system Exceptions *(Integrity servers Only)*

Name	OpenVMS Exception	Generated By	Code
SIGABRT	SS$_OPCCUS	The `abort` function	–
SIGALRM	SS$_ASTFLT	The `alarm` function	–
SIGBUS	SS$_ACCVIO	Access violation	–
SIGBUS	SS$_CMODUSER	Change mode user	–
SIGCHLD	C$_SIGCHLD	Child process stopped	–
SIGEMT	SS$_COMPAT	Compatibility mode trap	–
SIGFPE	SS$_DECOVF	Decimal overflow trap	FPE_DECOVF_TRAP
SIGFPE	SS$_DECDIV	Decimal divide trap	FPE_DECDIV_TRAP
SIGFPE	SS$_DECINV	Decimal invalid operand trap	FPE_DECINV_TRAP
SIGFPE	SS$_FLTDENORMAL	Denormal operand fault	FPE_FLTDENORMAL _FAULT
SIGFPE	SS$_FLTDIV	Floating/decimal division by 0	FPE_FLTDIV_TRAP

Name	OpenVMS Exception	Generated By	Code
SIGFPE	SS$_FLTDIV_F	Floating divide by 0 fault	FPE_FLTDIV_FAULT
SIGFPE	SS$_FLTINE	Inexact operation trap	FPE_FLTINE_TRAP
SIGFPE	SS$_FLTINV	Invalid operation trap	FPE_FLTINV_TRAP
SIGFPE	SS$_FLTINV_F	Invalid operation fault	FPE_FLTINV_FAULT
SIGFPE	SS$_FLTOVF	Floating overflow trap	FPE_FLTOVF_TRAP
SIGFPE	SS$_FLTUND	Floating underflow trap	FPE_FLTUND_TRAP
SIGFPE	SS$_INTDIV	Integer division by 0	FPE_INTDIV_TRAP
SIGFPE	SS$_INTOVF	Integer overflow	FPE_INTOVF_TRAP
SIGFPE	SS$_SUBRNG	Subscript-range	FPE_SUBRNG_TRAP
SIGHUP	SS$_HANGUP	Data set hangup	–
SIGILL	SS$_OPCDEC	Reserved instruction	ILL_PRIVIN_FAULT
SIGILL	SS$_ROPRAND	Reserved operand	ILL_RESOP_FAULT
SIGINT	SS$_CONTROLC	OpenVMS Ctrl/C interrupt	–
SIGIOT	SS$_OPCCUS	Customer-reserved opcode	–
SIGKILL	SS$_ABORT	External signal only	–
SIGQUIT	SS$_CONTROLY	The `raise` function	–
SIGPIPE	SS$_NOMBX	No mailbox	–
SIGPIPE	C$_SIGPIPE	Broken pipe	–
SIGSEGV	SS$_ACCVIO	Length violation	–
SIGSEGV	SS$_CMODSUPR	Change mode supervisor	–
SIGSYS	SS$_BADPARAM	Bad argument to system call	–
SIGTERM	Not implemented	–	–
SIGTRAP	SS$_TBIT	TBIT trace trap	–
SIGTRAP	SS$_BREAK	Breakpoint fault instruction	–
SIGUSR1	C$_SIGUSR1	The `raise` function	–
SIGUSR2	C$_SIGUSR2	The `raise` function	–
SIGWINCH	C$_SIGWINCH	The `raise`function	–

4.3. Program Example

Example 4.1 shows how the `signal`, `alarm`, and `pause` functions operate. It also shows how to establish a signal handler to catch a signal, which prevents program termination.

Example 4.1. Suspending and Resuming Programs

```
/*    CHAP_4_SUSPEND_RESUME.C                                    */

/* This program shows how to alternately suspend and resume a   */
```

```
/* program using the signal, alarm, and pause functions.        */

#define SECONDS 5

#include <stdio.h>
#include <signal.h>

int number_of_alarms = 5;       /*  Set alarm counter.            */

void alarm_action(int);

main()
{
    signal(SIGALRM, alarm_action); /*  Establish a signal handler. */
                                   /*  to catch the SIGALRM signal.*/

    alarm(SECONDS);     /* Set alarm clock for 5 seconds. */

    pause();    /*  Suspend the process until     *
                 *  the signal is received.       */
}

void alarm_action(int x)
{
    printf("\t<%d\007>", number_of_alarms); /*  Print the value of  */
                                           /*  the alarm counter.  */

    signal(SIGALRM, alarm_action);      /*  Reset the signal.       */

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

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

Here is the sample output from Example 4.1:

```
$ RUN  EXAMPLE
       <5>   <4>   <3>   <2>   <1>
```

Chapter 5. Subprocess Functions

The VSI C Run-Time Library (C RTL) provides functions that allow you to create subprocesses from a VSI C program. The creating process is called the *parent* and the created subprocess is called the *child.*

To create a child process within the parent process, use the `exec` functions (`execl`, `execle`, `execv`, `execve`, `execlp`, and `execvp`) and the `vfork` function. Other functions are available to allow the parent and child to read and write data across processes (`pipe`) and to allow for synchronization of the two processes (`wait`). This chapter describes how to implement and use these functions.

The parent process can execute VSI C programs in its children, either synchronously or asynchronously. The number of children that can run simultaneously is determined by the /PRCLM user authorization quota established for each user on your system. Other quotas that may affect the use of subprocesses are /ENQLM (Queue Entry Limit), /ASTLM (AST Waits Limit), and /FILLM (Open File Limit).

This chapter discusses the subprocess functions. Table 5.1 lists and describes all the subprocess functions found in the C RTL. For more detailed information on each function, see the Reference Section.

Table 5.1. Subprocess Functions

Function	Description
Implementation of Child Processes	
`system`	Passes a given string to the host environment to be executed by a command processor.
`vfork`	Creates an independent child process.
The exec Functions	

`execl`
`execle`
`execlp`
`execv`
`execve`
`execvp`
	Pass the name of the image to be activated in a child process.

Synchronizing Process	

`wait`
`wait3`
`wait4`
`waitpid`
	Suspend the parent process until a value is returned from a child.

Interprocess Communication	
`pipe`	Allows for communication between a parent and child.

5.1. Implementing Child Processes in VSI C

Child processes are created by VSI C functions with the OpenVMS LIB$SPAWN RTL routine. (See the *VMS Run-Time Library Routines Volume* for information on LIB$SPAWN.) Using LIB$SPAWN allows you to create multiple levels of child processes; the parent's children can also spawn children, and so on, up to the limits allowed by the user authorization quotas discussed in the introduction to this chapter.

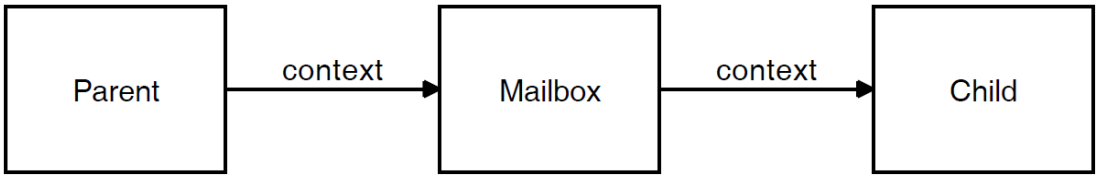
Child processes can only execute other VSI C programs. Other native-mode OpenVMS languages do not share the ability of VSI C to communicate between processes; if they do, they do not use the same mechanisms. The parent process must be run under an VSI supported command-language interpreter

(CLI), such as DCL. You cannot run the parent as a detached process or under control of a user-supplied CLI.

Enabling the DECC$DETACHED_CHILD_PROCESS feature logical allows child processes to be created as detached processes instead of subprocesses. This feature has only limited support. In some cases, the console cannot be shared between the parent process and the detached process, which can cause `exec` to fail.

Parent and child processes communicate through a mailbox as shown in Figure 5.1. This mailbox transfers the context in which the child will run. This *context mailbox* passes information to the child that it inherits from the parent, such as the names and file descriptors of all the files opened by the parent and the current location within those files. The mailbox is deleted by the parent when the child image exits.

Figure 5.1. Communications Links Between Parent and Child Processes



Note

The mailbox created by the `vfork` and `exec` functions is temporary. The logical name of this mailbox is VAXC$EXECMBX and is reserved for use by the C RTL.

The mailbox is created with a maximum message size and a buffer quota of 512 bytes each, unless the buffer size and quota are explicitly specified with the DECC$PIPE_BUFFER_SIZE or DECC$PIPE_BUFFER_QUOTA feature logicals, or with the *bufsize* or *bufquota* parameters of the `pipe` function. See the `pipe` function for more information.

You need the TMPMBX privilege to create a mailbox with these RTL functions. Since TMPMBX is the privilege required by the DCL commands PRINT and SUBMIT, most users on a system have this privilege. To see what system privileges you have, enter a SHOW PROCESS/PRIVILEGES command.

You cannot change the characteristics of these mailboxes. For more information on mailboxes, see the *VSI OpenVMS I/O User's Reference Manual*.

5.2. The exec Functions

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

When `vfork` is called by the parent, the `exec` function returns to the parent process. When `vfork` was called by an `exec` function, the `exec` function returns to itself, waits for the child to exit, and then exits the parent process. The `exec` function does not return to the parent process unless the parent calls `vfork` to save the return address.

In OpenVMS Version 7.2, the `exec` functions were enhanced to activate either executable images or DCL command procedures. If no file extension is specified in the *file_name* argument, the functions first search for the file with the .EXE file extension and then for the file with the .COM file extension. If both the executable image and the command procedure with the same name exist, you must explicitly specify the .COM file extension to force activating the command procedure.

For a DCL command procedure, the `exec` functions pass the first eight *arg0*, *arg1*, ..., arguments specified in the `exec` call to the command procedure as *P1*, *P2*, ... parameters, preserving the case.

Unlike UNIX based systems, the `exec` functions in the C RTL cannot always determine if the specified executable image or command procedure exists and can be activated and executed. Therefore, the `exec` functions might appear to succeed even though the specified file cannot be executed by the child process.

The status of the child process, returned to the parent process, indicates that the error occurred. You can retrieve this error code by using one of the functions from the `wait` family of functions.

Note

The `vfork` and `exec` functions in the C RTL on OpenVMS systems work differently than on UNIX systems:

- On UNIX systems, `vfork` creates a child process, suspends the parent, and starts the child running where the parent left off.
- On OpenVMS systems, `vfork` establishes context later used by an `exec` function, but it is the `exec` function, not `vfork`, that starts a process running the specified program.

For a programmer, the key differences are:

- On OpenVMS systems, code between the call to `vfork` and the call to an `exec` function runs in the parent process.

  On UNIX systems, this code runs in the child process.
- On OpenVMS systems, the child inherits open file descriptors and so on, at the point where the `exec` function is called.

  On UNIX systems, this occurs at the point where `vfork` is called.

5.2.1. exec Processing

The `exec` functions use the LIB$SPAWN routine to create the subprocess and activate the child image within the subprocess. This child process inherits the parent's environment, including all defined logical names and command-line interpreter symbols.

By default, child processes also inherit the default (working) directory of their parent process. However, you can use the `decc$set_child_default_dir` function to set the default directory for a child process as it begins execution. For more information about the `decc$set_child_default_dir` function, see the Reference Section.

The `exec` functions use the logical name VAXC$EXECMBX to communicate between parent and child; this logical name must not exist outside the context of the parent image.

The `exec` functions pass the following information to the child:

- The parent's `umask` value, which specifies whether any access is to be disallowed when a new file is created. For more information about the `umask` function, see the Reference Section.
- The file-name string associated with each file descriptor and the current position within each file. The child opens the file and calls `lseek` to position the file to the same location as the parent. If the

file is a record file, the child is positioned on a record boundary, regardless of the parent's position within the record. For more information about file descriptors, see Chapter 2. For more information on the `lseek` function, see the Reference Section.

This information is sent to the child for all descriptors known to the parent including all descriptors for open files, null descriptors, and duplicate descriptors.

File pointers are not transferred to the child. For files opened by a file pointer in the parent, only their corresponding file descriptors are passed to the child. The `fdopen` function must be called to associate a file pointer with the file descriptor if the child will access the file-by-file pointer. For more information about the `fdopen` function, see the Reference Section.

The DECC$EXEC_FILEATTR_INHERITANCE feature logical can be used to control whether or not a child process inherits file positioning, and if so, for which access modes. For more information on DECC$EXEC_FILEATTR_INHERITANCE, see Section 1.5.

- The signal database. Only SIG_IGN (ignore) actions are inherited. Actions specified as routines are changed to SIG_DFL (default) because the parent's signal-handling routines are inaccessible to the child.
- The environment and argument vectors.

When everything is transmitted to the child, `exec` processing is complete. Control in the parent process then returns to the address saved by `vfork` and the child's process ID is returned to the parent.

See Section 4.2.4 for a discussion of signal actions and the SIGCHLD signal.

5.2.2. exec Error Conditions

The `exec` functions will fail if LIB$SPAWN cannot create the subprocess. Conditions that can cause a failure include exceeding the subprocess quota or finding the communications by the context mailbox between the parent and child to be broken. Exceeding some quotas will not cause LIB$SPAWN to fail, but will put LIB$SPAWN into a wait state that can cause the parent process to hang. An example of such a quota is the Open File Limit quota.

You will need an Open File Limit quota of at least 20 files, with an average of three times the number of concurrent processes that your program will run. If you use more than one open pipe at a time, or perform I/O on several files at one time, this quota may need to be even higher. See your system manager if this quota needs to be increased.

When an `exec` function fails, a value of -1 is returned. After such a failure, the parent is expected to call either the `exit` or `_exit` function. Both functions then return to the parent's `vfork` call, which returns the child's process ID. In this case, the child process ID returned by the `exec` function is less than zero. For more information about the `exit` function, see the Reference Section.

5.3. Synchronizing Processes

A child process is terminated when the parent process terminates. Therefore, the parent process must check the status of its child processes before exiting. This is done using the C RTL function `wait`.

5.4. Interprocess Communication

A channel through which parent and child processes communicate is called a *pipe*. Use the `pipe` function to create a pipe.

5.5. Program Examples

Example 5.1 shows the basic procedures for executing an image in a child process. The child process in Example 5.1 prints a message 10 times.

Example 5.1. Creating the Child Process

```
/*      chap_5_exec_image.c     */

/* This example creates the child process.  The only    */
/* functionality given to the child is the ability to   */
/* print a message 10 times.                            */

#include <climsgdef.h>  /* CLI status values  */
#include <stdio.h>
#include <perror.h>
#include <processes.h>
#include <stdlib.h>

static const char *child_name = "chap_5_exec_image_child.exe" ;

main()
{
   int status,
       cstatus;

   /* NOTE:                                              */
   /*    Any local automatic variables, even those       */
   /*    having the volatile attribute, may have         */
   /*    indeterminant values if they are modified       */
   /*    between the vfork() call and the matching       */
   /*    exec() call.                                    */

❶   if ((status = vfork()) != 0) {
       /* This is either an error or                     */
       /* the "second" vfork return, taking us "back"    */
       /* to parent mode.                                */
❷       if (status < 0)
            printf("Parent - Child process failed\n");
       else {
            printf("Parent - Waiting for Child\n");
❸           if ((status = wait(&cstatus)) == -1)
              perror("Parent - Wait failed");
❹           else if (cstatus == CLI$_IMAGEFNF)
              printf("Parent - Child does not exist\n");
            else
              printf("Parent - Child final status: %d\n", cstatus);
        }
    }
❺    else {  /* The FIRST Vfork return is zero, do the exec */
          printf("Parent - Starting Child\n");
          if ((status = execl(child_name, 0)) == -1) {
              perror("Parent - Execl failed");
              exit(EXIT_FAILURE);
         }
    }
}
```

```
------------------------------------------------------

/*      CHAP_5_EXEC_IMAGE_CHILD.C                    */

/* This is the child program that writes a message   */
/* through the parent to "stdout"                    */

#include <stdio.h>

main()
{
    int i;

    for (i = 0; i < 10; i++)
        printf("Child - executing\n");
    return (255) ;    /* Set an unusual success stat */
}
```

Key to Example 5.1:

❶ The `vfork` function is called to set up the return address for the `exec` call.

The `vfork` function is normally used in the expression of an `if` statement. This construct allows you to take advantage of the double return aspect of `vfork`, since one return value is 0 and the other is nonzero.

❺ A 0 return value is returned the first time `vfork` is called and the parent executes the `else` clause associated with the `vfork` call, which calls `execl`.

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

❸ The `wait` function is used to synchronize the parent and child processes.

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

In Example 5.2, the parent passes arguments to the child process.

Example 5.2. Passing Arguments to the Child Process

```
/*       CHAP_5_CHILDARG.C                                    */

/* In this example, the arguments are placed in an array, gargv,     */
/* but they can be passed to the child explicitly as a zero-         */
/* terminated series of character strings. The child program in this */
/* example writes the arguments that have been passed it to stdout.  */

#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <perror.h>
#include <processes.h>

const char *child_name = "chap_5_childarg_child.exe" ;

main()
{
    int status,
        cstatus;
```

```
    char *gargv[] =
    {"Child", "ARGC1", "ARGC2", "Parent", 0};

    if ((status = vfork()) != 0) {
        if (status < -1)
            printf("Parent - Child process failed\n");
        else {
            printf("Parent - waiting for Child\n");
            if ((status = wait(&cstatus)) == -1)
                perror("Parent - Wait failed");
            else if (cstatus == CLI$_IMAGEFNF)
                printf("Parent - Child does not exist\n");
            else
                printf("Parent - Child final status: %x\n",
                       cstatus);
        }
    }
    else {
        printf("Parent - Starting Child\n");
        if ((status = execv(child_name, gargv)) == -1) {
            perror("Parent - Exec failed");
            exit(EXIT_FAILURE);
        }
    }
}

------------------------------------------------------------
/*       CHAP_5_CHILDARG_CHILD.C                           */

/* This is a child program that echo's its arguments   */

#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    int i;

    printf("Program name: %s\n", argv[0]);

    for (i = 1; i < argc; i++)
        printf("Argument %d: %s\n", i, argv[i]);
    return(255) ;
}
```

Example 5.3 shows how to use the `wait` function to check the final status of multiple children being run simultaneously.

Example 5.3. Checking the Status of Child Processes

```
/*        CHAP_5_CHECK_STAT.C                                */

/* In this example 5 child processes are started.  The wait()   */
/* function is placed in a separate for loop so that it is      */
/* called once for each child. If wait() were called within     */
/* the first for loop, the parent would wait for one child to   */
/* terminate before executing the next child. If there were     */
```

```
/* only one wait request, any child still running when the      */
/* parent exits would terminate prematurely.                    */

#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <perror.h>
#include <processes.h>

const char *child_name = "chap_5_check_stat_child.exe" ;

main()
{
    int status,
        cstatus,
        i;

    for (i = 0; i < 5; i++) {
        if ((status = vfork()) == 0) {
            printf("Parent - Starting Child %d\n", i);
            if ((status = execl(child_name, 0)) == -1) {
                perror("Parent - Exec failed");
                exit(EXIT_FAILURE);
            }
        }
        else if (status < -1)
            printf("Parent - Child process failed\n");
    }

    printf("Parent - Waiting for children\n");

    for (i = 0; i < 5; i++) {
        if ((status = wait(&cstatus)) == -1)
            perror("Parent - Wait failed");
        else if (cstatus == CLI$_IMAGEFNF)
            printf("Parent - Child does not exist\n");
        else
            printf("Parent - Child %X final status: %d\n",
                   status, cstatus);
    }
}
```

Example 5.4 shows how to use the `pipe` and `dup2` functions to communicate between a parent and child process through specific file descriptors. The `#define` preprocessor directive defines the preprocessor constants `inpipe` and `outpipe` as the names of file descriptors 11 and 12.

Example 5.4. Communicating Through a Pipe

```
/*        CHAP_5_PIPE.C                                      */

/* In this example, the parent writes a string to the pipe for  */
/* the child to read.  The child then writes the string back    */
/* to the pipe for the parent to read.  The wait function is    */
/* called before the parent reads the string that the child has */
/* passed back through the pipe.  Otherwise, the reads and      */
/* writes will not be synchronized.                             */

#include <perror.h>
```

```
#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <processes.h>
#include <unixio.h>

#define inpipe 11
#define outpipe 12

const char *child_name = "chap_5_pipe_child.exe" ;

main()
{
    int pipes[2];
    int mode,
        status,
        cstatus,
        len;
    char *outbuf,
        *inbuf;

    if ((outbuf = malloc(512)) == 0) {
        printf("Parent - Outbuf allocation failed\n");
        exit(EXIT_FAILURE);
    }

    if ((inbuf = malloc(512)) == 0) {
        printf("Parent - Inbuf allocation failed\n");
        exit(EXIT_FAILURE);
    }
    if (pipe(pipes) == -1) {
        printf("Parent - Pipe allocation failed\n");
        exit(EXIT_FAILURE);
    }

    dup2(pipes[0], inpipe);
    dup2(pipes[1], outpipe);
    strcpy(outbuf, "This is a test of two-way pipes.\n");

    status = vfork();

    switch (status) {
    case 0:
        printf("Parent - Starting child\n");
        if ((status = execl(child_name, 0)) == -1) {
            printf("Parent - Exec failed");
            exit(EXIT_FAILURE);
        }
        break;

    case -1:
        printf("Parent - Child process failed\n");
        break;

    default:
        printf("Parent - Writing to child\n");
```

```
        if (write(outpipe, outbuf, strlen(outbuf) + 1) == -1) {
            perror("Parent - Write failed");
            exit(EXIT_FAILURE);
        }
        else {
            if ((status = wait(&cstatus)) == -1)
                perror("Parent - Wait failed");

            if (cstatus == CLI$_IMAGEFNF)
                printf("Parent - Child does not exist\n");
            else {
                printf("Parent - Reading from child\n");
                if ((len = read(inpipe, inbuf, 512)) <= 0) {
                    perror("Parent - Read failed");
                    exit(EXIT_FAILURE);
                }
                else {
                    printf("Parent: %s\n", inbuf);
                    printf("Parent - Child final status: %d\n",
                            cstatus);
                }
            }
        }
        break;
    }
}

------------------------------------------------------------------
/*        CHAP_5_PIPE_CHILD.C                                     */

/* This is a child program which reads from a pipe and writes   */
/* the received message back to its parent.                     */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define inpipe 11
#define outpipe 12

main()
{
    char *buffer;
    int len;

    if ((buffer = malloc(512)) == 0) {
        perror("Child - Buffer allocation failed\n");
        exit(EXIT_FAILURE);
    }

    printf("Child - Reading from parent\n");
    if ((len = read(inpipe, buffer, 512)) <= 0) {
        perror("Child - Read failed");
        exit(EXIT_FAILURE);
    }
    else {
        printf("Child: %s\n", buffer);
```

```
        printf("Child - Writing to parent\n");
        if (write(outpipe, buffer, strlen(buffer) + 1) == -1) {
            perror("Child - Write failed");
            exit(EXIT_FAILURE);
        }
    }
    exit(EXIT_SUCCESS);
}
```

Chapter 6. Curses Screen Management Functions and Macros

This chapter describes the screen management routines available with VSI C for OpenVMS Systems.

The OpenVMS Curses screen management package is supported on all OpenVMS systems.

On OpenVMS Alpha systems, two screen management packages are supported: OpenVMS Curses and a more UNIX compatible package based on the Berkeley Standard Distribution (BSD) Curses software.[1] See Section 6.1 for more information.

Furthermore, the C RTL offers a Curses package based on the 4.4BSD Berkeley Software Distribution. Documentation on the 4.4BSD Curses package can be found in *Screen Updating and Cursor Movement Optimization: A Library Package*, by Kenneth C.R.C. Arnold.

The functions and macros in the OpenVMS and BSD-based Curses packages are nearly the same. Most differences between them are called out in this chapter. Otherwise, this chapter makes no distinction between the two Curses packages, and refers to "Curses" or the "Curses functions and macros".

6.1. Using the BSD-Based Curses Package (Alpha only)

The `<curses.h>` header file required to use the BSD-based Curses implementation is provided with the VSI C compiler on OpenVMS Alpha systems.

Existing programs are not affected by the BSD-based Curses functions because the OpenVMS Curses functions are still available as the default Curses package. (Note that is a change from previous versions of VSI C, where BSD-based Curses was the default.)

To get the 4.4BSD Curses implementation, you must compile modules that include `<curses.h>` with the following qualifier:

/DEFINE=_BSD44_CURSES

The BSD-based Curses functions do not provide the support required to call the OpenVMS SMG$ routines with the pasteboard and keyboard allocated by the Curses functions. Consequently, Curses programs that rely on calling SMG$ entry points, as well as Curses functions, must continue to use the OpenVMS Curses implementation.

The BSD-based Curses implementation is not interoperable with the old implementation. Attempts to mix calls to the new functions and the old functions will result in incorrect output displayed on the screen and could result in an exception from an SMG$ routine.

6.2. Curses Overview

Curses, the VSI C Screen Management Package, is composed of C RTL functions and macros that create and modify defined sections of the terminal screen and optimize cursor movement. Using the screen

[1]Copyright (c) 1981 Regents of the University of California.

management package, you can develop a user interface that is both visually attractive and user-friendly. Curses is terminal-independent and provides simplified terminal screen formatting and efficient cursor movement.

Most Curses functions and macros are listed in pairs where the first routine is a macro and the second is a function beginning with the prefix “w,”for “window.” These prefixes are delimited by brackets ([ ]). For example, `[w]addstr` designates the `addstr` macro and the `waddstr` function. The macros default to the window `stdscr`; the functions accept a specified window as an argument.

To access the Curses functions and macros, include the `<curses.h>` header file.

The terminal-independent Screen Management Software, which is part of the OpenVMS RTL, is used to implement Curses. For portability purposes,most functions and macros are designed to perform in a manner similar to other C implementations. However, the Curses routines depend on the OpenVMS system and its Screen Management Software, so performance of some functions and macros could differ slightly from those of other implementations.

Some functions and macros available on other systems are not available with the C RTL Curses package.

Some functions, such as `[w]clrattr`, `[w]insstr`, `mv[w]insstr`, and `[w]setattr` are specific to VSI C for OpenVMS systems and are not portable.

Table 6.1 lists all of the Curses functions and macros found in the C RTL. For more detailed information on each function and macro, see the Reference Section.

Table 6.1. Curses Functions and Macros

Function or Macro	Description
`[w]addch`	Adds a character to the window at the current position of the cursor.
`[w]addstr`	Adds a string to the window at the current position of the cursor.
`box`	Draws a box around the window.
`[w]clear`	Erases the contents of the specified window and resets the cursor to coordinates (0,0).
`clearok`	Sets the clear flag for the window.
`[w]clrattr`	Deactivates the video display attribute within the window.
`[w]clrtobot`	Erases the contents of the window from the current position of the cursor to the bottom of the window.
`[w]clrtoeol`	Erases the contents of the window from the current cursor position to the end of the line on the specified window.
`[no]crmode`	Sets and unsets the terminal from c break mode.
`[w]delch`	Deletes the character on the specified window at the current position of the cursor.
`[w]deleteln`	Deletes the line at the current position of the cursor.
`delwin`	Deletes the specified window from memory.
`[no]echo`	Sets the terminal so that characters mayor may not be echoed on the terminal screen.
`endwin`	Clears the terminal screen and frees any virtual memory allocated to Curses data structures.
`[w]erase`	Erases the window by painting it with blanks.

Function or Macro	Description
[w]getch	Gets a character from the terminal screen and echoes it on the specified window.
[w]getstr	Gets a string from the terminal screen,stores it in a character variable, and echoes it on the specified window.
getyx	Puts the (y,x) coordinates of the current cursor position on the window in the variables y and x.
[w]inch	Returns the character at the current cursor position on the specified window without making changes to the window.
initscr	Initializes the terminal-type data and all screen functions.
[w]insch	Inserts a character at the current cursor position in the specified window.
[w]insertln	Inserts a line above the line containing the current cursor position.
[w]insstr	Inserts a string at the current cursor position on the specified window.
leaveok	Leaves the cursor at the current coordinates after an update to the window.
longname	Assigns the full terminal name to a character name that must be large enough to hold the character string.
[w]move	Changes the current cursor position on the specified window.
mv[w]addch	Moves the cursor and adds a character to the specified window.
mv[w]addstr	Moves the cursor and adds a string to the specified window.
mvcur	Moves the terminal's cursor.
mv[w]delch	Moves the cursor and deletes a character on the specified window.
mv[w]getch	Moves the cursor, gets a character from the terminal screen, and echoes it on the specified window.
mv[w]getstr	Moves the cursor, gets a string from the terminal screen, stores it in a variable, and echoes it on the specified window.
mv[w]inch	Moves the cursor and returns the character on the specified window without making changes to the window.
mv[w]insch	Moves the cursor and inserts a character in the specified window.
mv[w]insstr	Moves the cursor and inserts a string in the specified window.
mvwin	Moves the starting position of the window to the specified coordinates.
newwin	Creates a new window with lines and columns starting at the coordinates on the terminal screen.
[no]nl	Provided only for UNIX software compatibility and has no functionality in the OpenVMS environment.
overlay	Writes the contents of one window that will fit over the contents of another window, beginning at the starting coordinates of both windows.
overwrite	Writes the contents of one window,insofar as it will fit, over the contents of another window beginning at the starting coordinates of both windows.
[w]printw	Performs a printf on the window starting at the current position of the cursor.
[no]raw	Provided only for UNIX software compatibility and has no functionality in the OpenVMS environment.

Function or Macro	Description
[w]refresh	Repaints the specified window on the terminal screen.
[w]scanw	Performs a scanf on the window.
scroll	Moves all the lines on the window up one line.
scrollok	Sets the scroll flag for the specified window.
[w]setattr	Activates the video display attribute within the window.
[w]standend	Deactivates the boldface attribute for the specified window.
[w]standout	Activates the boldface attribute of the specified window.
subwin	Creates a new sub window with lines and columns starting at the coordinates on the terminal screen.
touchwin	Places the most recently edited version of the specified window on the terminal screen.
wrapok	OpenVMS Curses only. Allows the wrapping of a word from the right border of the window to the beginning of the next line.

6.3. Curses Terminology

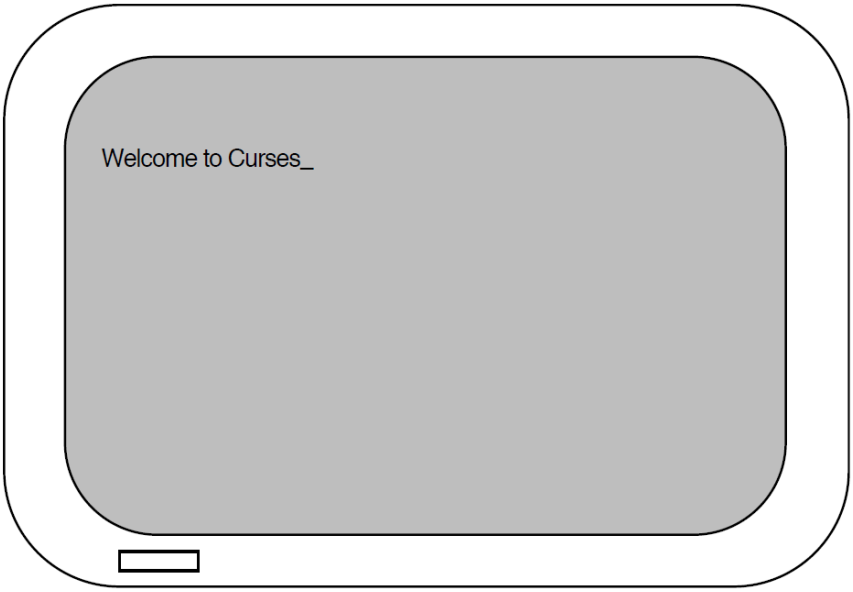
This section explains some of the Curses terminology and shows you how Curses looks on the terminal screen.

Consider a Curses application as being a series of overlapping windows. Window overlapping is called *occlusion*. To distinguish the boundaries of these occluding windows, you can outline the rectangular windows with specified characters, or you can turn on the reverse video option (make the window a light background with dark writing).

6.3.1. Predefined Windows (stdscr and curscr)

Initially, two windows the size of the terminal screen are predefined by Curses. These windows are called stdscr and curscr. The stdscr window is defined for your use. Many Curses macros default to this window. For example, if you draw a box around stdscr, move the cursor to the left-corner area of the screen, write a string to stdscr, and then display stdscr on the terminal screen, your display will look like that in Figure 6.1.

Figure 6.1. An Example of the stdscr Window



The second predefined window, `curscr`, is designed for internal Curses work; it is an image of what is currently displayed on the terminal screen. The only VSI C for OpenVMS Curses function that will accept this window as an argument is `clearok`. Do not write to or read from `curscr`. Use `stdscr` and user-defined windows for all your Curses applications.

6.3.2. User-Defined Windows

You can occlude `stdscr` with your own windows. The size and location of each window is given in terms of the number of lines, the number of columns, and the starting position.

The lines and columns of the terminal screen form a coordinate system, or grid, on which the windows are formed. You specify the starting position of a window with the (y,x) coordinates on the terminal screen where the upper left corner of the window is located. The coordinates (0,0) on the terminal screen, for example, are the upper left corner of the screen.

The entire area of the window must be within the terminal screen borders;windows can be as small as a single character or as large as the entire terminal screen. You can create as many windows as memory allows.

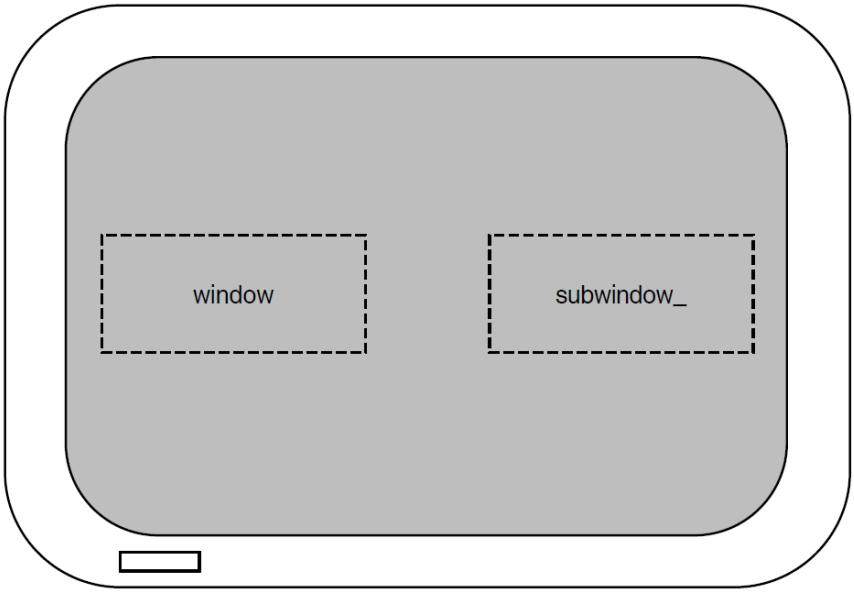
When writing to or deleting from windows, changes do not appear on the terminal screen until the window is *refreshed*. When refreshing a window, you place the updated window onto the terminal screen, which leaves the rest of the screen unaltered.

All user-defined windows, by default, occlude `stdscr`. You can create two or more windows that occlude each other as well as `stdscr`. When writing data to one occluding window, the data is not written to the underlying window.

You can create overlapping windows (called *subwindows*). A declared window must contain the entire area of its subwindow. When writing data to a subwindow or to the portion of the window overlapped by the subwindow, both windows contain the new data. For instance, if you write data to a subwindow and then delete that subwindow, the data is still present on the underlying window.

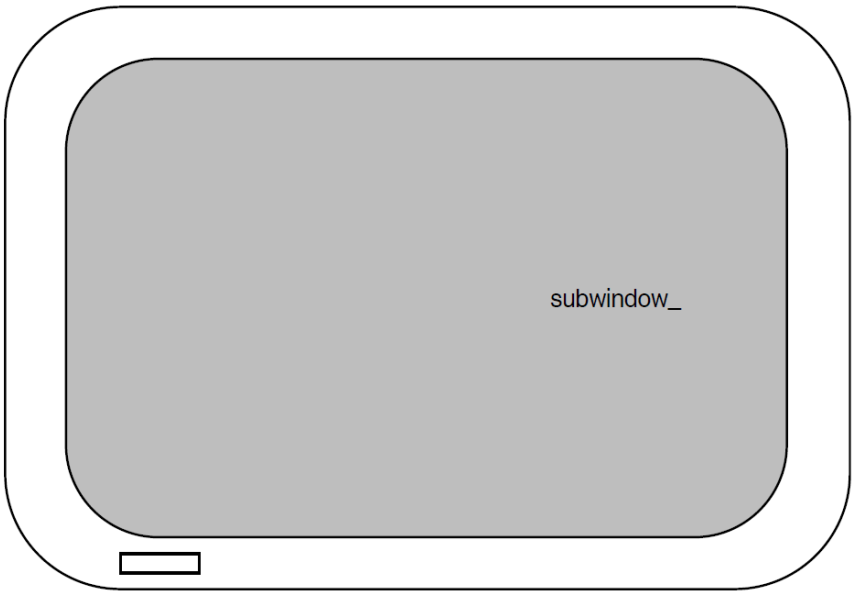
If you create a window that occludes `stdscr` and a subwindow of `stdscr`, your terminal screen will look like Figure 6.2.

Figure 6.2. Displaying Windows and Subwindows



If you delete both the user-defined window and the subwindow, and then update the terminal screen with the new image, your terminal screen will look like Figure 6.3.

Figure 6.3. Updating the Terminal Screen



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

6.4. Getting Started with Curses

There are commands that you must use to initialize and restore the terminal screen when using Curses Screen Management functions and macros. Also, there are predefined variables and constants on which Curses depends. Example 6.1 shows how to set up a program using Curses.

Example 6.1. A Curses Program

```
❶#include <curses.h>
```

```
❷WINDOW  *win1, *win2, *win3;

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

Key to Example 6.1:

❶ The preprocessor directive includes the <curses.h> header file, which defines the data structures and variables used to implement Curses. The <curses.h> header file includes the <stdio.h> header file, so it is not necessary to duplicate this action by including <stdio.h> again in the program source code. You must include <curses.h> to use any of the Curses functions or macros.

❷ In the example, WINDOW is a data structure defined in <curses.h>. You must declare each user-specified window in this manner. In Example 6.1, the three defined windows are win1, win2, and win3.

❸ The initscr and endwin functions begin and end the window editing session. The initscr function clears the terminal screen (for OpenVMS Curses only; BSD-based Curses does not clear the screen), and allocates space for the windows stdscr and curscr. The endwin function deletes all windows and clears the terminal screen.

Most Curses users wish to define and modify windows. Example 6.2 shows you how to define and write to a single window.

Example 6.2. Manipulating Windows

```
#include <curses.h>

WINDOW  *win1, *win2, *win3;

main()
{
   initscr();

❶   win1 = newwin(24, 80, 0, 0);
❷   mvwaddstr(win1, 2, 2, "HELLO");
      .
      .
      .
   endwin();
}
```

Key to Example 6.2:

❶ The newwin function defines a window 24 rows high and 80 columns wide with a starting position at coordinates (0,0), the upper left corner of the terminal screen. The program assigns these attributes to win1. The coordinates are specified as follows: (lines,columns) or (y,x).

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

Note

Most Curses macros update `stdscr` by default. Curses functions that update other windows have the same name as the macros but with the added prefix "w". For example, the `addstr` macro adds a given string to `stdscr` at the current cursor position. The `waddstr` function adds a given string to a specified window at the current cursor position.

When updating a window, specify the cursor position relative to the origin of the window, not the origin of the terminal screen. For example, if a window has a starting position of (10,10) and you want to add a character to the window at its starting position,specify the coordinates (0,0), not (10,10).

The string HELLO in Example 6.2 does not appear on the terminal screen until you refresh the screen. You accomplish this by using the `wrefresh` function. Example 6.3 shows how to display the contents of `win1` on the terminal screen.

Example 6.3. Refreshing the Terminal Screen

```
#include <curses.h>

WINDOW  *win1, *win2, *win3;

main()
{
   initscr();

   win1 = newwin(22, 60, 0, 0);
   mvwaddstr(win1, 2, 2, "HELLO");
   wrefresh(win1);
      .
      .
      .
   endwin();
}
```

The `wrefresh` function updates just the region of the specified window on the terminal screen. When the program is executed, the string HELLO appears on the terminal screen until the program executes the `endwin` function. The `wrefresh` function only refreshes the part of the window on the terminal screen that is not overlapped by another window. If `win1` was overlapped by another window and you want all of `win1` to be displayed on the terminal screen, call the `touchwin` function.

6.5. Predefined Variables and Constants

The `<curses.h>` header file defines variables and constants useful for implementing Curses (see Table 6.2).

Table 6.2. Curses Predefined Variables and #define Constants

Name	Type	Description
curscr	WINDOW *	Window of current screen
stdscr	WINDOW *	Default window
LINES	int	Number of lines on the terminal screen
COLS	int	Number of columns on the terminal screen

Name	Type	Description
ERR	—	Flag (0) for failed routines
OK	—	Flag (1) for successful routines
TRUE	—	Boolean true flag (1)
FALSE	—	Boolean false flag (0)
_BLINK	—	Parameter for `setattr` and `clrattr`
_BOLD	—	Parameter for `setattr` and `clrattr`
_REVERSE	—	Parameter for `setattr` and `clrattr`
_UNDERLINE	—	Parameter for `setattr` and `clrattr`

For example, you can use the predefined macro ERR to test the success or failure of a Curses function. Example 6.4 shows how to perform such a test.

Example 6.4. Curses Predefined Variables

```
#include <curses.h>

WINDOW  *win1, *win2, *win3;

main()
{
   initscr();
   win1 = newwin(10, 10, 1, 5);
      .
      .
      .
   if (mvwin(win1, 1, 10) == ERR)
      addstr("The MVWIN function failed.");
      .
      .
      .
   endwin();
}
```

In Example 6.4, if the `mvwin` function fails, the program adds a string to `stdscr` that explains the outcome. The Curses `mvwin` function moves the starting position of a window.

6.6. Cursor Movement

In the UNIX system environment, you can use Curses functions to move the cursor across the terminal screen. With other implementations, you can either allow Curses to move the cursor using the `move` function, or you can specify the origin and the destination of the cursor to the `mvcur` function, which moves the cursor in a more efficient manner.

In VSI C for OpenVMS systems, the two functions are functionally equivalent and move the cursor with the same efficiency.

Example 6.5 shows how to use the `move` and `mvcur` functions.

Example 6.5. The Cursor Movement Functions

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

```
main()
{
   initscr();
      .
      .
      .
❶   clear();
❷   move(10, 10);
❸   move(LINES/2, COLS/2);
❹   mvcur(0, COLS-1, LINES-1, 0);
      .
      .
      .
   endwin();
}
```

Key to Example 6.5:

❶ The `clear` macro erases `stdscr` and positions the cursor at coordinates (0,0).
❷ The first occurrence of `move` moves the cursor to coordinates (10,10).
❸ The second occurrence of `move` uses the predefined variables LINES and COLS to calculate the center of the screen (by calculating the value of half the number of LINES and COLS on the screen).
❹ The `mvcur` function forces absolute addressing. This function can address the lower left corner of the screen by claiming that the cursor is presently in the upper right corner. You can use this method if you are unsure of the current position of the cursor, but `move` works just as well.

6.7. Program Example

The following program example shows the effects of many of the Curses macros and functions. You can find explanations of the individual lines of code, if not self-explanatory, in the comments to the right of the particular line. Detailed discussions of the functions follow the source code listing.

Example 6.6 shows the definition and manipulation of one user-defined window and `stdscr`.

Example 6.6. stdscr and Occluding Windows

```
/*        CHAP_6_STDSCR_OCCLUDE.C                        */

/* This program defines one window: win1.  win1 is      */
/* located towards the center of the default window     */
/* stdscr.  When writing to an occluding window (win1)  */
/* that is later erased, the writing is erased as well. */

#include <curses.h>     /* Include header file.         */

WINDOW *win1;                   /* Define windows.      */

main()
{
    char str[80];               /* Variable declaration.*/

    initscr();  /* Set up Curses.   */
    noecho();   /* Turn off echo.   */

    /* Create window.             */
```

```
    win1 = newwin(10, 20, 10, 10);

    box(stdscr, '|', '-');  /* Draw a box around stdscr. */
    box(win1, '|', '-');    /* Draw a box around win1.   */

    refresh();  /* Display stdscr on screen.  */

    wrefresh(win1);     /* Display win1 on screen.  */

❶   getstr(str);      /* Pause. Type a few words!   */

   mvaddstr(22, 1, str);
❷   getch();
     /* Add string to win1.        */

    mvwaddstr(win1, 5, 5, "Hello");
    wrefresh(win1);     /* Add win1 to terminal scr.  */

    getch();    /* Pause.  Press Return.      */

    delwin(win1);       /* Delete win1.           */

❸   touchwin(stdscr); /* Refresh all of stdscr.     */

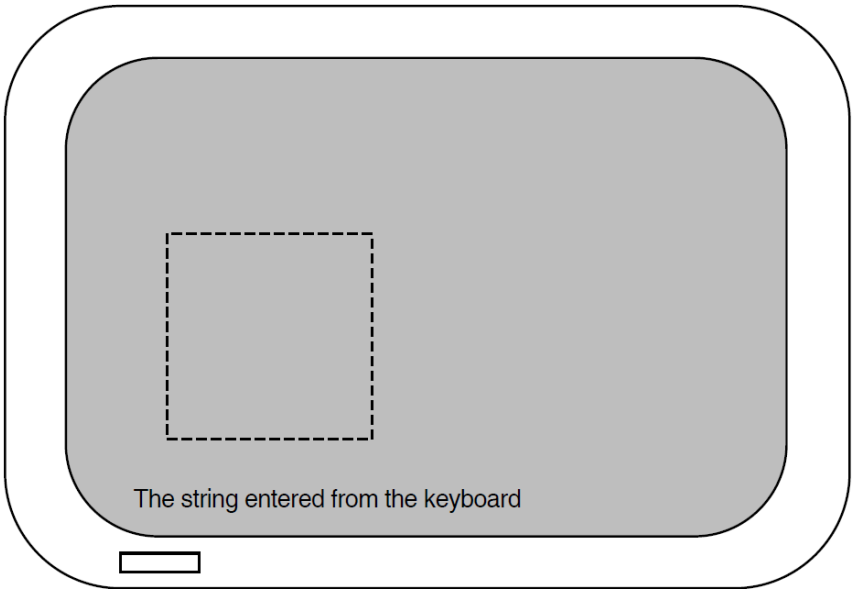
   getch();                  /* Pause.  Press Return.      */
   endwin();                 /* Ends session.              */
}
```

Key to Example 6.6:

❶ The program waits for input. The echo was disabled using the `noecho` macro, so the words that you type do not appear on `stdscr`. However, the macro stores the words in the variable `str` for use elsewhere in the program.

❷ The `getch` macro causes the program to pause. When you are finished viewing the screen, press Return so the program can resume. The `getch` macro refreshes `stdscr` on the terminal screen without calling `refresh`. The screen appears like Figure 6.4.

Figure 6.4. An Example of the getch Macro



❸ The `touchwin` function refreshes the screen so that all of `stdscr` is visible and the deleted occluding window no longer appears on the screen.

Chapter 7. Math Functions

Table 7.1 lists and describes the math functions in the VSI C Run-Time Library (C RTL). For more detailed information on each function, see the Reference Section.

Table 7.1. Math Functions

Function	Description
`abs`	Returns the absolute value of an integer.
`acos`	Returns the arc cosine of its radian argument, in the range [0, π] radians.
`acosd`	Returns the arc cosine of its radian argument, in the range [0,180] degrees.
`acosh`	Returns the hyperbolic arc cosine of its argument.
`asin`	Returns the arc sine of its radian argument in the range [-π/2,π/2] radians.
`asind`	Returns the arc sine of its radian argument, in the range [-90,90] degrees.
`asinh`	Returns the hyperbolic arc sine of its argument.
`atan`	Returns the arc tangent of its radian argument, in the range [-π/2,π/2] radians.
`atand`	Returns the arc tangent of its radian argument, in the range [-90,90] degrees.
`atan2`	Returns the arc tangent of *y*/ *x* (its two radian arguments), in the range [- π, π] radians.
`atand2`	Returns the arc tangent of *y*/ *x* (its two radian arguments), in the range [-180,180] degrees.
`atanh`	Returns the hyperbolic arc tangent of its radian argument.
`cabs`	Returns the absolute value of a complex number as: sqrt ($x^2 + y^2$).
`cbrt`	Returns the rounded cube root of its argument.
`ceil`	Returns the smallest integer greater than or equal to its argument.
`copysign`	Returns its first argument with the same sign as its second.
`cos`	Returns the cosine of its radian argument in radians.
`cosd`	Returns the cosine of its radian argument in degrees.
`cosh`	Returns the hyperbolic cosine of its argument.
`cot`	Returns the cotangent of its radian argument in radians.
`cotd`	Returns the cotangent of its radian argument in degrees.

`drand48`
`erand48`
`jrand48`
`lrand48`
`mrand48`
`nrand48`
	Generate uniformly distributed pseudorandom number sequences. Return 48-bit, non-negative, double-precision floating-point values.

Function	Description
`erf`	Returns the error function of its argument.
`erfc`	Returns (1.0–`erf`(*x*)).
`exp`	Returns the base e raised to the power of the argument.
`expm1`	Returns `exp`(*x*)–1.
`fabs`	Returns the absolute value of a floating-point value.
`finite`	Returns 1 if its argument is a finite number; 0 if not.
`floor`	Returns the largest integer less than or equal to its argument.
`fmod`	Computes the floating-point remainder of its first argument divided by its second.
`fp_class`	Determines the class of IEEE floating-point values, returning a constant from the `<fp_class.h>` header file.
`isnan`	Test for NaN. Returns 1 if its argument is a NaN; 0 if not.

`j0`
`j1`
`jn`
	Compute Bessel functions of the first kind.

`frexp`	Calculates the fractional and exponent parts of a floating-point value.
`hypot`	Returns the square root of the sum of the squares of two arguments.
`initstate`	Initializes random number generators.
`labs`	Returns the absolute value of an integer as a `long int`.
`lcong48`	Initializes a 48-bit uniformly distributed pseudorandom number sequence.
`lgamma`	Computes the logarithm of the gamma function.
`qabs` `llabs`	Return the absolute value of an `__int64` integer.
`ldexp`	Returns its first argument multiplied by 2 raised to the power of its second argument.
`ldiv` `div`	Return the quotient and remainder after the division of their arguments.
`qdiv` `lldiv`	Return the quotient and remainder after the division of their arguments.

`log`
`log2`
`log10`
	Return the logarithm of their arguments.

`log1p`	Computes ln(1+ *x*) accurately.
`logb`	Returns the radix-independent exponent of its argument.
`nextafter`	Returns the next machine-representable number following *x* in the direction of *y*.
`nint`	Returns the nearest integral value to the argument.

Function	Description
`modf`	Returns the positive fractional part of its first argument and assigns the integral part to the object whose address is specified by the second argument.
`pow`	Returns the first argument raised to the power of the second.
`rand` `srand`	Return pseudorandom numbers in the range 0 to $2^{31}-1$.
`random` `srandom`	Generate pseudorandom numbers in a more random sequence.
`rint`	Rounds its argument to an integral value according to the current IEEE rounding direction specified by the user.
`scalb`	Returns the exponent of a floating-point number.
`seed48` `srand48`	Initialize a 48-bit random number generator.
`setstate`	Restarts and changes random number generators.
`sin`	Returns the sine of its radian argument in radians.
`sind`	Returns the sine of its radian argument in degrees.
`sinh`	Returns the hyperbolic sine of its argument.
`sqrt`	Returns the square root of its argument.
`tan`	Returns the tangent of its radian argument in radians.
`tand`	Returns the tangent of its radian argument in degrees.
`tanh`	Returns the hyperbolic tangent of its argument.
`trunc`	Truncates its argument to an integral value.
`unordered`	Returns 1 if either or both of its arguments is a NaN; 0, if not.

`y0`
`y1`
`yn`
	Compute Bessel functions of the second kind.

7.1. Math Function Variants – float, long double

Additional math routine variants are supported for VSI C. They are defined in `<math.h>` and are `float` and `long double` variants of the routines listed in Table 7.1.

Float variants take `float` arguments and return `float` values. Their names have an `f` suffix. For example:

```
float cosf (float x);
float tandf (float x);
```

Long double variants take `long double` arguments and return `long double` values. Their names have an `l` suffix. For example:

```
long double cosl (long double x);
long double tandl (long double x);
```

All math routine variants are included in the Reference Section of this manual.

Note that for programs compiled without /L_DOUBLE=64 (that is, compiled with the default /L_DOUBLE=128), the `long double` variants of these C RTL math routines map to the X_FLOAT entry points.

7.2. Error Detection

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 mathematical functions:

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

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

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

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

main()
{
   double input, square_root;

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

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

If you did not check `errno` for this symbolic value, the `sqrt` function returns 0 when a negative number is entered. For more information about the `<errno.h>` header file, see Chapter 4.

7.3. The <fp.h> Header File

The `<fp.h>` header file implements some of the features defined by the Numerical C Extensions Group of the ANSI X3J11 committee. You might find this useful for applications that make extensive use of floating-point functions.

Some of the double-precision functions listed in this chapter return the value ±HUGE_VAL (defined in either `<math.h>` or `<fp.h>`) if the result is out of range. The `float` version of those functions return the value HUGE_VALF (defined only in `<fp.h>`) for the same conditions. The `long double` version returns the value HUGE_VALL (also defined in `<fp.h>`).

For programs compiled to enable IEEE infinity and NaN values, the values HUGE_VAL, HUGE_VALF, and HUGE_VALL are expressions, not compile-time constants. Initializations such as the following cause a compile-time error:

```
$ CREATE IEEE_INFINITY.C
#include <fp.h>

double my_huge_val = HUGE_VAL
^Z
$ CC /FLOAT=IEEE/IEEE=DENORM IEEE_INFINITY

double my_huge_val = HUGE_VAL;
.....................^
%CC-E-NEEDCONSTEXPR, In the initializer for my_huge_val,
 "decc$gt_dbl_infinity"
is not constant, but occurs in a context that requires a constant
 expression.
at line number 3 in file WORK1$:[RTL]IEEE_INFINITY.C;1
$
```

When using both <math.h> and <fp.h>, be aware that <math.h> defines a function isnan and <fp.h> defines a macro by the same name. Whichever header is included first in the application will resolve a reference to isnan.

7.4. Example

Example 7.1 shows how the tan, sin, and cos functions operate.

Example 7.1. Calculating and Verifying a Tangent Value

```
/*        CHAP_7_MATH_EXAMPLE.C                                    */

/* 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 <stdio.h>

/* This function calculates the tangent using the sin and         */
/* cos functions.                                                 */

double mytan(x)
    double x;
{
    double y,
           y1,
           y2;

    y1 = sin(x);
    y2 = cos(x);

    if (y2 == 0)
        y = 0;
    else
        y = y1 / y2;
```

```
    return y;
}
main()
{
    double x;

    /*  Print values: compare   */
    for (x = 0.0; x < 1.5; x += 0.1)
        printf("tan of %4.1f = %6.2f\t%6.2f\n", x, mytan(x), tan(x));
}
```

Example 7.1 produces the following output:

```
$ RUN  EXAMPLE
tan of  0.0 =   0.00      0.00
tan of  0.1 =   0.10      0.10
tan of  0.2 =   0.20      0.20
tan of  0.3 =   0.31      0.31
tan of  0.4 =   0.42      0.42
tan of  0.5 =   0.55      0.55
tan of  0.6 =   0.68      0.68
tan of  0.7 =   0.84      0.84
tan of  0.8 =   1.03      1.03
tan of  0.9 =   1.26      1.26
tan of  1.0 =   1.56      1.56
tan of  1.1 =   1.96      1.96
tan of  1.2 =   2.57      2.57
tan of  1.3 =   3.60      3.60
tan of  1.4 =   5.80      5.80
$
```

Chapter 8. Memory Allocation Functions

Table 8.1 lists and describes all the memory allocation functions found in the VSI C Run-Time Library (C RTL). For a more detailed description of each function, see the Reference Section.

Table 8.1. Memory Allocation Functions

Function	Description

`alloca`
`calloc`
`malloc`
	Allocate an area of memory.

`brk` `sbrk`	Determine the lowest virtual address that is not used with the program.
`cfree` `free`	Make available for reallocation the area allocated by a previous `calloc`, `malloc`, or `realloc` call.
`realloc`	Changes the size of the area pointed to by the first argument to the number of bytes given by the second argument.
`strdup`	Duplicates a string.

All C RTL functions requiring additional storage from the heap get that storage using the C RTL memory allocation functions `malloc`, `calloc`, `realloc`, `free`, and `cfree`. Memory allocated by these functions is quadword-aligned.

The ANSI C standard does not include `cfree`. For this reason, it is preferable to free memory using the functionally equivalent `free` function.

The `alloca` function allocates the memory on the stack. The memory is automatically freed when the function that calls `alloca` returns to its caller.

The `brk` and `sbrk` functions assume that memory can be allocated contiguously from the top of your address space. However, the `malloc` function and RMS may allocate space from this same address space. Do not use the `brk` and `sbrk` functions in conjunction with RMS and C RTL routines that use `malloc`.

Previous versions of the VAX C RTL documentation indicated that the memory allocation routines used the OpenVMS RTL functions LIB$GET_VM and LIB$FREE_VM to acquire and return dynamic memory. This is no longer the case; interaction between these routines and the C RTL memory allocation routines is no longer problematic (although LIB$SHOW_VM can no longer be used to track C RTL `malloc` and `free` usage).

The C RTL memory allocation functions `calloc`, `malloc`, `realloc`, and `free` are based on the LIB$ routines LIBVM_CALLOC, LIBVM_MALLOC, LIB$VM_REALLOC and LIB$VM_FREE, respectively.

The routines `VAXC$CALLOC_OPT`, `VAXC$CFREE_OPT`, `VAXC$FREE_OPT`, `VAXC$MALLOC_OPT`, and `VAXC$REALLOC_OPT` are now obsolete and should not be used in new development. However, versions of these routines that are equivalent to the standard C memory allocation routines are provided for backward compatibility.

8.1. Program Example

Example 8.1 shows the use of the `malloc`, `calloc`, and `free` functions.

Example 8.1. Allocating and Deallocating Memory for Structures

```
/*        CHAP_8_MEM_MANAGEMENT.C                                    */

/* This example takes lines of input from the terminal until    */
/* it encounters a Ctrl/Z, 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 the 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(void)
{
    char *buffer;

    /* Define pointers to        */
    /*  structure (input lines). */

    struct line_rec *first_line = NULL,
            *next_line,
            *last_line = NULL;

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

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

    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);
        }

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

        if (last_line == NULL)  /* Reset pointers.   */
            first_line = next_line;
        else
```

```
            last_line->next = next_line;

        last_line = next_line;
        /* Allocate space for the     */
        /*  next input line.          */
        buffer = malloc(MAX_LINE_LENGTH);

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

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

    exit(EXIT_SUCCESS);
}
```

The sample input and output for Example 8.1 are as follows:

```
$ RUN  EXAMPLE
line one
line two
Ctrl/Z
EXIT
line one
line two
$
```

Chapter 9. Shared Memory Functions

This chapter lists and describes all shared memory functions found in the VSI C Run-Time Library (C RTL), as well as current prerequisites, limitations, and restrictions for System V Shared Memory.

Table 9.1. Shared Memory Functions

Function	Description
`shmget`	Gets a shared memory segment.
`shmctl`	Shared memory control operations.
`shmat`	Shared memory attach operation.
`shmdt`	Shared memory detach operation.
`shm_open`	Opens a shared memory object.
`shm_unlink`	Removes a shared memory object.

9.1. System V Shared Memory Limitations

The following are the current limits in System V Shared Memory:

- Maximum number (SHMMNI) of System V Shared Memory segments allowed in a system is 1024.
- Maximum size (SHMMAX) of System V Shared Memory segment allowed is 512 MB.
- Minimum size (SHMMIN) of System V Shared Memory segment allowed is 1 byte.

Note

The global section mapped for System V Shared Memory segment is a multiple of 8 KB. For example, System V Shared Memory segment of size 1 byte will have an 8 KB global section.

9.2. Prerequisites

System V Shared Memory interfaces use file-backed global sections. These functions either create or delete the files based on request. The files are created in SYS$SPECIFIC:[DECC$SYSV_SHM]. The System Manager or Administrator must create the directory using the following command before using the Shared Memory Functions:

```
$ CREATE /DIRECTORY SYS$SPECIFIC:[DECC$SYSV_SHM]  -
  /OWNER_UIC = [SYSTEM]  -
  /PROTECTION = (S:RWE, O:RWE, G:RWE, W:RWE)
```

Note

If the SYS$SPECIFIC:[DECC$SYSV_SHM] directory does not exist, the directory SYS$SPECIFIC:[PSX$SEMAPHORES] will be used.

9.3. Restrictions

The following are the current restrictions:

- If an application passes its valid non-zero virtual address to the `shmat()` function, the current implementation expects this address to be a 32 bit virtual address; a 64 bit virtual address is not supported.
- System V Shared Memory Functions are implemented using file-backed global sections; hence each shared memory segment created would internally create a file to back the data for the section on system device (SYS$SYSDEVICE), and shared memory control operation with IPC_RMID deletes the file. So, the System Administrator must provide sufficient disk space on SYS$SYSDEVICE based on the usage of shared memory segments.
- Performance of shared memory functions depends on the size of segments so the application that uses these functions may experience delay when they are used for the first time. If the same segments are used frequently without IPC_RMID, then this delay will not be seen because with no IPC_RMID in use, the file will not be deleted, and thus will exist until it is deleted.
- Current release guarantees the initialization of shared memory segments with zero values only when the following conditions are true:
  - The High-Water Marking is enabled on the SYS$SYSDEVICE device.
  - The application is using $ERAPAT (erase pattern) system service, and the erase pattern generated for disk storage type is zero.

  In every other case, the initial contents of each shared memory segment created may or may not be initialized with zero values.

Note

For the following error cases, you must increase the value of the GBLSECTIONS SYSGEN parameter:

- The `shmget()` function returns ENOMEM (value=12) or ENOSPC (value=28).

  ENOMEM indicates 'Not enough core' and ENOSPC indicates no space left on device.
- The `shmat()` function returns ENOMEM (value=12).

Internally, shared memory uses global sections. Hence, a system having a large number of shared memory segments may result in exhaustion of GBLSECTIONS.

Chapter 10. System Functions

The C programming language is a good choice if you wish to write operating systems. For example, much of the UNIX operating system is written in C. When writing system programs, it is sometimes necessary to retrieve or modify the environment in which the program is running. This chapter describes the VSI C Run-Time Library (C RTL) functions that accomplish this and other system tasks.

Table 10.1 lists and describes all the system functions found in the C RTL. For a more detailed description of each function, see the Reference Section.

Table 10.1. System Functions

Function	Description
System Functions – Searching and Sorting Utilities	
`bsearch`	Performs a binary search on an array of sorted objects for a specified object.
`qsort` `qsort_r`	Sort an array of objects in place by implementing the quick-sort algorithm.
System Functions – Retrieving Process Information	
`ctermid`	Returns a character string giving the equivalence string of SYS$COMMAND, which is the name of the controlling terminal.
`cuserid`	Returns a pointer to a character string containing the name of the user who initiated the current process.
`getcwd`	Returns a pointer to the file specification for the current working directory.

`getegid`
`geteuid`
`getgid`
`getuid`
	Return, in OpenVMS terms, group and member numbers from the user-identification code (UIC).

`getenv`	Searches the environment array for the current process and returns the value associated with a specified environment.
`getlogin`	Gets the login name of the user associated with the current session.
`getpid`	Returns the process ID of the current process.
`getppid`	Returns the parent process ID of the calling process.
`getpwnam`	Accesses user-name information in the user database.
`getpwuid`	Accesses user-ID information in the user database.
System Functions – Changing Process Information	
`chdir`	Changes the default directory.
`chmod`	Changes the file protection of a file.
`chown`	Changes the owner user identification code (UIC) of a file.
`mkdir`	Creates a directory.
`nice`	Increases or decreases the process priority to the process base priority by the amount of the argument.
`putenv`	Sets an environmental variable.

Function	Description
setenv	Inserts or resets the environment variable name in the current environment list.
setgid setuid	Implemented for program portability and have no functionality.
sleep usleep	Suspend the execution of the current process for at least the number of seconds indicated by its argument.
umask	Creates a file protection mask that is used whenever a new file is created. It returns the old mask value.
System Functions – Retrieving and Converting Date/Time Information	
asctime	Converts a broken-down time into a 26-character string.
clock	Determines the CPU time, in microseconds, used since the beginning of the program execution.
clock_getres	Gets the resolution for the specified clock.
clock_gettime	Returns the current time (in seconds and nanoseconds) for the specified clock.
clock_settime	Sets the specified clock.
ctime	Converts a time, in seconds, to an ASCII string in the form generated by the asctime function.
decc$fix_time	Converts OpenVMS binary system times to UNIX binary times.
difftime	Computes the difference, in seconds, between the two times specified by its arguments.
ftime	Returns the elapsed time since 00:00:00, January 1, 1970, in the structure timeb.
getclock	Gets the current value of the systemwide clock.
getdate	Converts a formatted string to a time/date structure.
getitimer	Returns the value of interval timers.
gettimeofday	Gets the date and time.
gmtime	Converts time units to broken-down UTC time.
localtime	Converts a time (expressed as the number of seconds elapsed since 00:00:00, January 1, 1970) into hours, minutes, seconds, and so on.
mktime	Converts a local-time structure into time since the Epoch.
nanosleep	High-resolution sleep (REALTIME). Suspends a process from execution for the specified timer interval.
setitimer	Sets the value of interval timers.
strftime wcsftime	Place characters into an array, as controlled by a specified format string.
strptime	Converts a character string into date and time values.
time	Returns the time elapsed since 00:00:00, January 1, 1970, in seconds.
times	Returns the accumulated times of the current process and of its terminated child processes.
tzset	Sets and accesses time-zone conversion.

Function	Description
ualarm	Sets or changes the timeout of interval timers.
wcsftime	Uses date and time information stored in a tm structure to create a wide-character output string.
System Function – Miscellaneous	
VAXC$CRTL_INIT	Initializes the run-time environment and establishes an exit and condition handler, which makes it possible for C RTL functions to be called from other languages.

Example 10.1 shows how the cuserid function is used.

Example 10.1. Accessing the User Name

```
/*        CHAP_9_GET_USER.C                                    */

/* Using cuserid, this program returns the user name.         */

#include <stdio.h>

main()
{
    static char string[L_cuserid];

    cuserid(string);
    printf("Initiating user: %s\n", string);
}
```

If a user named TOLLIVER runs the program, the following is displayed on stdout:

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

Example 10.2 shows how the getenv function is used.

Example 10.2. Accessing Terminal Information

```
/*        CHAP_9_GETTERM.C                                 */

/* Using getenv, this program returns the terminal.     */

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

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

Running Example 10.2 on a VT100 terminal in 132-column mode displays the following:

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

Example 10.3 shows how to use getenv to find the user's default login directory and how to use chdir to change to that directory.

Example 10.3. Manipulating the Default Directory

```
/*       CHAP_9_CHANGE_DIR.C                                    */

/* This program performs the equivalent of the DCL command      */
/* SET DEFAULT SYS$LOGIN. However, once the program exits, the  */
/* directory is reset to the directory from which the program   */
/* was run.                                                     */

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

main()
{
    char *dir;
    int i;

    dir = getenv("HOME");
    if ((i = chdir(dir)) != 0) {
        perror("Cannot set directory");
        exit(0);
    }

    printf("Current directory: %s\n", dir);
}
```

Running Example 10.3 displays the following:

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

Example 10.4 shows how to use the `time`, `localtime`, and `strftime` functions to print the correct date and time at the terminal.

Example 10.4. Printing the Date and Time

```
/*        CHAP_9_DATE_TIME.C                                    */

/* The time function returns the time in seconds; the localtime */
/* function converts the time to hours, minutes, and so on;     */
/* the strftime function uses these values to obtain a string   */
/* in the desired format.                                       */

#include <time.h>
#include <stdio.h>

#define MAX_STRING 80

main()
{
    struct tm *time_structure;
    time_t time_val;
    char output_str[MAX_STRING];

    time(&time_val);
    time_structure = localtime(&time_val);
```

```
    /* Print the date  */

    strftime(output_str, MAX_STRING,
             "Today is %A, %B %d, %Y", time_structure);

    printf("%s\n", output_str);

    /* Print the time using a 12-hour clock format. */

    strftime(output_str, MAX_STRING,
             "The time is %I:%M %p", time_structure);

    printf("%s\n", output_str);
}
```

Running Example 10.4 displays the following:

```
$ RUN  EXAMPLE5
Today is Thursday, May 20, 1993
The time is 10:18 AM
$
```

Chapter 11. Developing International Software

This chapter describes typical features of international software and the features provided with the VSI C Run-Time Library (C RTL) that enable you to design and implement international software.

See the Reference Section for more detailed information on the functions described in this chapter.

11.1. Internationalization Support

The C RTL has added capabilities to allow application developers to create international software. The C RTL obtains information about a language and a culture by reading this information from *locale* files.

11.1.1. Installation

If you are using these C RTL capabilities, you must install a separate kit to provide these files to your system. See the appendix "Installing OpenVMS Internationalization data kit" in the *OpenVMS Upgrade and Installation Guide*.

On OpenVMS Alpha systems the save set is provided on the Layered Product CD, and is named VMSI18N0nn or ALPVMSI18N0n_07nn.

To install this save set, follow the standard OpenVMS installation procedures using this save-set name as the name of the kit. There are several categories of locales that you can select to install. You can select as many locales as you need by answering the following prompts:

```
* Do you want European and US support? [YES]?
* Do you want Chinese GB18030 support (locale and Unicode converters)
 [YES]?
* Do you want Chinese support? [YES]?
* Do you want Japanese support? [YES]?
* Do you want Korean support? [YES]?
* Do you want Thai support? [YES]?
* Do you want the Unicode converters? [YES]?
```

This kit also has an Installation Verification Procedure that we recommend you run to verify the correct installation of the kit.

11.1.2. Unicode Support

In OpenVMS Version 7.2, the VSI C Run-Time Library added the Universal Unicode locale, which is distributed with the OpenVMS system, not with the VMSI18N0nn kit. The name of the Unicode locale is:

```
UTF8-20
```

Like those locales shipped with the VMSI18N0nn kit, the Unicode locale is located at the standard location referred to by the SYS$I18N_LOCALE logical name.

The UTF8-20 Unicode is based on Unicode standard Version V2.0. The Unicode locale uses UCS-4 as wide-character encoding and UTF-8 as multibyte character encodings.

C RTL also includes converters that perform conversions between Unicode and any other supported character sets. The expanded set of converters includes converters for UCS-2, UCS-4, and UTF-8

Unicode encoding. The Unicode converters can be used by the ICONV CONVERT utility and by the `iconv` family of functions in the VSI C Run-Time Library.

In OpenVMS Version 7.2, the VSI C Run-Time Library added Unicode character set converters for Microsoft Code Page 437.

11.2. Features of International Software

International software is software that can support multiple languages and cultures. An international program should be able to:

- Display messages in the user's own language. This includes screen displays, error messages, and prompts.
- Handle culture-specific information such as:
  - Date and time formatting

    The conventions for representing dates and times vary from one country to another. For example, in the US the month is given first; in the UK the day is specified first. Therefore, the date 12/5/1993 is interpreted as December 5, 1993 in the US, and as May 12, 1993 in the UK.
  - Numeric formatting

    The character that represents the decimal point (the radix character) and the thousands separator character vary from one country to another. For example, in the UK the period (.) is used to represent the radix character, and the comma is used as a separator. However, in Germany, the comma is used as the radix character and the period is the separator character. Therefore, the number 2,345.67 in the UK is the same as 2.345,67 in Germany.
  - Monetary formatting

    Currency values are represented by different symbols and can be formatted using a variety of separator characters, depending on the currency.
- Handle different coded character sets (not just ASCII).
- Handle a mixture of single and multibyte characters.
- Provide multipass string comparisons.

  String comparison functions such as `strcmp` compare strings by comparing the code point values of the characters in the strings. However, some languages require more complex comparisons to correctly sort strings.

To meet the previous requirements, an application should not make any assumptions about the language, local customs, or the coded character set used. All this localization data should be defined separately from the program, and only bound to it at run time.

The rest of this chapter describes how you can create international software using VSI C.

11.3. Developing International Software Using VSI C

The VSI C environment provides the following facilities to create international software:

- A method for separating localization data from a program.

  Localization data is held in a database known as a *locale*. This stores all the language and culture information required by a program. See Section 11.4 for details of the structure of locales.

  A program specifies what locales to use by calling the `setlocale` function. See Section 11.5 for more information.

- A method of separating message text from the program source.

  This is achieved using *message catalogs* that store all the messages for an application. The message catalog is linked to the application at run time. This means that the messages can be translated into different languages and then the required language version is selected at run time. See Section 11.6.

- C RTL functions that are sensitive to localization data.

  The C RTL includes functions for:

  - Converting between different codesets. See Section 11.7.
  - Handling culture-specific information. See Section 11.8.
  - Multipass string collation. See Section 11.10.

- A special wide-character data type defined in the C RTL makes it easier to handle codesets that have a mixture of single and multibyte characters. A set of functions is also defined to support this wide-character data type. See Section 11.9.

11.4. Locales

A locale consists of different categories, each of which determines one aspect of the international environment. Table 11.1 lists the categories in a locale and describes the information in each.

Table 11.1. Locale Categories

Category	Description
LC_COLLATE	Contains information about collating sequences.
LC_CTYPE	Contains information about character classification.
LC_MESSAGES	Defines the answers that are expected in response to yes/no prompts.
LC_MONETARY	Contains monetary formatting information.
LC_NUMERIC	Contains information about formatting numbers.
LC_TIME	Contains time and date information.

The locales provided reside in the directory defined by the SYS$I18N_LOCALE logical name. The file-naming convention for locales is:

```
language_country_codeset.locale
```

Where:

- *language* is the mnemonic for the language. For example, EN indicates an English locale.
- *country* is the mnemonic for the country. For example, GB indicates a British locale.

- *codeset* is the name of the ISO standard codeset for the locale. For example, ISO8859-1 is the ISO 8859 codeset for the Western European languages. See Section 11.7 for more information about the codesets supported.

11.5. Using the setlocale Function to Set Up an International Environment

An application sets up its international environment at run time by calling the `setlocale` function. The international environment is set up in one of two ways:

- The environment is defined by one locale. In this case, each of the locale categories is defined by the same locale.
- Categories are defined separately. This lets you define a mixed environment that uses different locales depending on the operation performed. For example, if an English user has some Spanish files that are to be processed by an application, the LC_COLLATE category could be defined by a Spanish locale while the other categories are defined by an English locale. To do this you would call `setlocale` once for each category.

The syntax for the `setlocale` function is:

```
char *setlocale(int category, const char *locale)
```

Where:

- *category* is either the name of a category, or LC_ALL. Specifying LC_ALL means that all the categories are defined by the same locale. Specify a category name to set up a mixed environment.
- *locale* is one of the following:
  - The name of the locale to use.

    If you want users to specify the locale interactively, your application could prompt the user for a locale name, and then pass the name as an argument to the `setlocale` function. A locale name has the following format:

    ```
    language_country.codeset[@modifier]
    ```

    For example, `setlocale(LC_COLLATE, "en_US.ISO8859-1")` selects the locale en_US.ISO8859-1 for the LC_COLLATE category.
  - ""

    This causes the function to use logical names to determine the locale for the category specified. See the section called “Specifying the Locale Using Logical Names” for details.

If an application does not call the `setlocale` function, the default locale is the C locale. This allows such applications to call those functions that use information in the current locale.

Specifying the Locale Using Logical Names

If the `setlocale` function is called with "" as the *locale* argument, the function checks for a number of logical names to determine the locale name for the category specified.

There are a number of logical names that users can set up to define their international environment:

- Logical name corresponding to a category

  For example, the LC_NUMERIC logical name defines the locale associated with the LC_NUMERIC category within the user's environment.

- LC_ALL

- LANG

  The LANG logical name defines the user's language.

In addition to the logical names defined by a user, there are a number of systemwide logical names, set up during system startup, that define the default international environment for all users on a system:

- SYS$ *category*

  Where *category* is the name of a category. This specifies the system default for that category.

- SYS$LC_ALL

- SYS$LANG

The `setlocale` function checks for user-defined logical names first, and if these are not defined, it checks the system logical names.

11.6. Using Message Catalogs

An important requirement for international software is that it should be able to communicate with the user in the user's own language. The messaging system enables program messages to be created separately from the program source, and linked to the program at run time.

Messages are defined in a message text source file, and compiled into a message catalog using the GENCAT command. The message catalog is accessed by a program using the functions provided in the C RTL.

The functions provided to access the messages in a catalog are:

- The `catopen` function, which opens a specified catalog ready for use.

- The `catgets` function, which enables the program to read a specific message from a catalog.

- The `catclose` function, which closes a specified catalog. Open message catalogs are also closed by the `exit` function.

For information on generating message catalogs, see the GENCAT command description in the OpenVMS system documentation.

11.7. Handling Different Character Sets

The C RTL supports a number of state-independent codesets and codeset encoding schemes that contain the ASCII encoded Portable Character Set. It does not support state-dependent codesets. The codesets supported are:

- ISO8859- *n*

  where *n* = 1,2,5,7,8 or 9. This covers codesets for North America, Europe (West and East), Israel, and Turkey.

- eucJP, SJIS, DECKANJI, SDECKANJI: Codesets used in Japan.

- eucTW, DECHANYU, BIG5, DECHANZI: Chinese codesets used in China (PRC), Hong-Kong, and Taiwan.

- DECKOREAN: Codeset used in Korea.

11.7.1. Charmap File

The characters in a codeset are defined in a charmap file. The charmap files supplied by VSI are located in the directory defined by the SYS$I18N_LOCALE logical name. The file type for a charmap file is .CMAP.

11.7.2. Converter Functions

As well as supporting different coded character sets, the C RTL provides the following converter functions that enable you to convert characters from one codeset to another:

- `iconv_open`—Specifies the type of conversion. It allocates a conversion descriptor required by the `iconv` function.

- `iconv`—Converts characters in a file to the equivalent characters in a different codeset. The converted characters are stored in a separate file.

- `iconv_close`—Deallocates a conversion descriptor and the resources allocated to the descriptor.

11.7.3. Using Codeset Converter Files

The file-naming convention for codeset converters is:

fromcode_tocode`.iconv`

Where *fromcode* is the name of the source codeset, and *tocode* is the name of the codeset to which characters are converted.

You can add codeset converters to a given system by installing the converter files in the directory pointed by the logical name SYS$I18N_ICONV.

Codeset converter files can be implemented either as table-based conversion files or as algorithm-based converter files created as OpenVMS shareable images.

Creating a Table-Based Conversion File

The following summarizes the necessary steps to create a table-based codeset converter file:

1. Create a text file that describes the mapping between any character from the source codeset to the target codeset. For the format of this file, see the DCL command ICONV COMPILE in the OpenVMS *New Features Manual*, which processes such a file and creates a codeset converter table file.

2. Copy the resulting file from the previous step to the directory pointed by the logical SYS$I18N_ICONV, assuming you have the privilege to do so.

Creating an Algorithm-Based Conversion File

To create an algorithm-based codeset converter file implemented as a shareable image, follow these steps:

1. Create C source files that implement the codeset converter. The API is documented in the public header file `<iconv.h>` as follows:

   - The universal entry point `_u_iconv_open` is called by the C RTL routine `iconv_open` to initialize a conversion.
   - `_u_iconv_open` returns to `iconv_open` a pointer to the structure `__iconv_extern_obj_t`.
   - Within this structure, the converter exports its own conversion entry point and conversion close routine, which are called by the C RTL routines `iconv` and `iconv_close`, respectively.
   - The major and minor identifier fields are required by `iconv_open` to test for a possible mismatch between the library and the converter. The converter usually assigns the constants __ICONV_MAJOR and __ICONV_MINOR, defined in the `<iconv.h>` header file.
   - The field *tcs_mb_cur_max* is used only by the DCL command ICONV CONVERT to optimize its buffer usage. This field reflects the maximum number of bytes that comprise a single character in the target codeset, including the shift sequence (if any).

2. Compile and link the modules that comprise the codeset converter as an OpenVMS shareable image, making sure that the filename adheres to the preceding conventions.

3. Copy the resulting file from the previous step to the directory pointed by the logical SYS$I18N_ICONV, assuming you have the privilege to do so.

Some Final Notes

By default, SYS$I18N_ICONV is a search list where the first directory in the list SYS$SYSROOT:[SYS$I18N.ICONV.USER] is meant for use as a site-specific repository for `iconv` codeset converters.

The number of codesets and locales installed vary from system to system. Check the SYS$I18N directory tree for the codesets, converters, and locales installed on your system.

11.8. Handling Culture-Specific Information

Each locale contains the following cultural information:

- Date and time information

  The LC_TIME category defines the conventions for writing date and time, the names of the days of the week, and the names of months of the year.

- Numeric information

  The LC_NUMERIC category defines the conventions for formatting non-monetary values.

- Monetary information

  The LC_MONETARY category defines currency symbols and the conventions used to format monetary values.

- Yes and no responses

  The LC_MESSAGES category defines the strings expected in response to yes/no questions.

You can extract some of this cultural information using the `nl_langinfo` function and the `localeconv` function. See Section 11.8.1.

11.8.1. Extracting Cultural Information From a Locale

The `nl_langinfo` function returns a pointer to a string that contains an item of information obtained from the program's current locale. The information you can extract from the locale is:

- Date and time formats
- The names of the days of the week, and months of the year in the local language
- The radix character
- The character used to separate groups of digits in non-monetary values
- The currency symbol
- The name of the codeset for the locale
- The strings defined for responses to yes/no questions

The `localeconv` function returns a pointer to a data structure that contains numeric formatting and monetary formatting data from the LC_NUMERIC and LC_MONETARY categories.

11.8.2. Date and Time Formatting Functions

The functions that use the date and time information are:

- `strftime`—Takes date and time values stored in a data structure and formats them into an output string. The format of the output string is controlled by a format string.
- `strptime`—Converts a string (of type `char`) into date and time values. A format string defines how the string is interpreted.
- `wcsftime`—Does the same as `strftime` except that it creates a wide-character string.

11.8.3. Monetary Formatting Function

The `strfmon` function uses the monetary information in a locale to convert a number of values into a string. The format of the string is controlled by a format string.

11.8.4. Numeric Formatting

The information in LC_NUMERIC is used by various functions. For example, `strtod`, `wcstod`, and the print and scan functions determine the radix character from the LC_NUMERIC category.

11.9. Functions for Handling Wide Characters

A character can be represented by single-byte or multibyte values depending on the codeset. To make it easier to handle both single-byte and multibyte characters in the same way, the C RTL defines a wide-character data type, `wchar_t`. This data type can store characters that are represented by 1-, 2-, 3-, or 4-byte values.

The functions provided to support wide characters are:

- Character classification functions. See Section 11.9.1.
- Case conversion functions. See Section 11.9.2.
- Input and output functions. See Section 11.9.3.
- Multibyte to wide-character conversion functions. See Section 11.9.4.
- Wide-character to multibyte conversion functions. See Section 11.9.4.
- Wide-character string manipulation functions. See Section 11.9.5.
- Wide-character string collation and comparison functions. See Section 11.10.

11.9.1. Character Classification Functions

The LC_CTYPE category in a locale classifies the characters in the locale's codeset into different types (alphabetic, numeric, lowercase, uppercase, and so on). There are two sets of functions, one for wide characters and one for single-byte characters, that test whether a character is of a specific type. The `is*` functions test single-byte characters, and the `isw*` functions test wide characters.

For example, the `iswalnum` function tests if a wide character is classed as either alphabetic or numeric. It returns a nonzero value if the character is one of these types. For more information about the classification functions, see Chapter 3 and the Reference Section.

11.9.2. Case Conversion Functions

The LC_CTYPE category defines mapping between pairs of characters of the locale. The most common character mapping is between uppercase and lowercase characters. However, a locale can support more than just case mappings.

Two functions are provided to map one character to another according to the information in the LC_CTYPE category of the locale:

- `wctrans`—Looks for the named mapping (predefined in the locale) between characters.
- `towctrans`—Maps one character to another according to the named mapping given to the `wctrans` function.

Two functions are provided for character case mapping:

- `towlower`—Maps an uppercase wide character to its lowercase equivalent.
- `towupper`—Maps a lowercase wide character to its uppercase equivalent.

For more information about these functions, see the Reference Section.

11.9.3. Functions for Input and Output of Wide Characters

The set of input and output functions manages wide characters and wide-character strings.

Read Functions

The functions for reading wide characters and wide-character strings are `fgetwc`, `fgetws`, `getwc`, and `getwchar`.

There is also an `ungetwc` function that pushes a wide character back into the input stream.

Write Functions

The functions for writing wide characters and wide-character strings are `fputwc`, `fputws`, `putwc`, and `putwchar`.

Scan Functions

All the scan functions allow for a culture-specific radix character, as defined in the LC_NUMERIC category of the current locale.

The %lc, %C, %ls, and %S conversion specifiers enable the scan functions `fwscanf`, `wscanf`, `swscanf`, `fscanf`, `scanf`, and `sscanf` to read in wide characters.

Print Functions

All the print functions can format numeric values according to the data in the LC_NUMERIC category of the current locale.

The %lc, %C and %ls, %S conversion specifiers used with print functions convert wide characters to multibyte characters and print the resulting characters.

See Chapter 2 for details of all input and output functions.

11.9.4. Functions for Converting Multibyte and Wide Characters

Wide characters are used internally by an application to manage single-byte or multibyte characters. However, text files are generally stored in multibyte character format. To process these files, the multibyte characters need converting to wide-character format. This can be achieved using the following functions:

- `mbtowc`, `mbrtowc`, `btowc`—Convert one multibyte character to a wide character.
- `mbsrtowcs`, `mbstowcs`—Convert a multibyte character string to a wide-character string.

Similarly, the following functions convert wide characters into their multibyte equivalent:

- `wcrtomb`, `wctomb`, `wctob`—Convert a single wide character to a multibyte character.
- `wcsrtombs`, `wcstombs`—Convert a wide-character string to a multibyte character string.

Associated with these conversion functions, the `mblen` and `mbrlen` functions are used to determine the size of a multibyte character.

Several of the wide-character functions take an argument of type "pointer to `mbstate_t`", where `mbstate_t` is an opaque datatype (like `FILE` or `fpos_t`) intended to keep the conversion state for the state-dependent codesets.

11.9.5. Functions for Manipulating Wide-Character Strings and Arrays

The C RTL contains a set of functions (the `wcs*` and `wmem*` functions) that manipulate wide-character strings. For example, the `wcscat` function appends a wide-character string to the end of another string in the same way that the `strcat` function works on character strings of type *char*.

See Chapter 3 for details of the string manipulation functions.

11.10. Collating Functions

In an international environment, string comparison functions need to allow for multipass collations. The collation requirements include:

- Ordering accented characters.
- Collating a character sequence as a single character. For example, ch in Spanish should be collated after c but before d.
- Collating a single character as a two-character sequence.
- Ignoring some characters.

Collating information is stored in the LC_COLLATE category of a locale. The C RTL includes the `strcoll` and `wcscoll` functions that use this collating information to compare two strings.

Multipass collations by `strcoll` or `wcscoll` can be slower than using the `strcmp` or `wcscmp` functions. If your program needs to do many string comparisons using `strcoll` or `wcscoll`, it may be quicker to transform the strings once, using the `strxfrm` or `wcsxfrm` function, and then use the `strcmp` or `wcscmp` function.

The term *collation* refers to the relative order of characters. The collation order is locale-specific and might ignore some characters. For example, an American dictionary ignores the hyphen in words and lists *take-out* between *takeoff* and *takeover*.

Comparison, on the other hand, refers to the examination of characters for sameness or difference. For example, *takeout* and *take-out* are different words, although they may collate the same.

Suppose an application sorts a list of words so it can later perform a binary search on the list to quickly retrieve a word. Using `strcmp`, *take-in*, *take-out*, and *take-up* would be grouped in one part of the table. Using `strcoll` and a locale that ignores hyphens, *take-out* would be grouped with *takeoff* and *takeover*, and would be considered a duplicate of *takeout*. To avoid a binary search finding *takeout* as a duplicate of *take-out*, an application would most likely use `strcmp` rather than `strcoll` for forming a binary tree.

Chapter 12. Date/Time Functions

This chapter describes the date/time functions available with VSI C for OpenVMS Systems. For more detailed information on each function, see the Reference Section.

Table 12.1. Date/Time Functions

Function	Description
`asctime`	Converts a broken-down time from `localtime` into a 26-character string.
`ctime`	Converts a time, in seconds, since 00:00:00, January 1, 1970 to an ASCII string of the form generated by the `asctime` function.
`ftime`	Returns the elapsed time since 00:00:00, January 1, 1970 in the structure pointed to by its argument.
`getclock`	Gets the current value of the systemwide clock.
`gettimeofday`	Gets the date and time.
`gmtime`	Converts time units to GMT (Greenwich Mean Time).
`localtime`	Converts a time (expressed as the number of seconds elapsed since 00:00:00, January 1, 1970) into hours, minutes, seconds, and so on.
`mktime`	Converts a local time structure to a calendar time value.
`time`	Returns the time elapsed since 00:00:00, January 1, 1970, in seconds.
`tzset`	Sets and accesses time-zone conversion.

Also, the time-related information returned by `fstat` and `stat` uses the new date/time model described in Section 12.1.

12.1. Date/Time Support Models

Beginning with OpenVMS Version 7.0, the C RTL changed its date/time support model from one based on local time to one based on Universal Coordinated Time (UTC). This allows the C RTL to implement ANSI C/POSIX functionality that previously could not be implemented. A UTC time-based model also makes the C RTL compatible with the behavior of the Tru64 UNIX time functions.

By default, newly compiled programs will generate entry points into UTC-based date/time routines.

For compatibility with OpenVMS systems prior to Version 7.0, previously compiled programs that relink on an OpenVMS Version 7.0 system will retain local-time-based date/time support. Relinking alone will not access UTC support.

Compiling programs with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined will also enable local-time-based entry points. That is, the new OpenVMS Version 7.0 date/time functions will not be enabled.

Functions with both UTC-based and local-time-based entry points are:

```
ctime           mktime
fstat           stat
ftime           strftime
gmtime          time
```

```
localtime       wcsftime
```

Note

Introducing a UTC-based, date/time model implies a certain loss of performance because time-related functions supporting UTC must read and interpret time-zone files instead of doing simple computations in memory as was done for the date/time model based on local time.

To decrease this performance degradation, OpenVMS Version 7.1 and higher can maintain the processwide cache of time-zone files. The size of the cache (that is, the number of files in the memory) is determined by the value of the DECC$TZ_CACHE_SIZE logical name. The default value is 2.

Because the time-zone files are relatively small (about 3 blocks each), consider defining DECC$TZ_CACHE_SIZE as the maximum number of time zones used by the application. For example, the default cache size fits an application that does not switch time zones during the run and runs on a system where the `TZ` environment variable is defined with both Standard and Summer time zone.

12.2. Overview of Date/Time Functions

In the UTC-based model, times are represented as seconds since the Epoch. The Epoch is defined as the time 0 hours, 0 minutes, 0 seconds, January 1, 1970 UTC. Seconds since the Epoch is a value interpreted as the number of seconds between a specified time and the Epoch.

The functions `time` and `ftime` return the time as seconds since the Epoch.

The functions `ctime`, `gmtime`, and `localtime` take as their argument a time value that represents the time in seconds from the Epoch.

The function `mktime` converts a broken-down time, expressed as local time, into a time value in terms of seconds since the Epoch.

The values `st_ctime`, `st_atime`, and `st_mtime` returned in the `stat` structure by the `stat` and `fstat` functions are also in terms of UTC.

Time support new to OpenVMS Version 7.0 includes the functions `tzset`, `gettimeofday`, and `getclock`, and the external variables `tzname`, `timezone`, and `daylight`.

The UTC-based time model enables the C RTL to:

- Implement the ANSI C `gmtime` function, which returns a structure in terms of GMT time.
- Specify the ANSI `tm_isdst` field of the `tm` structure, which specifies whether daylight savings time is in effect.
- Provide time-related POSIX and X/Open extensions (such as the `tzset` function (which lets you get time information from any time zone), and the external variables `tzname`, `timezone`, and `daylight`.
- Correctly compute the local time for times in the past, something that the time functions like `localtime` need to do.
- Enable `localtime` and `gmtime`, through the use of feature-test macros (see Section 1.4), to return two additional fields: `tm_zone` (an abbreviation of the time-zone name) and `tm_gmtoff` (the offset from UTC in seconds) in the `tm` structure they return.

12.3. C RTL Date/Time Computations – UTC and Local Time

Universal Coordinated Time (UTC) is an international standard for measuring time of day. Under the UTC time standard, zero hour occurs when the Greenwich Meridian is at midnight. UTC has the advantage of always increasing, unlike local time, which can go backwards/forwards depending on daylight saving time.

Also, UTC has two additional components:

- A measure of inaccuracy (optional)
- A time-differential factor, which is an offset applied to UTC to derive local time.

  The time-differential factor associates each local time zone with UTC; the time differential factor is applied to UTC to derive local time. (Local times can vary up to –12 hours West of the Greenwich Meridian and +13 hours East of it).

For the C RTL time support to work correctly on OpenVMS Version 7.0 and higher, the following must be in place:

- Your OpenVMS system must be correctly configured to use a valid OpenVMS TDF. Make sure this is set correctly by checking the value of the SYS$TIMEZONE_DIFFERENTIAL logical. This logical should contain the time difference added to UTC to arrive at your local time.
- Your OpenVMS installation must correctly set the local time zone that describes the location that you want to be your default local time zone. In general, this is the local time zone in which your system is running.

For more information, see the section on setting up your system to compensate for different time zones in your *OpenVMS System Manager's Manual: Essentials*.

The C RTL uses local time-zone conversion rules to compute local time from UTC, as follows:

1. The C RTL internally computes time in terms of UTC.
2. The C RTL then uses time-zone conversion rules to compute a time-differential factor to apply to UTC to derive local time. See the `tzset` function in the Reference Section of this manual for more information on the time-zone conversion rules.

By default, the time-zone conversion rules used for computing local time from UTC are specified in time-zone files defined by the SYS$LOCALTIME and SYS$POSIXRULES system logicals. These logicals are set during an OpenVMS installation to point to time-zone files that represent the system's best approximation to local wall-clock time:

- SYS$LOCALTIME defines the time-zone file containing the default conversion rules used by the C RTL to compute local time.
- SYS$POSIXRULES defines the time-zone file that specifies the default rules to be applied to POSIX style time zones that do not specify when to change to summer time and back.

SYS$POSIXRULES can be the same as SYS$LOCALTIME. See the `tzset` function for more information.

12.4. Time-Zone Conversion Rule Files

The time-zone files pointed to by the SYS$LOCALTIME and SYS$POSIXRULES logicals are part of a public-domain, time-zone support package installed on OpenVMS Version 7.0 and higher systems.

This support package includes a series of source files that describe the time-zone conversion rules for computing local time from UTC in worldwide time zones. OpenVMS Version 7.0 and higher systems provide a time-zone compiler called ZIC. The ZIC compiler compiles time-zone source files into binary files that the C RTL reads to acquire time-zone conversion specifications. For more information on the format of these source files, see the OpenVMS system documentation for ZIC.

The time-zone files are organized as follows:

- The root time-zone directory is SYS$COMMON:[SYS$TIMEZONE.SYSTEM]. The system logical SYS$TZDIR is set during installation to point to this area.
- Time-zone source files are found in SYS$COMMON:[SYS$TIMEZONE.SYSTEM.SOURCES].
- Binary time-zone files use SYS$COMMON:[SYS$TIMEZONE.SYSTEM] as their root directory. Some binaries reside in this directory while others reside in its subdirectories.
- Binaries residing in subdirectories are time-zone files that represent specific time zones in a larger geographic area. For example, SYS$COMMON:[SYS$TIMEZONE.SYSTEM] contains a subdirectory for the United States and a subdirectory for Canada, because each of these geographic locations contains several time zones. Each time zone in the US is represented by a time-zone file in the Unites States subdirectory. Each time zone in Canada is represented by a time-zone file in the Canada subdirectory.

Several of the time-zone files have names based on acronyms for the areas that they represent. Table 12.2 lists these acronyms.

Table 12.2. Time-zone Filename Acronyms

Time-Zone Acronym	Description
CET	Central European Time
EET	Eastern European Time
Factory	Specifies No Time Zone
GB-Eire	Great Britain/Ireland
GMT	Greenwich Mean Time
NZ	New Zealand
NZ-CHAT	New Zealand, Chatham Islands
MET	Middle European Time
PRC	Peoples Republic of China
ROC	Republic of China
ROK	Republic of Korea
SystemV	Specific to System V operating system
UCT	Universal Coordinated Time
US	United States

Time-Zone Acronym	Description
UTC	Universal Coordinated Time
Universal	Universal Coordinated Time
W-SU	Middle European Time
WET	Western European Time

A mechanism is available for you to define and implement your own time-zone rules. For more information, see the OpenVMS system documentation on the ZIC compiler and the description of `tzset` in the reference section of this manual.

Also, the SYS$LOCALTIME and SYS$POSIXRULES system logicals can be redefined to user-supplied time zones.

12.5. Sample Date/Time Scenario

The following example and explanation shows how to use the C RTL time functions to print the current time:

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

main ()
{
  time_t t;

  t = time((time_t)0);
  printf ("The current time is: %s\n",asctime (localtime (&t)));
}
```

This example:

1. Calls the `time` function to get the current time in seconds since the Epoch, in terms of UTC.
2. Passes this value to the `localtime` function, which uses time-conversion information as specified by `tzset` to determine which time-zone conversion rules should be used to compute local time from UTC. By default, these rules are specified in the file defined by SYS$LOCALTIME:
   a. For a user in the Eastern United States interested in their local time, SYS$LOCALTIME would be defined during installation to SYS$COMMON:[SYS$ZONEINFO.US]EASTERN, the time-zone file containing conversion rules for the Eastern U.S. time zone.
   b. If the local time falls during daylight savings time (DST), SYS$COMMON:[SYS$ZONEINFO.US]EASTERN indicates that a time differential factor of -4 hours needs to be applied to UTC to get local time.

      If the local time falls during Eastern standard time (EST), SYS$COMMON:[SYS$ZONEINFO.US]EASTERN indicates that a time differential factor of -5 hours needs to be applied to UTC to get local time.
   c. The C RTL applies -4 (DST) or -5 (EST) to UTC, and `localtime` returns the local time in terms of a `tm` structure.
3. Pass this `tm` structure to the `asctime` function to print the local time in a readable format.

Chapter 13. Symbolic Links and POSIX Pathname Support

OpenVMS Version 8.3 and higher provides Open Group compliant symbolic link support and POSIX pathname processing support. This support will help partners and customers who port UNIX and LINUX applications to OpenVMS or who use a UNIX style development environment to reduce the application development costs and complexity previously associated with such porting efforts.

The following OpenVMS features are provided to support symbolic links and POSIX pathname processing:

- The Open Group compliant symbolic-link functions `symlink`, `readlink`, `unlink`, `realpath`, `lchown` and `lstat` are added to the C Run-Time Library (C RTL) (see Section 13.3.3).
- Existing C RTL functions such as `creat`, `open`, `delete`, and `remove`, now behave in accordance with Open Group specifications for symbolic links (see Section 13.3.4).
- RMS allows the C RTL to implement the above-mentioned functions. RMS routines such as SYS$OPEN, SYS$CREATE, SYS$PARSE, and SYS$SEARCH now support symbolic links (see Section 13.4).
- The contents of symbolic links on OpenVMS are interpreted as POSIX pathnames when encountered during pathwalks and searches. POSIX pathnames are now supported in OpenVMS and are usable through C RTL and RMS interfaces (see Section 13.1.2.)
- A new feature logical DECC$POSIX_COMPLIANT_PATHNAMES is added to the C RTL to indicate that an application is operating in a POSIX-compliant mode. In POSIX-compliant mode, only the newest version of a file is visible. Access to multiple versions of a file is not allowed (see Section 13.3.1).
- The DCL command CREATE/SYMLINK is used to create a symbolic link. (see Section 13.2.1).
- The DCL command SET ROOT is used to create the system POSIX root (see Section 13.5).
- Two GNV utilities, `mnt` and `umnt`, are provided to set mount points (see Section 13.7).
- DCL commands and utilities are modified to behave appropriately when acting on and encountering symbolic links (see Section 13.9).
- The TCP/IP Services for OpenVMS Network File System (NFS) client and server are enhanced to support symbolic links on ODS5 volumes.
- Relevant GNV utilities such as `ln` (which can create a symbolic link) and `ls` (which can display the contents of a symbolic link) are updated to provide access to and management of symbolic links (see Sections 13.2.2 and 13.10).

13.1. POSIX Pathnames and Filenames

A POSIX pathname is a non-empty character string consisting of 0 or more filenames, separated by a `/` (the path delimiter). The path delimiter can be the first or last character of a pathname and is also the separator between filenames. Multiple consecutive delimiters are valid and are equivalent to a single delimiter.

POSIX filenames can consist of all ASCII characters except / (the path delimiter) and NUL, which is used as a string terminator in many APIs. (On OpenVMS systems, POSIX filenames are also currently restricted to not contain the ? or * character.) A filename is sometimes referred to as a *pathname component*.

13.1.1. POSIX Pathname Interpretation

If the pathname begins with a /, it is considered to be an absolute pathname, and pathname processing begins at the device and directory specified by the POSIX *root* (see Section 13.5 for the method to define the root).

If the pathname does not begin with a /, it is considered to be a relative pathname, and pathname processing begins with the current working directory.

See Section 13.6 for how OpenVMS interprets the current working directory.

13.1.1.1. The POSIX Root Directory

The POSIX *root* directory is a directory that is used in pathname resolution for absolute pathnames.

The root directory, is the top-most node of the hierarchy, and has itself as its parent directory. The pathname of the root directory is /, and the parent directory of the root directory is /. Note that only files and directories actually entered in the root directory are accessible with a POSIX pathname. For additional details, see the note in Section 13.3.1.

See Section 13.5 for defining the POSIX root on OpenVMS.

13.1.1.2. Symbolic Links

A *symbolic link* is a special kind of file that points to another file. It is a directory entry that associates a filename with a text string that is interpreted as a POSIX pathname when accessed by certain services. Most operations that access a symbolic link are rerouted to the object to which the text contents refer; if that object does not exist, the operation fails. A symbolic link is implemented on OpenVMS as a file of organization SPECIAL and type SYMBOLIC_LINK.

A symbolic link can specify an absolute pathname or a relative pathname. The relative path in a symbolic link is relative to the location of the link. For example, a symbolic link whose content begins with the path "../" refers to its parent directory. If the symbolic link is moved to another directory, it refers to its new parent directory. Note that the path specified by a symbolic link is a POSIX pathname and is subject to the restrictions described in Section 13.3.1.

See Section 13.2 for symbolic link usage on OpenVMS.

13.1.1.3. Mount Points

A *mount point* is a location (in fact, a directory) where a file system is attached. The UNIX term *file system* is essentially equivalent to the OpenVMS concept of a disk volume. After a mount point is established, any contents of the original directory are inaccessible.

Mount points do not persist across reboots.

See Section 13.7 for establishing mount points on OpenVMS.

13.1.1.4. Reserved Filenames . and ..

There are two reserved POSIX filenames:

`.` (the current directory)
`..` (the parent directory)

13.1.1.5. Character Special Files

On UNIX systems, a *special file* is associated with a particular hardware device or other resource of the computer system. The operating system uses character special files, sometimes called device files, to provide I/O access to specific character devices.

Special files, at first glance, appear to be just like ordinary files, in that they:

- Have path names that appear in a directory
- Have the same access protection as ordinary files
- Can be used in almost every way that ordinary files can be used

However, there is an important difference between the two: an ordinary file is a logical grouping of data recorded on disk, whereas a special file corresponds to a device entity. Examples are:

- An actual device, such as a line printer
- A logical subdevice, such as a large section of the disk drive
- A pseudo device, such as the physical memory of the computer (`/dev/mem`), the terminal file (`/dev/tty`), or the null file (`/dev/null`).

Only the null special file (`/dev/null`) is supported. Data written on the null special file is discarded. Reads from a null special file always return 0 bytes.

13.1.2. Using POSIX Pathnames with OpenVMS Interfaces

POSIX compliant pathnames can be passed to most OpenVMS applications, utilities, and APIs.

The existing DCL convention of doubling any quote character that is in a quoted string has been adopted.

Also, in order to avoid constraints on future use of quoted strings for pathnames (for example, to support another syntactic variation, such as Windows-compatible names), the leading tag `^UP^` is required on the quoted pathname to identify it as a POSIX pathname to DCL, RMS, and OpenVMS utilities. It is *not* required for the C RTL or GNV.

So, for example, `/a/b/c` becomes `"^UP^ /a/b/c"` and `/a/b"/c` becomes `"^UP^ /a/b""/c"`.

Throughout the remainder of this chapter, the term *POSIX pathname* refers to the unmodified pathname (for example, `/a/b/c`) and *quoted pathname* refers to our quoted, tagged pathname (for example, `"^UP^ /a/b/c"`).

13.1.2.1. Special Considerations with POSIX Filenames

When a user creates a file on OpenVMS using a POSIX filename, it would be ideal if the filename viewed as an OpenVMS filename were the same. For example, when a user asks to create a file with

POSIX filename `a.b`, OpenVMS could create a file with the name `a.b;` (where `;` represents the file version).

But because one of the components of an OpenVMS filename is the file type (consisting of at least a period), POSIX filenames without a period cannot be mapped in as direct a fashion (for example, POSIX filename `a` cannot be mapped to OpenVMS filename `a;`). The reasonable solution is for OpenVMS to append a period (representing a null file-type field) so that POSIX filename `a` becomes OpenVMS filename `a.;`

Note that OpenVMS appends a period (or null type) even to a POSIX filename that itself ends in a period. This is necessary to distinguish POSIX filenames `a.` and `a` from each other.

Furthermore, OpenVMS directories have a type and version of `.DIR;1`. When a directory of POSIX filename `a` is created on OpenVMS, it is created with an OpenVMS filename of `a.DIR;1`. And because `a.DIR` is a valid POSIX filename, it needs to be distinguished from a directory with POSIX filename `a` residing in the same directory. So OpenVMS adds a period (or null type) to POSIX filenames ending in `.DIR` (so a file with POSIX filename `a.DIR` becomes `a.DIR.;`).

Keeping the above rules in mind, the following are POSIX filenames that are unmodified on OpenVMS, except for adding a version:

```
POSIX filename    OpenVMS name    Type    Version   Displayed by DIR as
--------------    ------------    ----    -------   -------------------
     a.b               a          .b         ;             a.b;
     a.b;              a          .b;        ;             a.b;;
     a.b;2             a          .b;2       ;             a.b;2;
```

The following are examples of POSIX filenames that end with a period (.) or `.DIR` and, therefore, have a period appended when creating the OpenVMS filename:

```
POSIX filename    OpenVMS name    Type    Version   Displayed by DIR as
--------------    ------------    ----    -------   -------------------
     a                 a           .         ;             a.;
     a.                a.          .         ;             a^..;
     a..               a..         .         ;             a^.^..;
     a.b.              a.b.        .         ;             a^.b^..;
     a.DIR             a.DIR       .         ;             a^.DIR.;
```

Finally, OpenVMS will add `.DIR;1` to the end of any POSIX directory:

```
POSIX filename    OpenVMS name    Type    Version   Displayed by DIR as
--------------    ------------    ----    -------   -------------------
     a                 a          .DIR      ;1           a.DIR;1
     a.dir             a.dir      .DIR      ;1           a^.dir.DIR;1
     a.                a.         .DIR      ;1           a^..DIR;1
```

Note that both the OpenVMS filenames `a.DIR;1` and `a.;` map to the POSIX filename `a`. Specifying the POSIX filename `a` will find either `a.DIR;1` or `a.;`. If both are present, it finds `a.;` . The ambiguous mapping applies whether `a.DIR;1` is a directory or an ordinary file.

All POSIX filenames that contain the `*` or `?` characters are unmappable. This is an open issue to be addressed in a future release of OpenVMS.

13.1.2.2. Special Considerations with OpenVMS Filenames

When mapping OpenVMS filenames to POSIX filenames (a feature provided by C RTL function `readdir`), the following rules apply:

If the OpenVMS filename ends in `.DIR;1` and the type of a file is a directory, then `.DIR;1` is removed. For all other files, the ending semi-colon and version number are removed from the Open VMS filename.

If the OpenVMS filename ends in a period, the period is removed.

All OpenVMS filenames that contain the `/` or NUL characters are unmappable and, therefore, not returned to the caller.

OpenVMS directories `..DIR;1` and `...DIR;1` are unmappable because their corresponding POSIX filenames would be the special filenames `.` and `...`

OpenVMS file `.` is unmappable because the resulting POSIX filename would be a NULL string.

13.2. Using Symbolic Links

There are several ways to create a symbolic link:

- Using the DCL command CREATE/SYMLINK (Section 13.2.1)
- Using the `ln -s` utility within the GNV bash shell (Section 13.2.2)
- Using the C RTL `symlink` function (Section 13.3.3)

13.2.1. Creating and Using Symbolic Links with DCL

The following examples show how to create and access symbolic links using DCL commands:

Example 13.1. Examples

1. ```
   $ create/symlink="a/b.txt" link_to_b.txt
   $
   $ dir/date link_to_b.txt

   Directory DKB0:[TEST]

   link_to_b.txt -> a/b.txt
                    27-MAY-2005 09:45:15.20
   ```

   This example creates the symbolic link file `link_to_b.txt` with contents equal to the specified text string representing the pathname to which the symbolic link points, `a/b.txt`. Note that `a/b.txt` may or may not exist. Also note that the symlink file must be reachable from the POSIX root.

2. ```
   $ type link_to_b.txt
     %TYPE-W-OPENIN, error opening DKB0:[TEST]LINK_TO_B.TXT; as input
    -RMS-E-FNF, file not found
   ```

   To access the file referenced in a symbolic link, specify the symbolic link name on the command. The TYPE command error message in this example indicates that the file `a/b.txt`, referenced through the symbolic link `link_to_b.txt` created in the first example, does not exist.

3. ```
   $ create [.a]b.txt
      This is a text file.
   Exit
   ```
   ```

```
$
$ type link_to_b.txt
This is a text file
$
```

This example creates the missing file `b.txt`. Once the referenced file exists, the TYPE command through the symbolic link is successful.

Alternatively, you can create the file through the symbolic link:

```
$ create link_to_b.txt
   This is a text file.
   Exit
$ dir [.a]

Directory DKB0:[TEST.A]

b.txt;1
$
$ type link_to_b.txt
   This is a text file.
$
```

13.2.2. Using Symbolic Links through GNV POSIX and DCL Commands

You can create a symbolic link within the GNV bash shell with the `ln` utility: `bash$ ln -s` *`filename slinkname`*

Most utilities and commands that reference *slinkname* are redirected so that they act on *filename*. Some commands, such as `ls`, are symbolic link-aware, and show information about the link itself (such as its contents).

A link can be removed with the `rm` command (which cannot be made to follow the link and remove what it points to).

The examples that follow show how to manage symbolic links through GNV POSIX commands as compared with DCL commands.

Example 13.2. Examples

1. From GNV:

   ```
   bash$ cd /symlink_example
   bash$ echo This is a test. > text
   bash$ cat text
   This is a test.
   bash$ ln -s text LINKTOTEXT
   bash$
   ```

   From DCL:

   ```
   $ set def DISK$XALR:[PSX$ROOT.symlink_example]
   $ create text.
   This is a test.
   Exit
   ```

```
$ type text.
This is a test.
$ create/symlink="text" LINKTOTEXT
```

This example creates a symbolic link called LINKTOTEXT that points to a text file called `text`.

2. From GNV:

```
bash$ cat LINKTOTEXT
This is a test.
bash$
```

From DCL:

```
$ type LINKTOTEXT.
This is a test.
$
```

This example displays the contents of the text file using the symbolic link.

3. From GNV:

```
bash$ ls
LINKTOTEXT  text
bash$ ls -l
total 1
lrwxr-x---   1 SYSTEM   1        4 May 12 08:41 LINKTOTEXT -> text
-rw-r-----   1 SYSTEM   1        16 May 12 08:40 text
bash$
```

From DCL:

```
$ DIR

Directory DISK$XALR:[PSX$ROOT.symlink_example]

LINKTOTEXT.;1       text.;1

Total of 2 files.

$ DIR/DATE

Directory DISK$XALR:[PSX$ROOT.symlink_example]

LINKTOTEXT.;1 -> /symlink_example/text
                 12-MAY-2005 08:46:31.40
text.;1          12-MAY-2005 08:43:47.53

Total of 2 files.
$
```

This example displays the content of a symbolic link. Notice that when you do an `ls` or a DIR using switches that show any file attribute, then the content of the symbolic link is also displayed.

4. From GNV:

```
bash$ rm text
bash$ cat LINKTOTEXT
```

```
    cat: linktotext: i/o error
    bash$ rm LINKTOTEXT

From DCL:

  $ DEL text.;1
  $ TYPE LINKTOTEXT.
  %TYPE-W-OPENIN, error opening
  DISK$XALR:[PSX$ROOT.symlink_example]LINKTOTEXT.;1 as input
  -RMS-E-ACC, ACP file access failed
  -SYSTEM-F-FILNOTACC, file not accessed on channel

  $ DEL LINKTOTEXT.;1
  $
```

This example deletes files.

13.3. C RTL Support

The following sections describe POSIX-compliant pathname and symbolic link support in the C RTL.

13.3.1. DECC$POSIX_COMPLIANT_PATHNAMES Feature Logical

In order for a POSIX-compliant user to have consistency between the pathnames stored in symbolic links and the pathnames used as input to C RTL functions, the C RTL provides a feature logical, DECC$POSIX_COMPLIANT_PATHNAMES, to allow an application to present POSIX-compliant pathnames to any C RTL function that accepts a pathname.

By default DECC$POSIX_COMPLIANT_PATHNAMES is disabled, and the usual C RTL behavior prevails. Notice that this disabled mode includes interpretation of pathnames as UNIX style specifications and uses rules that are different and unrelated to POSIX-compliant pathname processing.

To disable DECC$POSIX_COMPLIANT_PATHNAMES when necessary, DEFINE it to 0 or "DISABLE":

```
$ DEFINE DECC$POSIX_COMPLIANT_PATHNAMES 0
```

To enable DECC$POSIX_COMPLIANT_PATHNAMES, set it to one of the following values:

1	POSIX only - All pathnames are designated as POSIX style.
2	Leans POSIX - Pathnames that end in " `:`" or contain any of the bracket characters " `[] <>`", and that can be successfully parsed by the SYS$FILESCAN service, are designated as OpenVMS style. Otherwise, they are designated as POSIX style.
3	Leans OpenVMS - The pathnames " `.`" and " `..`", or pathnames that contain " `/`" are designated as POSIX style. Otherwise, they are designated as OpenVMS style.
4	OpenVMS only - All pathnames are designated as OpenVMS style.

Note

Modes 1 and 4 are not recommended because of interactions with other shareable libraries and utilities.

With DECC$POSIX_COMPLIANT_PATHNAMES thus enabled, the C RTL examines pathnames to determine if they should be designated as POSIX style or OpenVMS style, following rules determined by the value assigned to DECC$POSIX_COMPLIANT_PATHNAMES.

For example, to designate that most pathnames are to be POSIX-compliant:

```
$ DEFINE DECC$POSIX_COMPLIANT_PATHNAMES 2
```

When the C RTL designates a pathname as POSIX style, it does not convert a POSIX pathname it receives into an OpenVMS file specification; instead, it converts the POSIX pathname into the quoted pathname format (previously described) and allows RMS to process the pathname as it would process a pathname found in a symbolic link.

Note

Starting with OpenVMS Version 8.4, logical names and device names *are* supported with DECC$POSIX_COMPLIANT_PATHNAMES defined, whereas in previous OpenVMS versions they were not supported.

When DECC$POSIX_COMPLIANT_PATHNAMES is defined, the following C RTL feature logicals are ignored:

DECC$ARGV_PARSE_STYLE
DECC$DISABLE_POSIX_ROOT
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION
DECC$EFS_CASE_PRESERVE
DECC$EFS_CASE_SPECIAL
DECC$EFS_CHARSET
DECC$EFS_NO_DOTS_IN_DIRNAME
DECC$ENABLE_TO_VMS_LOGNAME_CACHE
DECC$FILENAME_UNIX_NO_VERSION
DECC$FILENAME_UNIX_ONLY
DECC$FILENAME_UNIX_REPORT
DECC$NO_ROOTED_SEARCH_LISTS
DECC$READDIR_DROPDOTNOTYPE
DECC$READDIR_KEEPDOTDIR
DECC$RENAME_ALLOW_DIR
DECC$RENAME_NO_INHERIT
DECC$UNIX_PATH_BEFORE_LOGNAME

13.3.2. decc$to_vms, decc$from_vms, and decc$translate_vms

When operating in POSIX-compliant mode, the C RTL function `decc$to_vms` converts a POSIX pathname into the RMS quoted pathname format. If an RMS quoted pathname is passed to either `decc$from_vms` or `decc$translate_vms`, these functions change the quoted pathname into a POSIX pathname by removing the quotes and the `^UP^` prefix (so, for example, `"^UP^a/b"` becomes `a/b`). When not in POSIX-compliant mode, these functions operate as they did previously.

13.3.3. Symbolic Link Functions

Table 13.1 lists and describes the symbolic-link functions in the VSI C Run-Time Library (C RTL). For more detailed information on each function, see the Reference Section.

Table 13.1. Symbolic Link Functions

Function	Description
`lchown`	Changes the owner and/or group of a file. If the file is a symbolic link, the owner of the symbolic link is modified (in contrast to `chown` which would modify the file that the symbolic link points to).
`lstat`	Retrieves information about the specified file. If the file is a symbolic link, information about the link itself is returned (in contrast to `stat`, which would return information about the file that the symbolic link points to).
`readlink`	Reads the contents of a symbolic link and places them into a user-supplied buffer.
`realpath`	Returns an absolute pathname from the POSIX root. This function is only supported in POSIX-compliant modes; that is, with DECC$POSIX_COMPLIANT_PATHNAMES enabled.
`unlink`	Deletes a symbolic link from the system.
`symlink`	Creates a symbolic link containing the specified contents.

13.3.4. Modifications to Existing Functions

In addition to the previously described new functions, the following C RTL functions are modified to support symbolic links:

- `creat`

  When a new version of a file is created, and the named file already exists as a symbolic link, the file to which the symbolic link refers is created.

- `open`

  If the *file_spec* parameter refers to a symbolic link, the `open` function opens the file pointed to by the symbolic link.

- `delete`, `remove`

  When `delete` or `remove` is used to delete a symbolic link, the link itself is deleted, not the file to which it refers.

Also, in accordance with the Open Group specification, an `errno` value of ELOOP is returned if a loop exists in the resolution of a symbolic link.

13.3.5. Non POSIX-Compliant Behavior

13.3.5.1. Multiple Versions of Files

When using POSIX-compliant pathnames, if multiple versions of a file exist, you can access only the latest version of the file. Deleting a file results in the deletion of only the highest-numbered version of a file.

13.3.5.2. Ambiguous Filenames

If the file `"a"` and directory `"a.DIR;1"` both exist in a directory, users in POSIX-compliant mode will have the following access:

- Functions that take only a directory name as input (such as `chdir` and `opendir`) will return an error (ENOEXIST).
- Functions whose input is a non-directory file (such as `open`) will act on the non-directory file.
- Functions whose input can be either a file or a directory (such as `stat`) will act on the non-directory file.
- `readdir` will return entries for both the file and the directory.

13.3.5.3. POSIX Security Behavior for File Deletion

The OpenVMS C RTL library follows traditional OpenVMS security rules for permitting the deletion or unlinking of files and uses the protection on the files or links, whereas the POSIX security behavior for deleting files is to follow the security settings for the parent directory of the files or links.

To get the POSIX security behavior, files in a directory must have an Access Control Entry (ACE) placed on them that always grants delete access to those files by the class of users or programs that you want to have this behavior. You may also want to create a default protection ACE on the parent directory so newly created files get the desired behavior.

Please consult the *VSI OpenVMS Guide to System Security* for more information on how to manage Access Control Entries.

13.4. RMS Interface

13.4.1. RMS Input/Output of POSIX Pathnames

As previously described, a quoted pathname is a POSIX pathname prefixed with the tag string `^UP^` and enclosed with opening and closing quotation marks. In addition, any quotation marks present in the POSIX pathname need to be doubled, consistent with the way DCL handles quoted strings.

Quoted pathnames are allowed in the primary name, default name, and/or related names. By identifying the `^UP^` prefix and the opening and closing quotation marks, RMS interprets the rest of the characters as a standard POSIX pathname (with the exception that every pair of quotation marks in the pathname is treated as a single quotation mark). If RMS detects a quoted pathname, then the expanded and resultant file specifications are supplied by RMS as quoted pathnames as well.

The components of the returned quoted pathname are referenced by the NAM or NAML as follows:

- The *node* is NULL
- The *dev* is "^UP^
- The *dir* consists of all characters between the dev string up to and including the final slash (/)
- The *name* consists of all characters following the dir up to the final period (.) (If there is no final period, the name consists of all characters following the *dir* up to the closing double quote (")
- The *type* consists of all characters following the *name* up to the closing double quote (")
- The version is double quote (")

The *dir*, *name*, and *type* fields can be NUL.

For example, the quoted pathname `"^UP^/a/b.c"` consists of the following components:

The *node* is NUL
The *dev* is "^UP^
The *dir* is /a/
The *name* is b
The *type* is .c
The *version* is "

The following RMS routines support symbolic links:

- SYS$OPEN operates on the target file pointed to by the symbolic link.
- SYS$CREATE creates the file pointed to by the symbolic link.
- SYS$PARSE analyzes the file specification string and fills in various NAM block fields.
- SYS$SEARCH returns the DVI and FID of the target file. The DID is zero. The resultant name is that of the symbolic link and not the target file.
- Setting NAML$V_OPEN_SPECIAL causes SYS$OPEN and SYS$SEARCH to not follow the symbolic link.

The following restrictions apply to RMS processing of POSIX pathnames:

- If the primary file specification is a quoted pathname, then SYS$SEARCH returns only one file; in pathnames, no characters are defined as being wildcards.
- If the primary file specification is a quoted pathname, then the default and related specifications are ignored.
- If the primary file specification is a traditional OpenVMS file specification, then the default specification can be a quoted pathname; however, only the filename and type are used from the default specification.

13.4.2. Application Control of RMS Symbolic Link Processing

Applications, including DCL commands and utilities, need the ability to control RMS behavior when RMS encounters symbolic links during typical file operations. Specifically, applications need to be able to specify whether or not a pathname stored in a symbolic link should be included as part of pathname processing and directory searches. Furthermore, applications need to be able to specify different behavior depending on how the symbolic link is encountered.

A NAML$L_INPUT_FLAGS flag is now provided to allow necessary application flexibility:

- If SYS$OPEN is supplied a pathname that is a symbolic link, a non-symbolic link is the default behavior is to open the file specified by the pathname stored in the symbolic link. If the flag NAML$V_OPEN_SPECIAL is set, then the symbolic link file itself is opened.
- If the result of a call to SYS$SEARCH is a symbolic link and the flag NAML$V_OPEN_SPECIAL is set, the DVI/FID of the symbolic link itself is returned. Otherwise, the default behavior is that the DVI/FID of the file specified by the pathname stored in the symbolic link is returned, unless that pathname is also a symbolic link, in which case the process is repeated until anon-symbolic link is reached before returning the DVI/FID.

- When SYS$SEARCH performs wildcard directory lookups, then any symbolic links encountered are not examined to determine if they refer to directories.

Also, on ODS-5 volumes, a SPECIAL_FILES flag is used to communicate to RMS operations whether to follow symbolic links or not. The RMS operations that follow symbolic links are SYS$OPEN, SYS$CREATE, SYS$SEARCH and all directory path interpretations.

For more information on how to set the SPECIAL_FILES flag, see *VSI OpenVMS System Services Reference Manual* and *VSI OpenVMS DCL Dictionary*.

13.5. Defining the POSIX Root

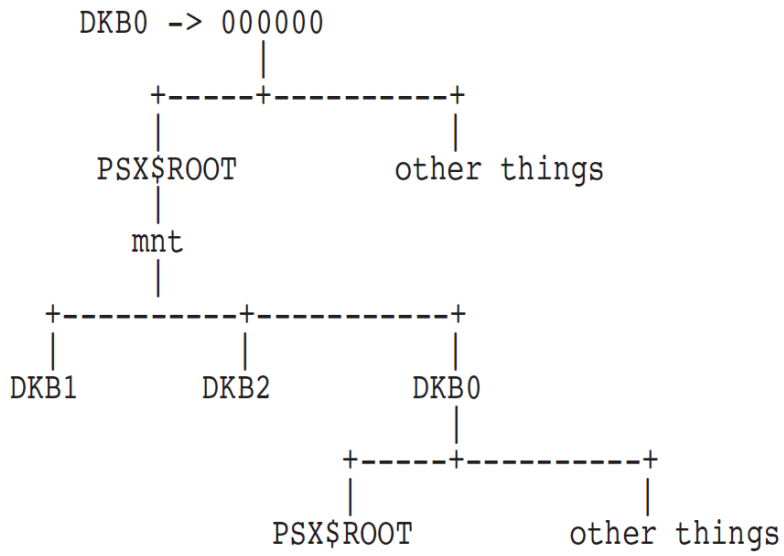
For absolute POSIX pathnames, RMS needs to interpret the leading '/' as a UNIX-style root. POSIX expects that everything starts from this POSIX root. The POSIX root can be established with the GNV installation procedure or with the new DCL command, SET ROOT.

13.5.1. Suggested Placement of the POSIX Root

During the GNV installation and configuration procedure or when using the SET ROOT command, you need to provide a location for the POSIX root for the system. By default, the GNV installation establishes the location of the root at SYS$SYSDEVICE:[PSX$ROOT]. You can accept this directory as the root or specify another directory. GNV then optionally creates a mount point in "/mnt" (which is directory "mnt" in your POSIX root directory) for each disk device on your system.

If you accept the default for the root, or specify any other directory that is not the top-level directory of a device, then files that do not reside in the directory tree under the root directory will be inaccessible using a POSIX pathname, unless the device containing the root directory is mounted by means of a mount point under the root. (This mount point is optionally generated by GNV during the configuration.) See Figure 13.1.

Figure 13.1. POSIX Root Placement



An alternative is to specify the top-level directory, the Master File Directory (MFD), of a device as the root and not create a mount point for that device. This makes all of the files on the device accessible via POSIX pathnames. However, using the MFD of a device (especially the system device) as the root means that all of the files in the MFD are visible when a user displays the files in a root directory (such

as "ls /"). GNV users will see OpenVMS files that they might not expect to see; similarly, DCL users will see UNIX files they might not expect to see. Also, the MFD of a device might not be accessible to most general users, resulting in unexpected failures.

We recommend you use the supplied default root directory.

For more information on mount points, see Section 13.7.

13.5.2. The SET ROOT Command

The SET ROOT command has the following format:

SET ROOT ***device-name:directory-spec***

This command defines the POSIX root. In POSIX pathname processing mode, RMS and the C RTL will treat the leading '/' of a pathname as referring to the defined root. By default, the root is SYS$SYSDEVICE:[PSX$ROOT].

The root definition does not persist across a reboot.

The SET ROOT command has the following qualifier:

/LOG (default)
/NOLOG

Controls whether the SET ROOT command displays a success indication after the root definition is set.

13.5.3. The SHOW ROOT Command

The SHOW ROOT command displays the current value of the system root and, if defined, the process root.

This command has the following format:

SHOW ROOT

13.6. Current Working Directory

On UNIX systems, the current working directory is a directory that is used in pathname resolution for pathnames that do not begin with a slash.

If RMS encounters a symbolic link, the directory in which the symbolic link is found is the current working directory for the purposes of pathname resolution.

For the resolution of POSIX pathnames passed directly to RMS, the current working directory is the current OpenVMS default directory for the process. If the OpenVMS default directory is not a single directory (for example, the directory specification includes a search list), then the current working directory is the first directory encountered through the search list.

13.7. Establishing Mount Points

Two new utilities, `mnt` and `umnt` provide away for a user to mount and dismount file systems from mount points. Only users who have CMKRNL privilege can mount and dismount file systems. These utilities are shipped as part of GNV.

The utilities maintain their own logical name table of mount points, which is used to display currently available mount points upon request.

The interfaces for these utilities are as follows:

`mnt`

List existing mount points.

`mnt` ***file_system mount_point***

Mount file system *file_system* on mount point *mount_point*.

`umnt` ***mount_point***

Unmount the file system mounted on mount point *mount_point*.

`umnt` ***file_system***

Unmount the file system *file_system* from its mount point.

`umnt -A`

Unmount all file systems from their mount points.

Note that both OpenVMS file specifications and POSIX pathnames are allowed for the arguments.

13.8. NFS

In TCP/IP Services for OpenVMS, Version 5.4 and older, there was limited symbolic link support in the NFS client for the purpose of backing up symbolic links from a UNIX server into an OpenVMS saveset and correctly restoring them. The client used an application ACE to flag a file as a symbolic link. In Version 5.5, the client preserves this behavior for an ODS-2 mount. For an ODS-5 mount, it creates a symbolic link when it sees the application ACE. This allows savesets created with older versions of the client to still be restored with newer versions.

13.9. DCL

OpenVMS DCL commands and utilities behave in predictable ways when encountering symbolic links. Usually, the symbolic link is followed; for example, if symbolic link LINKFILE.TXT points to file TEXT.TXT, then entering the command "type LINKFILE.TXT" displays the text contained in TEXT.TXT.

The symbolic link is followed even if the symbolic link itself is not directly specified; for example, if a symbolic link that references a directory is encountered when processing a command such as "`dir [...]*.TXT`", any files with a type of "`.TXT`" that exist in the pointed-to directory (and its subdirectories) would be displayed. (Note: For OpenVMS Version 8.3, during directory wildcard operations, symbolic links are not examined to see if they refer to directories.)

There are some cases, however, where the symbolic link should not be followed by default. Also, it is sometimes desirable to give the user the option of whether or not to follow a symbolic link.

The following commands either do not follow the symbolic link by default and/or provide an option whether or not to follow a symbolic link.

BACKUP

When a symbolic link is encountered during a backup operation, the symbolic link itself is copied. This is true for all backup types – physical, image and file.

COPY

/SYMLINK

/NOSYMLINK (default)

If an input file is a symbolic link, the file referred to by the symbolic link is the file that is copied.

The /SYMLINK qualifier indicates that any input symbolic link is copied.

CREATE

/SYMLINK=" *text*"

Creates a symbolic link containing the specified text without the enclosing quotation marks.

If the created symbolic link is subsequently encountered during any filename processing, the contents of the symbolic link are read and treated as a POSIX pathname specification.

No previous version of the symbolic link can exist.

DELETE

If an input file specification parameter is a symbolic link, the symbolic link itself is deleted.

DIRECTORY

/SYMLINK (default)

/NOSYMLINK

If an input file specification parameter is a symbolic link, the displayed file attributes are those of the symbolic link itself. If any file attribute is requested, then the contents of the symbolic link are also displayed with an arrow appearing between the filename and the contents (for example, LINK.TXT -> FILE.TXT).

The /NOSYMLINK qualifier indicates that if an input file specification is a symbolic link, then the file attributes of the file to which the symbolic link refers are displayed; the displayed name is still the name of the symbolic link itself.

DUMP

/SYMLINK

/NOSYMLINK (default)

If an input file is a symbolic link, the file referred to by the symbolic link is the file that is dumped.

The /SYMLINK qualifier indicates that any input symbolic link is dumped.

PURGE

If an input file specification is a symbolic link, the symbolic link itself is purged. Because only one version of a symbolic link can exist, this command has no effect on that file.

RENAME

If an input file specification is a symbolic link, the symbolic link itself is renamed.

If the output file specification is a symbolic link, the operation fails.

SET FILE

/SYMLINK

/NOSYMLINK (default)

If an input file is a symbolic link, the file referred to by the symbolic link is the file that is set.

The /SYMLINK qualifier indicates that the symbolic link itself is set.

Notes

- The /EXCLUDE qualifier excludes symbolic links that match the qualifier's file specification of files to be excluded. The filename of a file that is referred to by a symbolic link is not excluded on the basis of its own name.

  Example:

  If symbolic link LINK.TXT refers to TEXT.TXT in another directory, then:

  /EXCLUDE=LINK.TXT would exclude LINK.TXT (and, therefore, TEXT.TXT) from the operation.
  /EXCLUDE=TEXT.TXT would not exclude LINK.TXT or TEXT.TXT from the operation.

- Given a symbolic link, F$FILE_ATTRIBUTES acts on the file to which the symbolic link refers. A new lexical function, F$SYMLINK_ATTRIBUTES, returns information on the symbolic link itself. New lexical function F$READLINK returns the text contents of a symbolic link, and NULL if the input is not a symbolic link.

- Some compilers and utilities will accept quoted pathnames as input file specifications and as arguments to qualifiers. These include the C and C++ compilers, CXXLINK, the Linker, and the Librarian. Linker option files can also contain quoted pathnames. Note that the /INCLUDE_DIRECTORY qualifier for the compilers continues to accept standard POSIX pathnames rather than the quoted pathnames. The Java compiler also continues to accept standard POSIX pathnames. In addition, SET DEFAULT accepts quoted pathnames as input.

13.10. GNV

GNV is the mechanism by which the Open Group utilities are provided on OpenVMS systems. GNV is an open source project with updates released on the Sourceforge web site. GNV is also available on the OpenVMS Open Source CD.

Because GNV relies on the C RTL for all file access, no modifications are made to GNV for its provided utilities to behave as defined by the Open Group specification for symbolic links. To make use of the new symbolic-link and POSIX pathname support features, users should set the C RTLDECC$POSIX_COMPLIANT_PATHNAMES feature logical appropriately, before invoking BASH.

13.11. Restrictions

The following are known restrictions with symbolic link support:

- SET FILE/REMOVE on a symbolic link silently fails.

  When SET FILE/REMOVE is executed on a symbolic link, the link is not removed. No error is displayed.

- Coding consideration with RMS and symbolic links:

  With the introduction of symbolic links and POSIX pathname processing, the device name that occurs in a file specification may be different from the device on which the file exists. The NAM$T_DVI returned by SYS$PARSE/SYS$SEARCH properly identifies the device on which the file resides.

- Symlinks are limited to a single version.

  If you use the BACKUP/NEW_VERSION, COPY, or RENAME commands on a symbolik link, you may get the following error message:

  ```
  NOSYMLINKERS, cannot create multiple versions of a symlink
  ```

  This message indicates that the file system has rejected an attempt to create multiple versions of a file, where at least one version is a symbolic link.

  To avoid this error, specify a different file name for the output file, or use the /REPLACE qualifier to delete the existing file and replace it with the new one.

Reference Section

This section describes the functions contained in the VSI C Run-Time Library (C RTL), listed alphabetically.

a64l

a64l — Converts a character string to a `long` integer.

Format

```
#include <stdlib.h>
long a64l (const char *s);
```

Argument

s

A pointer to the character string that is to be converted to a `long` integer.

Description

The `a64l` and `l64a` functions are used to maintain numbers stored in base-64 ASCII characters as follows:

- `a64l` converts a character string to a `long` integer.
- `l64a` converts a `long` integer to a character string.

Each character used for storing a `long` integer represents a numeric value from 0 through 63. Up to six characters can be used to represent a `long` integer.

The characters are translated as follows:

- A period (.) represents 0.
- A slash (/) represents 1.
- The numbers 0 through 9 represent 2 through 11.
- Uppercase letters A through Z represent 12 through 37.
- Lowercase letters a through z represent 38 through 63.

The `a64l` function takes a pointer to a base-64 representation, in which the first digit is the least significant, and returns a corresponding `long` value. If the string pointed to by the *s* parameter exceeds six characters, `a64l` uses only the first six characters.

If the first six characters of the string contain a null terminator, `a64l` uses only characters preceding the null terminator.

The `a64l` function translates a character string from left to right with the least significant number on the left, decoding each character as a 6-bit base-64 number.

If *s* is the NULL pointer or if the string pointed to by *s* was not generated by a previous call to `l64a`, the behavior of `a64l` is unspecified.

See also `l64a`.

Return Values

n

Upon successful completion, the `long` value resulting from conversion of the input string.

0L

Indicates that the string pointed to by *s* is an empty string.

abort

abort — Sends the signal SIGABRT that terminates execution of the program.

Format

```
#include <stdlib.h>
void abort (void);
```

abs

abs — Returns the absolute value of an integer.

Format

```
#include <stdlib.h>
int abs (int x);
```

Argument

x

An integer.

Return Value

x

The absolute value of the input argument. If the argument is LONG_MIN, `abs` returns LONG_MIN because –LONG_MIN cannot fit in an `int` variable.

access

access — Checks a file to see whether a specified access mode is allowed. The `access` function does not accept network files as arguments.

Format

```
#include <unistd.h>
int access (const char *file_spec, int mode);
```

Arguments

file_spec

A character string that gives an OpenVMS or UNIX style file specification. The usual defaults and logical name translations are applied to the file specification.

mode

Interpreted as shown in Table 33.

Table 33. Interpretation of the mode Argument

Mode Argument	Access Mode
F_OK	Tests to see if the file exists
X_OK	Execute
W_OK	Write (implies delete access)
R_OK	Read

Combinations of access modes are indicated by ORing the values. For example, to check to see if a file has RWED access mode, invoke `access` as follows:

```
access (file_spec, R_OK | W_OK | X_OK);
```

Description

The `access` function checks a file to see whether a specified access mode is allowed. If the DECC$ACL_ACCESS_CHECK feature logical is enabled, this function checks OpenVMS Access Control Lists (ACLs) as well as the UIC protection.

Return Values

0

Indicates that the access is allowed.

-1

Indicates that the access is not allowed.

Example

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

main()
{
```

```
    if (access("sys$login:login.com", F_OK)) {
        perror("ACCESS - FAILED");
        exit(2);
    }
}
```

acos

acos — Returns the arc cosine of its argument.

Format

```
#include <math.h>
double acos (double x);
float acosf (float x);
long double acosl (long double x);
double acosd (double x);
float acosdf (float x);
long double acosdl (long double x);
```

Argument

x

A radian expressed as a real value in the domain [-1,1].

Description

The `acos` functions compute the principal value of the arc cosine of *x* in the range [0, π] radians for *x* in the domain [-1,1].

The `acosd` functions compute the principal value of the arc cosine of *x* in the range [0,180] degrees for *x* in the domain [-1,1].

For `abs`(*x*) > 1, the value of `acos`(*x*) is 0, and `errno` is set to EDOM.

acosh

acosh — Returns the hyperbolic arc cosine of its argument.

Format

```
#include <math.h>
double acosh (double x);
float acoshf (float x);
long double acoshl (long double x);
```

Argument

x

A radian expressed as a real value in the domain [1, +Infinity].

Description

The `acosh` functions return the hyperbolic arc cosine of *x* for *x* in the domain [1, +Infinity], where `acosh`(*x*) = ln(*x* + sqrt(*x***2 - 1)).

The `acosh` function is the inverse function of `cosh` where `acosh`(`cosh`(*x*)) = | *x*|.

x < 1 is an invalid argument.

[w]addch

[w]addch — Add a character to the window at the current position of the cursor.

Format

```
#include <curses.h>
int addch (char ch);
int waddch (WINDOW *win, char ch);
```

Arguments

win

A pointer to the window.

ch

The character to be added. A new-line character (\n) clears the line to the end, and moves the cursor to the next line at the same x coordinate. A return character (\r) moves the cursor to the beginning of the line on the window. A tab character (\t) moves the cursor to the next tabstop within the window.

Description

When the `waddch` function is used on a subwindow, it writes the character onto the underlying window as well.

The `addch` routine performs the same function as `waddch`, but on the `stdscr` window.

The cursor is moved after the character is written to the screen.

Return Values

OK

Indicates success.

ERR

Indicates that writing the character would cause the screen to scroll illegally. For more information, see the `scrollok` function.

[w]addstr

[w]addstr — Add the string pointed to by *str* to the window at the current position of the cursor.

Format

```
#include <curses.h>
int addstr (char *str);
int waddstr (WINDOW *win, char *str);
```

Arguments

win

A pointer to the window.

str

A pointer to a character string.

Description

When the `waddstr` function is used on a subwindow, the string is written onto the underlying window as well.

The `addstr` routine performs the same function as `waddstr`, but on the `stdscr` window.

The cursor position changes as a result of calling this routine.

Return Values

OK

Indicates success.

ERR

Indicates that the function causes the screen to scroll illegally, but it places as much of the string onto the window as possible. For more information, see the `scrollok` function.

alarm

alarm — Sends the signal SIGALRM (defined in the `<signal.h>` header file) to the invoking process after the number of seconds indicated by its argument has elapsed.

Format

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds); (ISO POSIX-1)
int alarm (unsigned int seconds); (Compatibility)
```

Argument

seconds

Has a maximum limit of LONG_MAX seconds.

Description

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

Unless it is intercepted or ignored, the signal generated by `alarm` terminates the process. Successive `alarm` calls reinitialize the alarm clock. Alarms are not stacked.

Because the clock has a 1-second resolution, the signal may occur up to 1 second early. If the SIGALRM signal is intercepted, resumption of execution may be held up due to scheduling delays.

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

Return Value

n

The number of seconds remaining from a previous alarm request.

alloca

alloca — Allocates memory on the stack.

Format

```
#include <alloca.h>
void *alloca (unsigned int size);
```

Arguments

size

The total number of bytes to be allocated.

Description

The `alloca` function allocates *size* bytes from the stack frame of the caller. The memory is automatically freed when the function that calls `alloca` returns to its caller. See the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/] for the description of the __ALLOCA macro.

Return Value

x

Returns a pointer to the allocated memory.

asctime, asctime_r

asctime, asctime_r — Converts a broken-down time in a `tm` structure into a 26-character string in the following form: `Sun Sep 16 01:03:52 1984 \n \0`. All fields have a constant width.

Format

```
#include <time.h>
char *asctime (const struct tm *timeptr);
char *asctime_r (const struct tm *timeptr, char *buffer); (ISO POSIX-1)
```

Arguments

timeptr

A pointer to a structure of type `tm`, which contains the broken-down time.

The `tm` structure is defined in the `<time.h>` header file, and also shown in Table 36 in the description of `localtime`.

buffer

A pointer to a character array that is at least 26 bytes long. This array is used to store the generated date-and-time string.

Description

The `asctime` and `asctime_r` functions convert the contents of `tm` into a 26-character string and returns a pointer to the string.

The difference between `asctime_r` and `asctime` is that the former puts the result into a user-specified buffer. The latter puts the result into thread-specific static memory allocated by the C RTL, which can be overwritten by subsequent calls to `ctime` or `asctime`; you must make a copy if you want to save it.

On success, `asctime` returns a pointer to the string; `asctime_r` returns its second argument. On failure, these functions return the NULL pointer.

See the `localtime` function for a list of the members in `tm`.

Note

Generally speaking, UTC-based time functions can affect in-memory timezone information, which is processwide data. However, if the system time zone remains the same during the execution of the application (which is the common case) and the cache of timezone files is enabled (which is the default), then the `_r` variant of the time functions `asctime_r`, `ctime_r`, `gmtime_r` and `localtime_r`, is both thread-safe and AST-reentrant.

If, however, the system time zone can change during the execution of the application or the cache of timezone files is not enabled, then both variants of the UTC-based time functions belong to the third class of functions, which are neither thread-safe nor AST-reentrant.

Return Values

x

A pointer to the string, if successful.

NULL

Indicates failure.

asin

asin — Returns the arc sine of its argument.

Format

```
#include <math.h>
double asin (double x);
float asinf (float x);
long double asinl (long double x);
double asind (double x);
float asindf (float x);
long double asindl (long double x);
```

Argument

x

A radian expressed as a real number in the domain [-1,1].

Description

The `asin` functions compute the principal value of the arc sine of *x* in the range [- π/2, π/2] radians for *x* in the domain [-1,1].

The `asind` functions compute the principal value of the arc sine of *x* in the range [-90,90] degrees for *x* in the domain [-1,1].

When `abs`(*x*) is greater than 1.0, the value of `asin`(*x*) is 0, and `errno` is set to EDOM.

asinh

asinh — Returns the hyperbolic arc sine of its argument.

Format

```
#include <math.h>
double asinh (double x);
float asinhf (float x);
long double asinhl (long double x);
```

Argument

x

A radian expressed as a real value in the domain [-Infinity,+Infinity].

Description

The `asinh` functions return the hyperbolic arc sine of *x* for *x* in the domain [-Infinity, +Infinity], where `asinh`(*x*) = ln(*x* + sqrt(*x***2+ 1)).

The `asinh` function is the inverse function of `sinh` where `asinh`(`sinh`(*x*)) = x.

assert

assert — Used for implementing run-time diagnostics in programs.

Format

```
#include <assert.h>
void assert (int expression);
```

Argument

expression

An expression that has an `int` type.

Description

When `assert` is executed, if `expression` is false (that is, it evaluates to 0), `assert` writes information about the particular call that failed (including the text of the argument, the name of the source file, and the source line number; the latter two are, respectively, the values of the preprocessing macros `__FILE__` and `__LINE__`) to the standard error file in an implementation-defined format. Then, it calls the `abort` function.

The `assert` function writes a message in the following form:

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

If *expression* is true (that is, it evaluates to nonzero) or if the signal SIGABRT is being ignored, `assert` returns no value.

Note

If a null character ('\0') is part of the expression being asserted, then only the text up to and including the null character is printed, since the null character effectively terminates the string being output.

Compiling with the CC command qualifier /DEFINE=NDEBUG or with the preprocessor directive `#define` NDEBUG ahead of the `#include assert` statement causes the `assert` function to have no effect.

Example

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

main()
{
```

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

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

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

atan

atan — Returns the arc tangent of its radian argument, in the range [-π/2, π/2] radians.

Format

```
#include <math.h>
double atan (double x);
float atanf (float x);
long double atanl (long double x);
double atand (double x);
float atandf (float x);
long double atandl (long double x);
```

Argument

x

A radian expressed as a real number.

Description

The `atan` functions compute the principal value of the arc tangent of *x* in the range [- π/2, π/2] radians.

The `atand` functions compute the principal value of the arc tangent of *x* in the range [-90,90] degrees.

atan2

atan2 — Returns the arc tangent of *y/x* (its two radian arguments), in the range [-π, π] radians.

Format

```
#include <math.h>
double atan2 (double y, double x);
float atan2f (float y, float x);
long double atan2l (long double y, long double x);
double atand2 (double y, double x);
float atand2f (float y, float x);
long double atand2l (long double y, long double x);
```

Arguments

y

A radian expressed as a real number.

x

A radian expressed as a real number.

Description

The `atan2` functions compute the principal value of the arc tangent of *y*/ *x* in the range [- π, π] radians. The sign of `atan2` and `atan2f` is determined by the sign of *y*. The value of `atan2`(*y*, *x*) is computed as follows, where *f* is the number of fraction bits associated with the data type:

Value of Input Arguments	Angle Returned
x = 0 or *y*/*x* > 2**(*f*+1)	π/2 * (sign *y*)
x > 0 and *y*/*x* <= 2**(*f*+1)	atan(*y*/*x*)
x < 0 and *y*/*x* <= 2**(*f*+1)	π * (sign *y*) + atan(y/x)

The `atand2` functions compute the principal value of the arc tangent of *y*/ *x* in the range [-180,180] degrees. The sign of `atand2` and `atand2f` is determined by the sign of *y*.

The following are invalid arguments for the `atan2` and `atand2` functions:

Function	Exceptional Argument
`atan2, atan2f, atan2l`	*x* = *y* = 0
`atan2, atan2f, atan2l`	| *x* | = | *y* | = Infinity
`atand2, atand2f, atand2l`	*x* = *y* = 0
`atand2, atand2f, atand2l`	| *x* | = | *y* | = Infinity

atanh

atanh — Returns the hyperbolic arc tangent of its argument.

Format

```
#include <math.h>
double atanh (double x);
float atanhf (float x);
long double atanhl (long double x);
```

Argument

x

A radian expressed as a real value in the domain [-1,1].

Description

The `atanh` functions return the hyperbolic arc tangent of *x*. The `atanh` function is the inverse function of `tanh` where `atanh`(`tanh`(*x*)) = *x*.

|*x*| > 1 is an invalid argument.

atexit

atexit — Registers a function that is called without arguments at program termination.

Format

```
#include <stdlib.h>
int atexit (void (*func) (void));
```

Argument

func

A pointer to the function to be registered.

Return Values

0

Indicates that the registration has succeeded.

nonzero

Indicates failure.

Restriction

The `longjmp` function cannot be executed from within the handler, because the destination address of the `longjmp` no longer exists.

Example

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

static void hw(void);

main()
{
    atexit(hw);
}

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

Running this example produces the following output:

```
Hello, world
```

atof

atof — Converts an ASCII character string to a double-precision number.

Format

```
#include <stdlib.h>
double atof (const char *nptr);
```

Argument

nptr

A pointer to the character string to be converted to a double-precision number. The string is interpreted by the same rules that are used to interpret floating constants.

Description

The string to be converted has the following format:

```
[white-spaces][+|-]digits[radix-character][digits][e|E[+|-]integer]
```

Where *radix-character* is defined in the current locale.

The first unrecognized character ends the conversion.

This function is equivalent to `strtod(nptr, (char**) NULL)`.

Return Values

x

The converted value.

0

Indicates an underflow or the conversion could not be performed. The function sets `errno` to ERANGE or EINVAL, respectively.

±HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

atoi, atol

atoi, atol — 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);
```

Argument

nptr

A pointer to the character string to be converted to a numeric value.

Description

The `atoi` and `atol` functions convert the initial portion of a string to its decimal `int` or `long int` value, respectively. The `atoi` and `atol` functions do not account for overflows resulting from the conversion. The string to be converted has the following format: `[white-spaces][+|-]digits`

The function call `atol (str)` is equivalent to `strtol (str, (char**)NULL, 10)`, and the function call `atoi (str)` is equivalent to `(int) strtol(str, (char**)NULL, 10)`, except, in both cases, for the behavior on error.

Return Value

n

The converted value.

atoq, atoll

atoq, atoll — Convert strings of ASCII characters to the appropriate numeric values. `atoll` is a synonym for `atoq`.

Format

```
#include <stdlib.h>
__int64 atoq (const char *nptr);
__int64 atoll (const char *nptr);
```

Argument

nptr

A pointer to the character string to be converted to a numeric value.

Description

The `atoq` (or `atoll`) function converts the initial portion of a string to its decimal `__int64` value. This function does not account for overflows resulting from the conversion. The string to be converted has the following format:

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

The function call `atoq (str)` is equivalent to `strtoq (str, (char**)NULL, 10)`, except for the behavior on error.

Return Value

n

The converted value.

basename

basename — Returns the last component of a pathname.

Format

```
#include <libgen.h>
char *basename (char *path);
```

Function Variants

The `basename` function has variants named `_basename32` and `_basename64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

path

A UNIX style pathname from which the base pathname is extracted.

Description

The `basename` function takes the UNIX style pathname pointed to by *path* and returns a pointer to the pathname's final component, deleting any trailing slash (/) characters.

If *path* consists entirely of the slash (/) character, the function returns a pointer to the string "/".

If *path* is a NULL pointer or points to an empty string, the function returns a pointer to the string ".".

The `basename` function can modify the string pointed to by *path*.

Return Values

x

A pointer to the final component of *path*.

"/"

If *path* consists entirely of the '/' character.

"."

If *path* is a NULL pointer or points to an empty string.

bcmp

bcmp — Compares byte strings.

Format

```
#include <strings.h>
void bcmp (const void *string1, const void *string2, size_t length);
```

Arguments

string1, string2

The byte strings to be compared.

length

The length (in bytes) of the strings.

Description

The `bcmp` function compares the byte string in *string1* against the byte string in *string2*.

Unlike the string functions, there is no checking for null bytes. Zero-length strings are always identical.

Note that `bcmp` is equivalent to `memcmp`, which is defined by the ANSI C Standard. Therefore, using `memcmp` is recommended for portable programs.

Return Values

0

The strings are identical.

Nonzero

The strings are not identical.

bcopy

bcopy — Copies byte strings.

Format

```
#include <strings.h>
void bcopy (const void *source, void *destination, size_t length);
```

Arguments

source

Pointer to the source string.

destination

Pointer to the destination string.

length

The length (in bytes) of the string.

Description

The `bcopy` function operates on variable-length strings of bytes. It copies the value of the *length* argument, in bytes, from the string in the *source* argument to the string in the *destination* argument.

Unlike the string functions, there is no checking for null bytes. If the *length* argument is 0 (zero), no bytes are copied.

Note that `bcopy` is equivalent to `memcpy`, which is defined by the ANSI C Standard. Therefore, using `memcpy` is recommended for portable programs.

box

box — Draws a box around the window using the character `vert` as the character for drawing the vertical lines of the rectangle, and `hor` for drawing the horizontal lines of the rectangle.

Format

```
#include <curses.h>
int box (WINDOW *win, char vert, char hor);
```

Arguments

win

The address of the window.

vert

The character for the vertical edges of the window.

hor

The character for the horizontal edges of the window.

Description

The `box` function copies boxes drawn on subwindows onto the underlying window. Use caution when using functions such as `overlay` and `overwrite` with boxed subwindows. Such functions copy the box onto the underlying window.

Return Values

OK

Indicates success.

ERR

Indicates an error.

brk

brk — Determines the lowest virtual address that is not used with the program.

Format

```
#include <stdlib.h>
void *brk (unsigned long int addr);
```

Argument

addr

The lowest address, which the function rounds up to the next multiple of the page size. This rounded address is called the *break address*.

Description

An address that is greater than or equal to the break address and less than the stack pointer is considered to be outside the program's address space. Attempts to reference it will cause access violations.

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

Return Values

n

The new break address.

(void *)(-1)

Indicates that the program is requesting too much memory. `ecrrno` and `vaxc$errno` are updated.

Restriction

Unlike other C library implementations, the C RTL memory allocation functions (such as `malloc`) do not rely on `brk` or `sbrk` to manage the program heap space. Consequently, on OpenVMS systems, calling `brk` or `sbrk` can interfere with memory allocation routines. The `brk` and `sbrk` functions are provided only for compatibility purposes.

bsearch

bsearch — 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 *));
```

Function Variants

The `bsearch` function has variants named `_bsearch32` and `_bsearch64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

key

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

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

The number of objects in the array.

size

The size of an object, in bytes.

compar

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 must return an integer less than, equal to, or greater than 0.

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

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

Return Values

x

A pointer to the matching member of the array or a null pointer if no match is found.

NULL

Indicates that the key cannot be found in the array.

Example

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

#define SSIZE 30

extern int compare();  /* prototype for comparison function */
```

```
int array[SSIZE] = {30, 1, 29, 2, 28, 3, 27, 4, 26, 5,
                    24, 6, 23, 7, 22, 8, 21, 9, 20, 10,
                    19, 11, 18, 12, 17, 13, 16, 14, 15, 25};

 /*  This program takes an unsorted array, sorts it using qsort, */
 /*  and then calls bsearch for each element in the array,       */
 /*  making sure that bsearch returns the correct element.       */

main()
{
    int i;
    int failure = FALSE;
    int *rkey;

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

    /* search for each element */
    for (i = 0; i < SSIZE - 1; i++) {
        /* search array element i */
        rkey = bsearch((array + i), array, SSIZE,
                       sizeof(array[0]), &compare);
        /* check for successful search */
        if (&array[i] != rkey) {
            printf("Not in array, array element %d\n", i);
            failure = TRUE;
            break;
        }
    }
    if (!failure)
        printf("All elements successfully found!\n");
}

 /*  Simple comparison routine.  */
 /*                              */
 /*  Returns:  = 0 if a == b     */
 /*            < 0 if a < b      */
 /*            > 0 if a > b      */

int compare(int *a, int *b)
{
    return (*a - *b);
}
```

This example program outputs the following:

```
All elements successfully found!
```

btowc

btowc — Converts a one-byte multibyte character to a wide character in the initial shift state.

Format

```
#include <wchar.h>
wint_t btowc (int c);
```

Argument

c

The character to be converted to a wide-character representation.

Description

The `btowc` function determines whether (`unsigned char`) *c* is a valid one-byte multibyte character in the initial shift state, and if so, returns a wide-character representation of that character.

Return Values

x

The wide-character representation of `unsigned char` *c*.

WEOF

Indicates an error. The *c* argument has the value EOF or does not constitute a valid one-byte multibyte character in the initial shift state.

bzero

bzero — Copies null characters into byte strings.

Format

```
#include <strings.h>
void bzero (void *string, size_t length);
```

Arguments

string

Specifies the byte string into which you want to copy null characters.

length

Specifies the length (in bytes) of the string.

Description

The `bzero` function copies null characters ('\0') into the byte string pointed to by *string* for *length* bytes. If *length* is 0 (zero), then no bytes are copied.

cabs

cabs — Returns the absolute value of a complex number.

Format

```
#include <math.h>
double cabs (cabs_t z);
float cabsf (cabsf_t z);
long double cabsl (cabsl_t z);
```

Argument

z

A structure of type `cabs_t`, `cabsf_t`, or `cabsl_t`. These types are defined in the `<math.h>` header file as follows:

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

typedef struct { float x, y; } cabsf_t;
typedef struct { long double x, y; } cabsl_t;
```

Description

The `cabs` functions return the absolute value of a complex number by computing the Euclidean distance between its two points as the square root of their respective squares:

$\texttt{sqrt}(x^2 + y^2)$

On overflow, the return value is undefined.

The `cabs`, `cabsf`, and `cabsl` functions are equivalent to the `hypot`, `hypotf`, and `hypotl` functions, respectively.

cacos

cacos — Returns the complex arc cosine of its argument.

Format

```
#include <complex.h>
double complex cacos (double complex z);
float complex cacosf (float complex z);
long double complex cacosl (long double complex z);
```

Argument

z

A complex value.

Description

The `cacos` functions compute the complex arc cosine of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

Return Values

n

The complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis.

cacosh

cacosh — Returns the complex arc hyperbolic cosine of its argument.

Format

```
#include <complex.h>
double complex cacosh (double complex z);
float complex cacoshf (float complex z);
long double complex cacoshl (long double complex z);
```

Argument

z

A complex value.

Description

The `cacosh` functions compute the complex arc hyperbolic cosine of *z*, with a branch cut at values less than 1 along the real axis.

Return Values

n

The complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval [-i π, +i π] along the imaginary axis.

calloc

calloc — Allocates an area of zeroed memory. This function is AST-reentrant.

Format

```
#include <stdlib.h>
void *calloc (size_t number, size_t size);
```

Function Variants

The `calloc` function has variants named `_calloc32` and `_calloc64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

number

The number of items to be allocated.

size

The size of each item.

Description

The `calloc` function initializes the items to 0. The maximum amount of memory allocated at once is limited to 0xFFFFD000.

See also `malloc` and `realloc`.

Return Values

x

The address of the first byte, which is aligned on a quadword boundary (Alpha only) or an octaword boundary (Integrity servers only).

NULL

Indicates an inability to allocate the space.

carg

carg — Returns the phase angle of its complex argument.

Format

```
#include <complex.h>
double carg (double complex z);
float cargf (float complex z);
long double cargl (long double complex z);
```

Argument

z

A complex value.

Description

The `carg` functions compute the argument (also called phase angle) of *z*, with a branch cut along the negative real axis.

Return Values

n

The value of the argument of *z*, in the interval [- π, + π].

casin

casin — Returns the complex arc sine of its argument.

Format

```
#include <complex.h>
double complex casin (double complex z);
float complex casinf (float complex z);
long double complex casinl (long double complex z);
```

Argument

z

A complex value.

Description

The `casin` functions compute the complex arc sine of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

Return Values

n

The complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [- π/2, + π/2] along the real axis.

casinh

casinh — Returns the complex arc hyperbolic sine of its argument.

Format

```
#include <complex.h>
double complex casinh (double complex z);
float complex casinhf (float complex z);
long double complex casinhl (long double complex z);
```

Argument

z

A complex value.

Description

The `casinh` functions compute the complex arc hyperbolic sine of *z*, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Return Values

n

The complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i π/2, +i π/2] along the imaginary axis.

catan

catan — Returns the complex arc tangent of its argument.

Format

```
#include <complex.h>
double complex catan (double complex z);
float complex catanf (float complex z);
long double complex catanl (long double complex z);
```

Argument

z

A complex value.

Description

The `catan` functions compute the complex arc tangent of *z*, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Return Values

n

The complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [- π/2, + π/2] along the real axis.

catanh

catanh — Returns the complex arc hyperbolic tangent of its argument.

Format

```
#include <complex.h>
double complex catanh (double complex z);
float complex catanhf (float complex z);
long double complex catanhl (long double complex z);
```

Argument

z

A complex value.

Description

The `catanh` functions compute the complex arc hyperbolic tangent of *z*, with branch cuts outside the interval [-1, +1] along the imaginary axis.

Return Values

n

The complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i π/2, +i π/2] along the imaginary axis.

catclose

catclose — Closes a message catalog.

Format

```
#include <nl_types.h>
int catclose (nl_catd catd);
```

Argument

catd

A message catalog descriptor. This is returned by a successful call to *catopen*.

Description

The `catclose` function closes the message catalog referenced by *catd* and frees the catalog file descriptor.

Return Values

0

Indicates that the catalog was successfully closed.

-1

Indicates that an error occurred. The function sets `errno` to the following value:

- EBADF – The catalog descriptor is not valid.

catgets

catgets — Retrieves a message from a message catalog.

Format

```
#include <nl_types.h>
char *catgets (nl_catd catd, int set_id, int msg_id, const char *s);
```

Function Variants

The `catgets` function has variants named `_catgets32` and `_catgets64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

catd

A message catalog descriptor. This is returned by a successful call to *catopen*.

set_id

An integer set identifier.

msg_id

An integer message identifier.

s

A pointer to a default message string that is returned by the function if the message cannot be retrieved.

Description

The `catgets` function retrieves a message identified by *set_id* and *msg_id*, in the message catalog *catd*. The message is stored in a message buffer in the `nl_catd` structure, which is overwritten by

subsequent calls to *catgets*. If a message string needs to be preserved, it should be copied to another location by the program.

Return Values

x

Pointer to the retrieved message.

s

Pointer to the default message string. Indicates that the function is not able to retrieve the requested message from the catalog. This condition can arise if the requested pair (*set_d, msg_id*) does not represent an existing message from the open catalog, or it indicates that an error occurred. If an error occurred, the function sets `errno` to one of the following values:

- EBADF – The catalog descriptor is not valid.
- EVMSRR – An OpenVMS I/O read error; the OpenVMS error code can be found in `vaxc$errno`.

Example

```
#include <nl_types.h>
#include <locale.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unixio.h>

 /* This test makes use of all the message catalog routines. catopen  */
 /* opens the catalog ready for reading, then each of the three       */
 /* messages in the catalog are extracted in turn using catgets and   */
 /* printed out. catclose closes the catalog after use.               */
 /* The catalog source file used to create the catalog is as follows: */
 /* $ This is a message file
 * $
 * $quote "
 * $ another comment line
 * $set 1
 * 1 "First set, first message"
 * 2 "second message - This long message uses a backslash \
 * for continuation."
 * $set 2
 * 1 "Second set, first message"                                      */

char *default_msg = "this is the first message.";

main()
{
  nl_catd catalog;
  int msg1,
      msg2,
      retval;

  char *cat = "sys$disk:[]catgets_example.cat"; /*Force local catalog*/
```

```
  char *msgtxt;

  char string[128];

  /* Create the message test catalog */

  system("gencat catgets_example.msgx catgets_example.cat") ;

  if ((catalog = catopen(cat, 0)) == (nl_catd) - 1) {
      perror("catopen");
      exit(EXIT_FAILURE);
  }

  msgtxt = catgets(catalog, 1, 1, default_msg);
  printf("%s\n", msgtxt);

  msgtxt = catgets(catalog, 1, 2, default_msg);
  printf("%s\n", msgtxt);

  msgtxt = catgets(catalog, 2, 1, default_msg);
  printf("%s\n", msgtxt);

  if ((retval = catclose(catalog)) == -1) {
      perror("catclose");
      exit(EXIT_FAILURE);
  }

  delete("catgets_example.cat;") ;  /* Remove the test catalog */
}
```

Running the example program produces the following result:

```
First set, first message
second message - This long message uses a backslash for
                                                    continuation.
Second set, first message
```

catopen

catopen — Opens a message catalog.

Format

```
#include <nl_types.h>
nl_catd catopen (const char *name, int oflag);
```

Arguments

name

The name of the message catalog to open.

oflag

An object of type `int` that determines whether the locale set for the LC_MESSAGES category in the current program's locale or the logical name LANG is used to search for the catalog file.

Description

The `catopen` function opens the message catalog identified by *name*.

If *name* contains a colon (:), a square opening bracket ([), or an angle bracket (<), or is defined as a logical name, then it is assumed that *name* is the complete file specification of the catalog.

If it does not include these characters, `catopen` assumes that *name* is a logical name pointing to an existing catalog file. If *name* is not a logical name, then the logical name NLSPATH is used to define the file specification of the message catalog. NLSPATH is defined in the user's process. If the NLSPATH logical name is not defined, or no message catalog can be opened in any of the components specified by the NLSPATH, then the SYS$NLSPATH logical name is used to search for a message catalog file.

Both NLSPATH and SYS$NLSPATH are comma-separated lists of templates. The `catopen` function uses each template to construct a file specification. For example, NLSPATH could be defined as:

```
DEFINE NLSPATH SYS$SYSROOT:[SYS$I18N.MSG]%N.CAT,SYS$COMMON:[SYSMSG]%N.CAT
```

In this example, `catopen` first searches the directory SYS$SYSROOT:[SYS$I18N.MSG] for the named catalog. If the named catalog is not found there, the directory SYS$COMMON:[SYSMSG] is searched. The catalog name is constructed by substituting %N with the name passed to `catopen`, and adding the .cat suffix. %N is known as a substitution field. The following substitution fields are valid:

Field	Meaning
%N	Substitute the *name* passed to `catopen`
%L[1]	Substitute the locale name. The period (.) and at-sign (@) characters in the locale name are replaced by an underscore (_) character. For example, the "zh_CN.dechanzi@radical" locale name results in a substitution of ZH_CN_DECHANZI_RADICAL.
%l[1]	Substitute the *language* part of the locale name. For example, the language part of the en_GB.ISO8859-1 locale name is en.
%t[1]	Substitute the *territory* part of the locale name. For example, the territory part of the en_GB.ISO8859-1 locale is GB.
%c[1]	Substitute the *codeset* name from the locale name. For example, the codeset name of the en_GB.ISO8859-1 locale name is ISO8859-1.

[1]This substitution assumes that the locale name is of the form *language_territory.codeset@mode*

If the *oflag* argument is set to NL_CAT_LOCALE, then the current locale as defined for the LC_MESSAGES category is used to determine the substitution for the %L, %l, %t, and %c substitution fields. If the *oflag* argument is set to 0, then the value of the LANG environment variable is used as a locale name to determine the substitution for these fields. Note that using NL_CAT_LOCALE conforms to the XPG4 specification while a value of 0 (zero) exists for the purpose of preserving XPG3 compatibility. Note also, that `catopen` uses the value of the LANG environment variable without checking whether the program's locale can be set using this value. That is, `catopen` does not check whether this value can serve as a valid locale argument in the `setlocale` call.

If the substitution value is not defined, an empty string is substituted.

A leading comma or two adjacent commas (,,) is equivalent to specifying %N. For example,

```
DEFINE NLSPATH ",%N.CAT,SYS$COMMON:[SYSMSG.%L]%N.CAT"
```

In this example, `catopen` searches in the following locations in the order shown:

1. *name* (in the current directory)
2. *name*.cat (in the current directory)
3. SYS$COMMON:[SYSMSG. *locale_name*] *name*.cat

Return Values

x

A message catalog file descriptor. Indicates the call was successful. This descriptor is used in calls to `catgets` and `catclose`.

(nl_catd) -1

Indicates an error occurred. The function sets `errno` to one of the following values:

- EACCES – Insufficient privilege or file protection violation, or file currently locked by another user.
- EMFILE – Process channel count exceeded.
- ENAMETOOLONG – The full file specification for message catalog is too long
- ENOENT – Unable to find the requested message catalog.
- ENOMEM – Insufficient memory available.
- ENOTDIR – Part of the *name* argument is not a valid directory.
- EVMSERR – An error occurred that does not match any `errno` value. Check the value of `vaxc$errno`.

cbrt

cbrt — Returns the rounded cube root of *y*.

Format

```
#include <math.h>
double cbrt (double y);
float cbrtf (float y);
long double cbrtl (long double y);
```

Argument

y

A real number.

ccos

ccos — Returns the complex cosine of its argument.

Format

```
#include <complex.h>
double complex ccos (double complex z);
float complex ccosf (float complex z);
long double complex ccosl (long double complex z);
```

Argument

z

A complex value.

Description

The `ccos` functions return the complex cosine of *z*.

Return Values

x

The complex cosine value.

ccosh

ccosh — Returns the complex hyperbolic cosine of its argument.

Format

```
#include <complex.h>
double complex ccosh (double complex z);
float complex ccoshf (float complex z);
long double complex ccoshl (long double complex z);
```

Argument

z

A complex value.

Description

The ccosh functions return the complex hyperbolic cosine of *z*.

Return Values

x

The complex hyperbolic cosine value.

ceil

ceil — Returns the smallest integer that is greater than or equal to its argument.

Format

```
#include <math.h>
double ceil (double x);
float ceilf (float x);
long double ceill (long double x);
```

Argument

x

A real value.

Return Value

n

The smallest integer greater than or equal to the function argument.

cexp

cexp — Returns the complex exponent of its argument.

Format

```
#include <complex.h>
double complex cexp (double complex z);
float complex cexpf (float complex z);
long double complex cexpl (long double complex z);
```

Argument

z

A complex value.

Description

The `cexp` functions compute the complex exponential value of *z*, defined as e** *z*, where e is the constant used as a base for natural logarithms.

Return Values

x

The complex exponential value of the argument.

cfree

cfree — Makes available for reallocation the area allocated by a previous `calloc`, `malloc`, or `realloc` call. This function is AST-reentrant.

Format

```
#include <stdlib.h>
void cfree (void *ptr);
```

Argument

ptr

The address returned by a previous call to `malloc`, `calloc`, or `realloc`.

Description

The contents of the deallocated area are unchanged.

In VSI C for OpenVMS systems, the `free` and `cfree` functions are equivalent. Some other C implementations use `free` with `malloc` or `realloc`, and `cfree` with `calloc`. However, since the ANSI C standard does not include `cfree`, using `free` may be preferable.

See also `free`.

chdir

chdir — Changes the default directory.

Format

```
#include <unistd.h>
int chdir (const char *dir_spec); (ISO POSIX-1)
int chdir (const char *dir_spec, ...); (VSI C Extension)
```

Arguments

dir_spec

A null-terminated character string naming a directory in either an OpenVMS or UNIX style specification.

...

This argument is an VSI C extension available when not defining any of the standards-related feature-test macros (see Section 1.4) and not compiling in strict ANSI C mode (/STANDARD=ANSI89). The argument is an optional flag of type `int` that is significant only when calling `chdir` from USER mode.

If the value of the flag is 1, the new directory is effective across images. If the value is not 1, the original default directory is restored when the image exits.

Description

The `chdir` function changes the default directory. The change can be permanent or temporary. Permanent means that the new directory remains as the default directory after the image exits. Temporary means that on image exit, the default is set to whatever it was before the execution of the image.

There are two ways of making the change permanent:

- Call `chdir` from USER mode with the second argument set to 1.
- Call `chdir` from SUPERVISOR or EXECUTIVE mode, regardless of the value of the second argument.

Otherwise, the change is temporary.

Return Values

0

Indicates that the directory is successfully changed to the given name.

-1

Indicates that the change attempt has failed.

chmod

chmod — Changes the file protection of a file.

Format

```
#include <stat.h>
int chmod (const char *file_spec, mode_tmode);
```

Arguments

file_spec

The name of an OpenVMS or UNIX style file specification.

mode

A file protection. Modes are constructed by performing a bitwise OR on any of the values shown in Table 34.

Table 34. File Protection Values and Their Meanings

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

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

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

Description

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

The C RTL does not support the S_ISVTX bit. Setting the S_ISVTX mode has no effect.

Return Values

0

Indicates that the mode is successfully changed.

-1

Indicates that the change attempt has failed.

chown

chown — Changes the user ID and group ID of the specified file.

Format

```
#include <unistd.h>
int chown (const char *file_spec, uid_t owner, gid_t group);
```

Arguments

file_spec

The address of an ASCII filename.

owner

The new user ID of the file.

group

The new group ID of the file.

Return Values

0

Indicates success.

-1

Indicates failure.

cimag

cimag — Returns the imaginary part of its complex argument.

Format

```
#include <complex.h>
double cimag (double complex z);
float cimagf (float complex z);
long double cimagl (long double complex z);
```

Argument

z

A complex value.

Description

The `cimag` functions return the imaginary part of *z* as a real number.

Return Values

x

The imaginary part value.

[w]clear

[w]clear — Erase the contents of the specified window and reset the cursor to coordinates (0,0). The `clear` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int clear();
int wclear (WINDOW *win);
```

Argument

win

A pointer to the window.

Return Values

OK

Indicates success.

ERR

Indicates an error.

clearerr

clearerr — Resets the error and end-of-file indicators for a file (so that `ferror` and `feof` will not return a nonzero value).

Format

```
#include <stdio.h>
void clearerr (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

clearerr_unlocked

clearerr_unlocked — Same as the `clearerr` function, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

```
#include <stdio.h>
void clearerr_unlocked (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The reentrant version of the `clearerr` function is locked against multiple threads calling it simultaneously. This incurs overhead to ensure integrity of the stream. The unlocked version of this call, `clearerr_unlocked` can be used to avoid the overhead. The `clearerr_unlocked` macro is functionally identical to the `clearerr` macro, except that it is not required to be implemented in a thread-safe manner. The `clearerr_unlocked` function can be safely used only within a scope that is protected by the `flockfile` and `funlockfile` functions used as a pair. The caller must ensure that the stream is locked before `clearerr_unlocked` is used.

See also `flockfile`, `ftrylockfile`, and `funlockfile`.

clearok

clearok — Sets the clear flag for the window.

Format

```
#include <curses.h>
clearok (WINDOW *win, bool boolf);
```

Arguments

win

The entire size of the terminal screen. You can use the windows `stdscr` and `curscr` with `clearok`.

boolf

A Boolean value of TRUE or FALSE. If the argument is TRUE, this forces a clear screen to be printed on the next call to `refresh`, or stops the screen from being cleared if *boolf* is FALSE.

The type `bool` is defined in the `<curses.h>` header file as follows:

```
#define bool int
```

Description

Unlike the `clear` function, the `clearok` function does not alter the contents of the window. If the *win* argument is `curscr`, the next call to `refresh` causes a clear screen, even if the window passed to `refresh` is not a window the size of the entire terminal screen.

clock

clock — Determines the CPU time (in 10-millisecond units) used since the beginning of the process. The time reported is the sum of the user and system times of the calling process and any terminated child processes for which the calling process has executed `wait` or `system`.

Format

```
#include <time.h>
clock_t clock (void);
```

Description

The value returned by the `clock` function must be divided by the value of the CLK_TCK, as defined in the standard header file `<time.h>`, to obtain the time in seconds.

The type `clock_t` is defined in the `<time.h>` header file as follows:

```
typedef long int clock_t;
```

Only the accumulated times for child processes running a C main program or a program that calls VAXC$CRTL_INIT or DECC$CRTL_INIT are included.

A typical usage of the `clock` function is to call it after a program does its initial setup, and then again after the program executes the code to be timed. Then subtract the two values to give elapsed CPU time.

Return Values

n

The processor time used.

-1

Indicates that the processor time used is not available.

clock_getres

clock_getres — Gets the resolution for the specified clock.

Format

```
#include <time.h>
int clock_getres (clockid_t clock_id, struct timespec *res);
```

Arguments

clock_id

The clock type used to obtain the resolution. The CLOCK_REALTIME clock is supported and represents the TIME-OF-DAY clock for the system.

res

A pointer to the `timespec` data structure that receives the value of the clock's resolution.

Description

The `clock_getres` function obtains the resolution value for the specified clock. Clock resolutions are implementation-dependent and cannot be set by a process.

If the *res* argument is not NULL, the resolution of the specified clock is stored in the location pointed to by *res*.

If *res* is NULL, the clock resolution is not stored.

If the time argument (*tp*) of `clock_settime` is not a multiple of *res*, then the value is truncated to a multiple of *res*.

On success, the function returns 0.

On failure, the function returns -1 and sets `errno` to indicate the error.

See also `clock_gettime`, `clock_settime`, `time`, and `ctime`.

Return Values

0

Indicates success.

-1

Indicates failure; `errno` is set to the following value:

- EINVAL– The *clock_id* argument does not specify a known clock.

clock_gettime

clock_gettime — Returns the current time (in seconds and nanoseconds) for the specified clock.

Format

```
#include <time.h>
int clock_gettime (clockid_t clock_id, struct timespec *tp);
```

Arguments

clock_id

The clock type used to obtain the time for the clock that is set. The CLOCK_REALTIME clock is supported and represents the TIME-OF-DAY clock for the system. CLOCK_MONOTONIC, CLOCK_MONOTONIC_COARSE, and CLOCK_MONOTONIC_RAW are also supported.

tp

A pointer to a `timespec` data structure.

Description

The `clock_gettime` function returns the current *tp* value for the specified clock, *clock_id*.

On success, the function returns 0.

On failure, the function returns -1 and sets `errno` to indicate the error.

See also `clock_getres`, `clock_settime`, `time`, and `ctime`.

Return Values

0

Indicates success.

-1

Indicates failure; `errno` is set to the following value:

- EINVAL– The *clock_id* argument does not specify a known clock, or the *tp* argument specifies a nanosecond value less than 0 or greater than or equal to 1 billion.

clock_settime

clock_settime — Sets the specified clock.

Format

```
#include <time.h>
int clock_settime (clockid_t clock_id, const struct timespec *tp);
```

Arguments

clock_id

The clock type used for the clock that is to be set. The CLOCK_REALTIME clock is supported and represents the TIME-OF-DAY clock for the system.

tp

A pointer to a `timespec` data structure.

Description

The `clock_settime` function sets the specified clock, *clock_id*, to the value specified by *tp*. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock are truncated down to the smaller multiple of the resolution.

A clock can be systemwide (that is, visible to all processes) or per-process (measuring time that is meaningful only within a process).

The CLOCK_REALTIME clock, defined in `<time.h>`, represents the real time clock for the system. For this clock, the values specified by `clock_settime` and returned by `clock_gettime` represent the amount of time elapsed, in seconds and nanoseconds, since the Epoch. The Epoch is defined as 00:00:00:00 January 1, 1970 Greenwich Mean Time (GMT).

You must have OPER, LOG_IO, and SYSPRV privileges to use the `clock_settime` function.

On success, the function returns 0.

On failure, the function returns -1 and sets `errno` to indicate the error.

See also `clock_getres`, `clock_gettime`, `time`, and `ctime`.

Return Values

0

Indicates success.

-1

Indicates failure; `errno` is set to the following value:

- EINVAL– The *clock_id* argument does not specify a known clock, or the *tp* argument is outside the range for the given *clock_id* or specifies a nanosecond value less than 0 or greater than or equal to 1 billion.
- EPERM– The requesting process does not have the appropriate privilege to set the specified clock.

clog

clog — Returns the complex natural (base e) logarithm of its argument.

Format

```
#include <complex.h>
double complex clog (double complex z);
float complex clogf (float complex z);
long double complex clogl (long double complex z);
```

Argument

z

A complex value.

Description

The `clog` functions return the complex natural (base e) logarithm of *z*, with a branch cut along the negative real axis.

Return Values

x

The complex natural logarithm value in the range of a strip mathematically unbounded along the real axis and in the interval [-i π, +i π] along the imaginary axis.

close

close — Closes the file associated with a file descriptor.

Format

```
#include <unistd.h>
int close (int file_desc);
```

Argument

file_desc

A file descriptor.

Description

The `close` function tries to write buffered data by using an implicit call to `fflush`. If the write fails (because the disk is full or the user's quota was exceeded, for example), `close` continues executing. It closes the OpenVMS channel, deallocates any buffers, and releases the memory associated with the file descriptor (or FILE pointer). Any buffered data is lost, and the file descriptor (or FILE pointer) no longer refers to the file.

If your program needs to recover from errors when flushing buffered data, it should make an explicit call to `fsync` (or `fflush`) before calling `close`.

Return Values

0

Indicates that the file is properly closed.

-1

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

Example

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

closedir

closedir — Closes directories.

Format

```
#include <dirent.h>
int closedir (DIR *dir_pointer);
```

Argument

dir_pointer

Pointer to the `dir` structure of an open directory.

Description

The `closedir` function closes a directory stream and frees the structure associated with the *dir_pointer* argument. Upon return, the value of *dir_pointer* does not necessarily point to an accessible object of the type `DIR`.

The type `DIR`, which is defined in the `<dirent.h>` header file, represents a directory stream that is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files. You can remove files from or add files to a directory asynchronously to the operation of the `readdir` function.

Note

An open directory must always be closed with the `closedir` function to ensure that the next attempt to open the directory is successful.

Example

The following example shows how to search a directory for the entry name, using the `opendir`, `readdir`, and `closedir` functions:

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

#define FOUND     1
#define NOT_FOUND 0

static int dir_example(const char *name, unsigned int unix_style)
{
    DIR *dir_pointer;
    struct dirent *dp;

    if ( unix_style )
        dir_pointer = opendir(".");
    else
        dir_pointer = opendir(getenv("PATH"));

    if ( !dir_pointer ) {
        perror("opendir");
        return NOT_FOUND;
    }

    /* Note, that if opendir() was called with UNIX style file  */
    /* spec like ".", readdir() will return only a single       */
    /* version of each file in the directory. In this case the  */
    /* name returned in d_name member of the dirent structure   */
    /* will contain only file name and file extension fields,   */
```

```
    /* both lowercased like "foo.bar".                              */

    /* If opendir() was called with OpenVMS style file spec,        */
    /* readdir() will return every version of each file in the      */
    /* directory. In this case the name returned in d_name          */
    /* member of the dirent structure will contain file name,       */
    /* file extension and file version fields. All in upper         */
    /* case, like "FOO.BAR;1".                                      */

    for ( dp = readdir(dir_pointer);
          dp && strcmp(dp->d_name, name);
          dp = readdir(dir_pointer) )
        ;

    closedir(dir_pointer);

    if ( dp != NULL )
        return FOUND;
    else
        return NOT_FOUND;
}

int main(void)
{
   char *filename = "foo.bar";
   FILE *fp;

   remove(filename);

   if ( !(fp = fopen(filename, "w")) ) {
        perror("fopen");
        return (EXIT_FAILURE);
   }

   if ( dir_example( "FOO.BAR;1", 0 ) == FOUND )
        puts("OpenVMS style: found");
   else
        puts("OpenVMS style: not found");

   if ( dir_example( "foo.bar", 1 ) == FOUND )
        puts("UNIX style: found");
   else
        puts("UNIX style: not found");

   fclose(fp);
   remove(filename);
   return( EXIT_SUCCESS );
}
```

Return Values

0

Indicates success.

-1

Indicates an error and is further specified in the global `errno`.

[w]clrattr

[w]clrattr — Deactivate the video display attribute *attr* within the window. The `clrattr` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int clrattr (int attr);
int wclrattr (WINDOW *win, int attr);
```

Arguments

win

A pointer to the window.

attr

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

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

Description

These functions are specific to VSI C for OpenVMS systems and are not portable.

Return Values

OK

Indicates success.

ERR

Indicates an error.

[w]clrtobot

[w]clrtobot — Erase the contents of the window from the current position of the cursor to the bottom of the window. The `clrtobot` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int clrtobot();
int wclrtobot (WINDOW *win);
```

Argument

win

A pointer to the window.

Return Values

OK

Indicates success.

ERR

Indicates an error.

[w]clrtoeol

[w]clrtoeol — Erase the contents of the window from the current cursor position to the end of the line on the specified window. The `clrtoeol` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int clrtoeol();
int wclrtoeol (WINDOW *win);
```

Argument

win

A pointer to the window.

Return Values

OK

Indicates success.

ERR

Indicates an error.

confstr

confstr — Determines the current value of a specified system variable defined by a string value.

Format

```
#include <unistd.h>
size_t confstr (int name, char *buf, size_t len);
```

Arguments

name

The system variable setting. Valid values for the *name* argument are the _CS_X names defined in the `<unistd.h>` header file.

buf

Pointer to the buffer where the `confstr` function copies the *name* value.

len

The size of the buffer storing the *name* value.

Description

The `confstr` function allows an application to determine the current setting of certain system parameters, limits, or options that are defined by a string value. The function is mainly used by applications to find the system default value for the `PATH` environment variable.

If the following conditions are true, then the `confstr` function copies that value into a *len*-byte buffer pointed to by *buf*:

- The *len* argument can be 0 (zero).
- The *name* argument has a system-defined value.
- The *buf* argument is not a NULL pointer.

If the returned string is longer than *len* bytes, including the terminating null, then the `confstr` function truncates the string to *len* - 1 bytes and adds a terminating null to the result. The application can detect that the string was truncated by comparing the value returned by the `confstr` function with the value of the *len* argument.

The `<limits.h>` header file contains system-defined limits. The `<unistd.h>` header file contains system-defined environmental variables.

Also, `confstr` supports the following three HP-UX symbolic constants, which are added to header file `<unistd.h>`:

- _CS_MACHINE_IDENT
- _CS_PARTITION_IDENT
- _CS_MACHINE_SERIAL

Example

To find out how big a buffer is needed to store the string value of *name*, enter:

```
confstr(_CS_PATH, NULL, (size_t) 0)
```

The `confstr` function returns the size of the buffer necessary.

Return Values

0

Indicates an error. When the specified *name* value:

- Is invalid, `errno` is set to EINVAL.
- Does not have a system-defined value, `errno` is not set.

n

The size of the buffer needed to hold the value.

- When the value of the *name* argument is system-defined, `confstr` returns the size of the buffer needed to hold the entire value. If this return value is greater than the *len* value, the string returned as the *buf* value is truncated.
- When the value of the *len* argument is set to 0 or the *buf* value is NULL, `confstr` returns the size of the buffer needed to hold the entire system-defined value. The string value is not copied.

conj

conj — Returns the complex conjugate of its argument.

Format

```
#include <complex.h>
double complex conj (double complex z);
float complex conjf (float complex z);
long double complex conjl (long double complex z);
```

Argument

z

A complex value.

Description

The `conj` functions return the complex conjugate of *z*, by reversing the sign of its imaginary part.

Return Values

x

The complex conjugate value.

copysign

copysign — Returns *x* with the same sign as *y*.

Format

```
#include <math.h>
double copysign (double x, double y);
```

```
float copysignf (float x, float y);
long double copysignl (long double x, long double y);
```

Arguments

x

A real value.

y

A real value.

Description

The `copysign` functions return *x* with the same sign as *y*. IEEE 754 requires `copysign`(*x*,NaN), `copysignf`(*x*,NaN), and `copysignl`(*x*,NaN) to return + *x* or - *x*.

Return Value

x

The value of *x* with the same sign as *y*.

cos

cos — Returns the cosine of its radian argument.

Format

```
#include <math.h>
double cos (double x);
float cosf (float x);
long double cosl (long double x);
double cosd (double x);
float cosdf (float x);
long double cosdl (long double x);
```

Argument

x

A radian expressed as a real value.

Description

The `cos` functions return the cosine of their argument, measured in radians.

The `cosd` functions return the cosine of their argument, measured in degrees.

|*x*| = Infinity is an invalid argument.

Return Values

x

The cosine of the argument.

HUGE_VAL

Indicates that the argument is too large; `errno` is set to ERANGE.

cosh

cosh — Returns the hyperbolic cosine of its radian argument.

Format

```
#include <math.h>
double cosh (double x);
float coshf (float x);
long double coshl (long double x);
```

Argument

x

A radian expressed as a real number.

Description

The `cosh` functions return the hyperbolic cosine of *x* and are defined as (e** *x* + e**(- *x*))/2.

Return Values

x

The hyperbolic cosine of the argument.

HUGE_VAL

Indicates that the argument is too large; `errno` is set to ERANGE.

cot

cot — Returns the cotangent of its radian argument.

Format

```
#include <math.h>
double cot (double x);
float cotf (float x);
long double cotl (long double x);
```

```
double cotd (double x);
float cotdf (float x);
long double cotdl (long double x);
```

Argument

x

A radian expressed as a real number.

Description

The `cot` functions return the cotangent of their argument, measured in radians.

The `cotd` functions return the cotangent of their argument, measured in degrees.

x = 0 is an invalid argument.

Return Values

x

The cotangent of the argument.

HUGE_VAL

Indicates that the argument is zero; `errno` is set to ERANGE.

cpow

cpow — Returns the complex power function *x*** *y*.

Format

```
#include <complex.h>
double complex cpow (double complex x, double complex y);
float complex cpowf (float complex x, float complex y);
long double complex cpowl (long double complex x, long double complex y);
```

Argument

x

A complex value.

y

A complex value.

Description

The `cpow` functions return the complex power function *x*** *y*, with a branch cut for the first parameter along the negative real axis.

Return Values

x

The complex power function value.

cproj

cproj — Returns a projection of its argument onto the Riemann sphere.

Format

```
#include <complex.h>
double complex cproj (double complex z);
float complex cprojf (float complex z);
long double complex cprojl (long double complex z);
```

Argument

z

A complex value.

Description

The `cproj` functions compute and return a projection of *z* onto the Riemann sphere: *z* projects to *z*, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If *z* has an infinite part, then `cproj(z)` is equivalent to:

INFINITY + I * `copysign`(0.0, `cimag`(*z*))

Return Values

x

The value of the projection onto the Riemann sphere.

creal

creal — Returns the real part of its complex argument.

Format

```
#include <complex.h>
double creal (double complex z);
float crealf (float complex z);
long double creall (long double complex z);
```

Argument

z

A complex value.

Description

The `creal` functions return the real part of *z*.

Return Values

x

The real part value.

creat

creat — Creates a new file.

Format

```
#include <fcntl.h>
int creat (const char *file_spec, mode_t mode); (ISO POSIX-1)
int creat (const char *file_spec, mode_t mode, ...); (VSI C Extension)
```

Arguments

file_spec

A null-terminated string containing any valid file specification.

mode

An unsigned value that specifies the file-protection mode. The compiler performs a bitwise AND operation on the mode and the complement of the current protection mode.

You can construct modes by using the bitwise OR operator (|) to create mode combinations. The modes are:

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

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

Note

To create files with OpenVMS RMS default protections using the UNIX system-call functions `umask`, `mkdir`, `creat`, and `open`, call `mkdir`, `creat`, and `open` with a file-protection mode argument of 0777 in a program that never specifically calls `umask`. These default protections include correctly establishing protections based on ACLs, previous versions of files, and so on.

In programs that do `vfork`/ `exec` calls, the new process image inherits whether `umask` has ever been called or not from the calling process image. The `umask` setting and whether the `umask` function has ever been called are both inherited attributes.

...

An optional argument list of character strings of the following form:

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

Or in the case of "acc" or "err", this form:

```
"keyword"
```

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

The RMS callback keywords "acc" and "err" are the only keywords that do not take values. Instead, they are followed by a pointer to the callback routine to be used, followed by a pointer to a user-specified value to be used as the first argument of the callback routine. For example, to set up an access callback routine called `acc_callback` whose first argument is a pointer to the integer variable `first_arg` in a call to `open`, you can use the following statement:

```
open("file.dat", O_RDONLY, 0 ,"acc", acc_callback, &first_arg)
```

The second and third arguments to the callback routine must be pointers to a FAB and RAB, respectively, and the routine must have a return type of `int`. If the callback returns a value less than 0, the `open`, `creat`, or `fopen` fails. The error callback can correct the error condition and return a status greater than or equal to 0 to continue the `creat` call. Assuming the previous `open` statement, the function prototype for `acc_callback` would be similar to the following statement:

```
#include <rms.h>
int acc_callback(int *first_arg, struct FAB *fab, struct RAB *rab);
```

FAB and RAB are defined in the `<rms.h>` header file, and the actual pointers passed to the routine are pointers to the RAB and FAB being used to open the file file.dat.

If an access callback routine is established, then it will be called in the open-type routine immediately before the call to the RMS function sys$create or sys$open. If an error callback routine is established and an error status is returned from the sys$create or sys$open function, then the callback routine will be invoked immediately after the status is checked and the error value is discovered.

Note

Any manipulation of the RAB or FAB in a callback function could lead to serious problems in later calls to the C RTL I/O functions.

Table 35 describes the RMS keywords and values.

Table 35. RMS Valid Keywords and Values

Keyword	Value	Description
“acc”	callback	Access callback routine.
“alq = n”	decimal	Allocation quantity.
“bls = n”	decimal	Block size.
“ctx = bin”	string	No translation of ' \n' to the terminal. Use this for writing binary data to files.
“ctx = cvt”	string	Negates a previous setting of “ctx=nocvt”. This is the default.
“ctx = nocvt”	string	No conversion of Fortran carriage-control bytes.
“ctx = rec”	string	Forces record mode access.
“ctx = stm”	string	Forces stream mode access.
“ctx = xplct”	string	Causes records to be written only when explicitly specified by a call to `fflush`, `close`, or `fclose`.
“deq = n”	decimal	Default extension quantity.
“dna = filespec”	string	Default file-name string.
“err”	callback	Error callback routine.
“fop = val, val , …”		File-processing options:
	ctg	Contiguous.
	cbt	Contiguous-best-try.
	dfw	Deferred write; only applicable to files opened for shared access.
	dlt	Delete file on close.
	tef	Truncate at end-of-file.
	cif	Create if nonexistent.
	sup	Supersede.
	scf	Submit as command file on close.
	spl	Spool to system printer on close.
	tmd	Temporary delete.
	tmp	Temporary (no file directory).
	nef	Not end-of-file.
	rck	Read check compare operation.
	wck	Write check compare operation.
	mxv	Maximize version number.
	rwo	Rewind file on open.
	pos	Current position.
	rwc	Rewind file on close.
	sqo	File can only be processed in a sequential manner.
“fsz = n”	decimal	Fixed header size.
“gbc = n”	decimal	The requested number of global buffers for a file.

Keyword	Value	Description
“mbc = n”	decimal	Multiblock count.
“mbf = n”	decimal	Multibuffer count.
“mrs = n”	decimal	Maximum record size.
“pmt=usr-prmpt”	string	Prompts for terminal input. Any RMS input from a terminal device will be preceded by “usr-prmpt”when this option and “rop=pmt” are specified.
“rat = val, val …”		Record attributes:
	cr	Carriage-return control.
	blk	Disallow records to span block boundaries.
	ftn	Fortran print control.
	none	Explicitly forces no carriage control.
	prn	Print file format.
“rfm = val”		Record format:
	fix	Fixed-length record format.
	stm	RMS stream record format.
	stmlf	Stream format with line-feed terminator.
	stmcr	Stream format with carriage-return terminator.
	var	Variable-length record format.
	vfc	Variable-length record with fixed control.
	udf	Undefined.
“rop = val, val …”		Record-processing operations:
	asy	Asynchronous I/O.
	cco	Cancels Ctrl/O (used with Terminal I/O).
	cvt	Capitalizes characters on a read from the terminal.
	eof	Positions the record stream to the end-of-file for the connect operation only.
	nlk	Do not lock record.
	pmt	Enables use of the prompt specified by “pmt=usr-prmpt”on input from the terminal.
	pta	Eliminates any information in the type-ahead buffer on a read from the terminal.
	rea	Locks record for a read operation for this process, while allowing other accessors to read the record.
	rlk	Locks record for write.
	rne	Suppresses echoing of input data on the screen as it is entered on the keyboard.
	rnf	Indicates that Ctrl/U, Ctrl/R, and DELETE are not to be considered control commands on terminal input, but are to be passed to the application program.
	rrl	Reads regardless of lock.

Keyword	Value	Description
	syncsts	Returns a success status of RMS$_SYNCH if the requested service completes its task immediately.
	tmo	Timeout I/O.
	tpt	Allows put/write services using sequential record access mode to occur at any point in the file, truncating the file at that point.
	ulk	Prohibits RMS from automatically unlocking records.
	wat	Wait until record is available, if currently locked by another stream.
	rah	Read ahead.
	wbh	Write behind.
“rtv=n”	decimal	The number of retrieval pointers that RMS has to maintain in memory (0 to 127,255).
“shr = val, val, …”		File sharing options:
	del	Allows users to delete.
	get	Allows users to read.
	mse	Allows multistream connects.
	nil	Prohibits file sharing.
	put	Allows users to write.
	upd	Allows users to update.
	upi	Allows one or more writers.
	nql	No query locking (file level).
“tmo = n”	decimal	I/O timeout value.

In addition to these options, any option that takes a key value (such as “fop” or “rat”) can be negated by prefixing the value with “no”. For example, specify “fop=notmp” to clear the “tmp” bit in the “fop” field.

Notes

- While these options provide much flexibility and functionality, many of them can also cause severe problems if not used correctly.
- You cannot share the default VSI C for OpenVMS stream file I/O. If you wish to share files, you must specify “ctx=rec” to force record access mode. You must also specify the appropriate “shr” options depending on the type of access you want.
- If you intend to share a file opened for append, you must specify appropriate share and record-locking options, to allow other accessors to read the record. The reason for doing this: the file is positioned at the end-of-file by reading records in a loop until end-of-file is reached.

For more information on these options, see the *VSI OpenVMS Record Management Services Reference Manual*.

Description

The C RTL opens the new file for reading and writing, and returns the corresponding file descriptor.

If the file exists:

- A version number one greater than any existing version is assigned to the newly created file.
- By default, the new file inherits certain attributes from the existing version of the file unless those attributes are specified in the `creat` call. The following attributes are inherited:
  - Record format (FAB$B_RFM)
  - Maximum record size (FAB$W_MRS)
  - Carriage control (FAB$B_RAT)
  - File protection
- When a new version of a file is created, and the named file already exists as a symbolic link, the file to which the symbolic link refers is created.

If the file did not previously exist:

- It is given the file protection that results from performing a bitwise AND on the *mode* argument and the complement of the current protection mask.
- It defaults to stream format with line-feed record separator and implied carriage-return attributes.

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

Return Values

n

A file descriptor.

-1

Indicates errors, including protection violations, undefined directories, and conflicting file attributes.

[no]crmode

[no]crmode — In the UNIX system environment, the `crmode` and `nocrmode` functions set and unset the terminal from cbreak mode. In cbreak mode, a single input character can be processed without pressing Return. This mode of single-character input is only supported with the Curses input routine `getch`.

Format

```
#include <curses.h>
crmode()
nocrmode()
```

Example

```
/* Program to demonstrate the use of crmod() and curses */

#include <curses.h>

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

    /*  Initialize standard screen, turn echo off.  */
    initscr();
    noecho();

    /*  Define a user window.  */
    win1 = newwin(22, 78, 1, 1);

    /*  Turn on reverse video and draw a box on border.  */
    setattr(_REVERSE);
    box(stdscr, vert, hor);
    mvwaddstr(win1, 2, 2, "Test cbreak input");
    refresh();
    wrefresh(win1);

    /*  Set cbreak, do some input, and output it.  */
    crmode();
    getch();
    nocrmode(); /* Turn off cbreak. */
    mvwaddstr(win1, 5, 5, str);
    mvwaddstr(win1, 7, 7, "Type something to clear the screen");
    wrefresh(win1);

    /*  Get another character, then delete the window.  */
    getch();
    wclear(win1);
    touchwin(stdscr);
    endwin();
}
```

In this example, the first call to `getch` returns as soon as one character is entered, because `crmode` was called before `getch` was called. The second time `getch` is called, it waits until the Return key is pressed before processing the character entered, because `nocrmode` was called before `getch` was called the second time.

crypt

crypt — The password encryption function.

Format

```
#include <unistd.h>
#include <stdlib.h>
char *crypt (const char *key, const char *salt;)
```

Function Variants

The `crypt` function has variants named `_crypt32` and `_crypt64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

key

A user's typed password.

salt

A 2-character string.

Description

The `crypt` function generates an encoded version of a password. It is based on the NBS Data Encryption Standard, with variations intended to frustrate use of hardware implementations of the DES for key search.

The first argument to `crypt` is normally a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The return value from `crypt` points to a static data area whose content is overwritten by each call.

See also `encrypt` and `setkey`.

Return Value

pointer

Pointer to the encrypted password.

csin

csin — Returns the complex sine of its argument.

Format

```
#include <complex.h>
double complex csin (double complex z);
float complex csinf (float complex z);
long double complex csinl (long double complex z);
```

Argument

z

A complex value.

Description

The `csin` functions compute the complex sine value of *z*.

Return Values

x

The complex sine value.

csinh

csinh — Returns the complex hyperbolic sine of its argument.

Format

```
#include <complex.h>
double complex csinh (double complex z);
float complex csinhf (float complex z);
long double complex csinhl (long double complex z);
```

Argument

z

A complex value.

Description

The `csinh` functions compute the complex hyperbolic sine of *z*.

Return Values

x

The complex hyperbolic sine value.

csqrt

csqrt — Returns the complex square root of its argument.

Format

```
#include <complex.h>
double complex csqrt (double complex z);
float complex csqrtf (float complex z);
long double complex csqrtl (long double complex z);
```

Argument

z

A complex value.

Description

The `csqrt` functions compute the complex square root of *z*, with a branch cut along the negative real axis.

Return Values

x

The complex square root value in the range of the right half-plane (including the imaginary axis).

ctan

ctan — Returns the complex tangent of its argument.

Format

```
#include <complex.h>
double complex ctan (double complex z);
float complex ctanf (float complex z);
long double complex ctanl (long double complex z);
```

Argument

z

A complex value.

Description

The `ctan` functions compute the complex tangent value of *z*.

Return Values

x

The complex tangent value.

ctanh

ctanh — Returns the complex hyperbolic tangent of its argument.

Format

```
#include <complex.h>
double complex ctanh (double complex z);
float complex ctanhf (float complex z);
long double complex ctanhl (long double complex z);
```

Argument

z

A complex value.

Description

The `ctanh` functions compute the complex hyperbolic tangent value of *z*.

Return Values

x

The complex hyperbolic tangent value.

ctermid

ctermid — Returns a character string giving the equivalence string of SYS$COMMAND. This is the name of the controlling terminal.

Format

```
#include <stdio.h>
char *ctermid (char *str);
```

Function Variants

The `ctermid` function has variants named `_ctermid32` and `_ctermid64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

str

Must be a pointer to an array of characters. If this argument is NULL, the filename is stored internally and might be overwritten by the next `ctermid` call. Otherwise, the filename is stored beginning at the location indicated by the argument. The argument must point to a storage area of length `L_ctermid` (defined by the `<stdio.h>` header file).

Return Value

pointer

Points to a character string.

ctime, ctime_r

ctime, ctime_r — 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);
char *ctime_r (const time_t *bintim, char *buffer); (ISO POSIX-1)
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to this function that is equivalent to the behavior before OpenVMS Version 7.0.

Arguments

bintim

A pointer to a variable that specifies the time value (in seconds) to be converted.

buffer

A pointer to a character array that is at least 26 bytes long. This array is used to store the generated date-and-time string.

Description

The `ctime` and `ctime_r` functions convert the time pointed to by `bintim` into a 26-character string, and return a pointer to the string.

The difference between the `ctime_r` and `ctime` functions is that the former puts its result into a user-specified buffer. The latter puts its result into thread-specific static memory allocated by the C RTL, which can be overwritten by subsequent calls to `ctime` or `asctime`; you must make a copy if you want to save it.

On success, `ctime` returns a pointer to the string; `ctime_r` returns its second argument. On failure, these functions return the NULL pointer.

The type `time_t` is defined in the `<time.h>` header file as follows:

```
typedef long int time_t
```

The `ctime` function behaves as if it called `tzset`.

Note

Generally speaking, UTC-based time functions can affect in-memory time-zone information, which is processwide data. However, if the system time zone remains the same during the execution of the application (which is the common case) and the cache of timezone files is enabled (which is the default), then the `_r` variant of the time functions `asctime_r`, `ctime_r`, `gmtime_r`, and `localtime_r`, is both thread-safe and AST-reentrant.

If, however, the system time zone can change during the execution of the application or the cache of timezone files is not enabled, then both variants of the UTC-based time functions belong to the third class of functions, which are neither thread-safe nor AST-reentrant.

Return Values

x

A pointer to the 26-character ASCII string, if successful.

NULL

Indicates failure.

cuserid

cuserid — Returns a pointer to a character string containing the name of the user initiating the current process.

Format

```
#include <unistd.h> (X/Open, POSIX-1)
#include <stdio.h> (X/Open)
char *cuserid (char *str);
```

Function Variants

The `cuserid` function has variants named `_cuserid32` and `_cuserid64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

str

If this argument is NULL, the user name is stored internally. If the argument is not NULL, it points to a storage area of length `L_cuserid` (defined by the `<stdio.h>` header file), and the name is written into that storage. If the user name is a null string, the function returns NULL.

Return Values

pointer

Points to a string.

NULL

If the user name is a null string.

DECC$CRTL_INIT

DECC$CRTL_INIT — Allows you to call the C RTL from other languages or to use the C RTL when your main function is not in C. It initializes the run-time environment and establishes both an exit and condition handler. `VAXC$CRTL_INIT` is a synonym for `DECC$CRTL_INIT`. Either name invokes the same routine.

Format

```
#include <signal.h>
void DECC$CRTL_INIT(void);
```

Description

The following example shows a Pascal program that calls the C RTL using the `DECC$CRTL_INIT` function:

```
$ PASCAL EXAMPLE1
$ LINK EXAMPLE1
$ TY EXAMPLE1.PAS
PROGRAM TESTC(input, output);
PROCEDURE DECC$CRTL_INIT; extern;
BEGIN
   DECC$CRTL_INIT;
END
```

A shareable image need only call this function if it contains an VSI C function for signal handling, environment variables, I/O, exit handling, a default file protection mask, or if it is a child process that should inherit context.

Although many of the initialization activities are performed only once, `DECC$CRTL_INIT` can safely be called multiple times.

At least one frame in the current call stack must have that handler established for OpenVMS exceptions to get mapped to UNIX signals.

decc$feature_get

decc$feature_get — Calls `decc$feature_get_value` with a character-string feature name, rather than an index.

Format

```
#include <unixlib.h>
int decc$feature_get (const char *name, int mode);
```

Argument

name

Pointer to a character string passed as a name in the list of supported features.

mode

An integer indicating which feature value to return. The values for mode are:

_ _FEATURE_MODE_DEFVAL - Default value
_ _FEATURE_MODE_CURVAL - Current value
_ _FEATURE_MODE_MINVAL - Minimum value
_ _FEATURE_MODE_MAXVAL - Maximum value
_ _FEATURE_MODE_INIT_STATE - Initialization state

Description

The decc$feature_get function allows you to call the decc$feature_get_value function with a character-string feature name, rather than an index into an internal C RTL table.

On error, -1 is returned and errno is set to indicate the error.

See also decc$feature_get_value, decc$feature_get_index, decc$feature_get_name, decc$feature_set, decc$feature_set_value, decc$feature_show, and decc$feature_show_all.

Return Values

n

An integer corresponding to the specified *name* and *mode* arguments.

-1

Indicates an error; errno is set.

decc$feature_get_index

decc$feature_get_index — Returns an index for accessing feature values.

Format

```
#include <unixlib.h>
int decc$feature_get_index (char *name);
```

Argument

name

Pointer to a character string passed as a name in the list of supported features.

Description

The decc$feature_get_index function looks up the string passed as *name* in the list of supported features. If the name is found, decc$feature_get_index returns a (nonnegative) index that can be used to set or retrieve the values for the feature. The comparison for *name* is case insensitive.

On error, -1 is returned and errno is set to indicate the error.

See also decc$feature_get, decc$feature_get_value, decc$feature_get_name, decc$feature_set, decc$feature_set_value, decc$feature_show, and decc$feature_show_all.

Return Values

n

A nonnegative index that can be used to set or retrieve the specified values for the feature.

-1

Indicates an error; `errno` is set.

decc$feature_get_name

decc$feature_get_name — Returns a feature name.

Format

```
#include <unixlib.h>
char *decc$feature_get_name (int index);
```

Argument

index

An integer value from 0 to the highest allocated feature.

Description

The `decc$feature_get_name` function returns a pointer to a null-terminated string containing the name of the feature for the entry specified by *index*. The *index* value can be 0 to the highest allocated feature. If there is no feature corresponding to the *index* value, then the function returns a NULL pointer.

On error, NULL is returned and `errno` is set to indicate the error.

See also `decc$feature_get`, `decc$feature_get_index`, `decc$feature_get_value`, `decc$feature_set`, `decc$feature_set_value`, `decc$feature_show`, and `decc$feature_show_all`.

Return Values

x

Pointer to a null-terminated string containing the name of the feature for the entry specified by *index*.

NULL

Indicates an error; `errno` is set.

decc$feature_get_value

decc$feature_get_value — Returns a feature value specified by the *index* and *mode* arguments.

Format

```
#include <unixlib.h>
int decc$feature_get_value (int index, int mode);
```

Arguments

index

An integer value from 0 to the highest allocated feature.

mode

An integer indicating which feature value to return. The values for mode are:

_ _FEATURE_MODE_DEFVAL - Default value
_ _FEATURE_MODE_CURVAL - Current value
_ _FEATURE_MODE_MINVAL - Minimum value
_ _FEATURE_MODE_MAXVAL - Maximum value
_ _FEATURE_MODE_INIT_STATE - Initialization state

Description

The `decc$feature_get_value` function retrieves a value for the feature specified by *index*. The *mode* determines which value is returned.

The default value is what is used if not set by a logical name or overridden by a call to `decc $feature_set_value`.

If *mode* = 4, then the initialization state is returned. Values for the initialization state are:

0 not initialized
1 set by logical name
2 forced by `decc$feature_set_value`
-1 – initialized to default value

On error, -1 is returned and `errno` is set to indicate the error.

See also `decc$feature_get`, `decc$feature_get_index`, `decc$feature_get_name`, `decc$feature_set`, `decc$feature_set_value`, `decc$feature_show`, and `decc$feature_show_all`.

Return Values

n

An integer corresponding to the specified *index* and *mode* arguments.

-1

Indicates an error; `errno` is set.

decc$feature_set

decc$feature_set — Calls `decc$feature_set_value` with a character-string feature name, rather than an index.

Format

```
#include <unixlib.h>
```

```
int decc$feature_set (const char *name, int mode, int value);
```

Argument

name

Pointer to a character string passed as a name in the list of supported features.

mode

An integer indicating which feature value to return. The values for mode are:

_ _FEATURE_MODE_DEFVAL - Default value
_ _FEATURE_MODE_CURVAL - Current value
_ _FEATURE_MODE_MINVAL - Minimum value
_ _FEATURE_MODE_MAXVAL - Maximum value
_ _FEATURE_MODE_INIT_STATE - Initialization state

value

The feature value to be set.

Description

The `decc$feature_set` function allows you to call the `decc$feature_set_value` function with a character-string feature name, rather than an index into an internal C RTL table.

If successful, the function returns the previous value.

On error, -1 is returned and `errno` is set to indicate the error.

See also `decc$feature_set_value`, `decc$feature_get`, `decc$feature_get_index`, `decc$feature_get_name`, `decc$feature_get_value`. `decc$feature_show`, and `decc$feature_show_all`.

Return Values

n

The previous feature value.

-1

Indicates an error; `errno` is set.

decc$feature_set_value

decc$feature_set_value — Sets the default value or the current value for the feature specified by *index*.

Format

```
#include <unixlib.h>
int decc$feature_set_value (int index, int mode, int value);
```

Arguments

index

An integer value from 0 to the highest allocated feature.

mode

An integer indicating whether to set the default or current feature value. The values for mode are:

0 default value
1 current value

value

The feature value to be set.

Description

The `decc$feature_set_value` function sets the default value or the current value (as determined by the *mode* argument) for the feature specified by *index*.

If this function is successful, it returns the previous value.

On error, -1 is returned and `errno` is set to indicate the error.

See also `decc$feature_get`, `decc$feature_get_index`, `decc$feature_get_name`, `decc$feature_get_value`, `decc$feature_set`, `decc$feature_show`, and `decc$feature_show_all`.

Return Values

n

The previous feature value.

-1

Indicates an error; `errno` is set.

decc$feature_show

decc$feature_show — Displays all feature values for the specified feature name.

Format

```
#include <unixlib.h>
int decc$feature_show (const char *name);
```

Argument

name

Pointer to a character string passed as a name in the list of supported features.

Description

The `decc$feature_show` function displays to `stdout` all values for the specified feature *name*. For example:

```
--------- C RTL Feature Name ---------   Cur  Def  Min  Max  Ini
DECC$V62_RECORD_GENERATION                 0    0    0    1    -1
```

On error, -1 is returned and `errno` is set to indicate the error.

See also `decc$feature_get`, `decc$feature_get_index`, `decc$feature_get_name`, `decc$feature_get_value`, `decc$feature_set`, `decc$feature_set_value`, and `decc$feature_show_all`.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set.

decc$feature_show_all

decc$feature_show_all — Displays all feature values for all feature names.

Format

```
#include <unixlib.h>
int decc$feature_show_all (void);
```

Description

The `decc$feature_show_all` function displays to `stdout` all values for all feature names.

On error, -1 is returned and `errno` is set to indicate the error.

See also `decc$feature_get`, `decc$feature_get_index`, `decc$feature_get_name`, `decc$feature_get_value`, `decc$feature_set`, `decc$feature_set_value`, and `decc$feature_show`.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set.

decc$fix_time

decc$fix_time — Converts OpenVMS binary system times to UNIX binary times.

Format

```
#include <unixlib.h>
unsigned int decc$fix_time (void *vms_time);
```

Argument

vms_time

The address of a quadword containing an OpenVMS binary time:

```
unsigned int quadword[2];
unsigned int *vms_time = quadword;
```

Description

The `decc$fix_time` routine converts an OpenVMS binary system time (a 64-bit quadword containing the number of 100-nanosecond ticks since 00:00 November 17, 1858) to a UNIX binary time (a longword containing the number of seconds since 00:00 January 1, 1970). This routine is useful for converting binary times returned by OpenVMS system services and RMS services to the format used by some C RTL routines, such as `ctime` and `localtime`.

Return Values

x

A longword containing the number of seconds since 00:00 January 1, 1970.

(`unsigned int`)(-1)

Indicates an error. Be aware, that a return value of (`unsigned int`)(-1) can also represent a valid date of Sun Feb 7 06:28:15 2106.

Example

```
#include <unixlib.h>
#include <stdio.h>
#include <starlet.h>  /* OpenVMS specific SYS$ routines)  */

main()
{
unsigned int current_vms_time[2]; /*quadword for OpenVMS time*/
unsigned int number_of_seconds;   /* number of seconds     */

/* first get the current system time */
sys$gettim(&current_vms_time[0]);

/* fix the time */
number_of_seconds = decc$fix_time(&current_vms_time[0]);

printf("Number of seconds since 00:00 January 1, 1970 = %d",
```

```
        number_of_seconds);
}
```

This example shows how to use the `decc$fix_time` routine in VSI C. It also shows the use of the SYS$GETTIM system service.

decc$from_vms

decc$from_vms — Converts OpenVMS file specifications to UNIX style file specifications.

Format

```
#include <unixlib.h>
int decc$from_vms (const char *vms_filespec, int action_routine,
 int wild_flag);
```

Arguments

vms_filespec

The address of a null-terminated string containing a name in OpenVMS file specification format.

action_routine

The address of a routine that takes as its only argument a null-terminated string containing the translation of the given OpenVMS filename to a valid UNIX style filename.

If the *action_routine* returns a nonzero value (TRUE), file translation continues. If it returns a zero value (FALSE), no further file translation takes place.

wild_flag

Either 0 or 1, passed by value. If a 0 is specified, wildcards found in *vms_filespec* are not expanded. Otherwise, wildcards are expanded and each one is passed to *action_routine*. Only expanded filenames that correspond to existing UNIX style files are included.

Description

The `decc$from_vms` routine converts the given OpenVMS file specification into the equivalent UNIX style file specification. It allows you to specify OpenVMS wildcards, which are translated into a list of corresponding existing files in UNIX style file specification format.

Return Value

x

The number of filenames that result from the specified OpenVMS file specification.

Example

```
/* This example must be run as a foreign command        */
/* and be supplied with an OpenVMS file specification.  */

#include <unixlib.h>
```

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int number_found;           /* number of files found */
    int print_name();           /* name printer          */

    printf("Translating: %s\n", argv[1]);
    number_found = decc$from_vms(argv[1], print_name, 1);
    printf("\n%d files found", number_found);
}

/* print the name on each line */
print_name(char *name)
{
    printf("\n%s", name);
    /* will continue as long as success status is returned */
    return (1);
}
```

This example shows how to use the `decc$from_vms` routine in VSI C. It produces a simple form of the `ls` command that lists existing files that match an OpenVMS file specification supplied on the command line. The matching files are displayed in UNIX style file specification format.

decc$match_wild

decc$match_wild — Matches a string to a pattern.

Format

```
#include <unixlib.h>
int decc$match_wild (char *test_string, char *string_pattern);
```

Arguments

test_string

The address of a null-terminated string.

string_pattern

The address of a string containing the pattern to be matched. This pattern can contain wildcards (such as asterisks (*), question marks (?), and percent signs (%) as well as regular expressions (such as the range [a-z]).

Description

The `decc$match_wild` routine determines whether the specified test string is a member of the set of strings specified by the pattern.

Return Values

1 (TRUE)

The string matches the pattern.

0 (FALSE)

The string does not match the pattern.

Example

```
/* Define as a foreign command and then provide */
/* two arguments: test_string, string_pattern.  */

#include <unixlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (decc$match_wild(argv[1], argv[2]))
        printf("\n%s matches %s", argv[1], argv[2]);
    else
        printf("\n%s does not match %s", argv[1], argv[2]);
}
```

decc$record_read

decc$record_read — Reads a record from a file.

Format

```
#include <stdio.h>
int decc$record_read (FILE *fp, void *buffer, int nbytes);
```

Arguments

fp

A file pointer. The specified file pointer must refer to a file currently opened for reading.

buffer

The address of contiguous storage in which the input data is placed.

nbytes

The maximum number of bytes involved in the read operation.

Description

The `decc$record_read` function is specific to OpenVMS systems and should not be used when writing portable applications.

This function is equivalent to the `read` function, except that the first argument is a file pointer, not a file descriptor.

Return Values

x

The number of characters read.

-1

Indicates a read error, including physical input errors, illegal buffer addresses, protection violations, undefined file descriptors, and so forth.

decc$record_write

decc$record_write — Writes a record to a file.

Format

```
#include <stdio.h>
int decc$record_write (FILE *fp, void *buffer, int nbytes);
```

Arguments

fp

A file pointer. The specified file pointer must refer to a file currently opened for writing or updating.

buffer

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

nbytes

The maximum number of bytes involved in the write operation.

Description

The `decc$record_write` function is specific to OpenVMS systems and should not be used when writing portable applications.

This function is equivalent to the `write` function, except that the first argument is a file pointer, not a file descriptor.

Return Values

x

The number of bytes written.

-1

Indicates errors, including undefined file descriptors, illegal buffer addresses, and physical I/O errors.

decc$set_child_default_dir

decc$set_child_default_dir — Sets the default directory for a child process spawned by a function from the `exec` family of functions.

Format

```
#include <unixlib.h>
int decc$set_child_default_dir (const char *default_dir);
```

Argument

default_dir

The default directory specification for child processes, or NULL.

Description

By default, child processes created by one of the `exec` family of functions inherit the default (working) directory of their parent process.

The `decc$set_child_default_dir` function lets you set the default directory for a child process. After calling `decc$set_child_default_dir`, newly spawned child processes have their default directory set to *default_dir* as they begin execution. The *default_dir* argument must represent a valid directory specification, or results of the call are unpredictable (subsequent calls to the child process might fail without notification). Both OpenVMS and UNIX style file specifications are supported for this function call.

You can reestablish the default behavior by specifying *default_dir* as NULL. Subsequently, newly created child processes will inherit their parent's working directory.

Return Values

0

Successful completion. The new inherited default directory was established.

-1

Indicates failure. No new default directory was established for child processes. The function sets `errno` to one of the following values:

- ENOMEM– Insufficient memory
- ENAMETOOLONG– *default_dir* is too long to issue the required SET DEFAULT command.

decc$set_child_standard_streams

decc$set_child_standard_streams — For a child spawned by a function from the `exec` family of functions, associates specified file descriptors with a child's standard streams: `stdin`, `stdout`, and `stderr`.

Format

```
#include <unixlib.h>
int decc$set_child_standard_streams (int fd1, int fd2, int fd3);
```

Arguments

fd1

The file associated with this file descriptor in the parent process is associated with file descriptor number 0 (`stdin`) in the child process. If -1 is specified, the file associated with the parent's file descriptor number 0 is used (the default).

fd2

The file associated with this file descriptor in the parent process is associated with file descriptor number 1 (`stdout`) in the child process. If -1 is specified, the file associated with the parent's file descriptor number 1 is used (the default).

fd3

The file associated with this file descriptor in the parent process is associated with file descriptor number 2 (`stderr`) in the child process. If -1 is specified, the file associated with the parent's file descriptor number 2 is used (the default).

Description

The `decc$set_child_standard_streams` function allows mapping of specified file descriptors to the child's `stdin/stdout/stderr` streams, thereby compensating, to a certain degree, the lack of a real `fork` function on OpenVMS systems.

On UNIX systems, the code between `fork` and `exec` is executed in the context of the child process:

```
parent:
  create pipes p1, p2 and p3
  fork
child:

  map stdin to p1  like dup2(p1, stdin);
  map stdout to p2 like dup2(p2, stdout);
  map stderr to p3 like dup2(p3, stderr);

  exec (child reads from stdin and writes to stdout and stderr)
  exit
parent:
  communicates with the child using pipes
```

On OpenVMS systems, the same task could be achieved as follows:

```
parent:
  create pipes p1, p2 and p3
  decc$set_child_standard_streams(p1, p2, p3);
  vfork
  exec (child reads from stdin and writes to stdout and stderr)
parent:
  communicates with the child using pipes
```

Once established through the call to `decc$set_child_standard_streams`, the mapping of the child's standard streams remains in effect until explicitly disabled by one of the following calls:

```
decc$set_child_standard_streams(-1, -1, -1);
```

Or:

```
decc$set_child_standard_streams(0, 1, 2);
```

Usually, the child process inherits all its parent's open file descriptors. However, if file descriptor number *n* was specified in the call to `decc$set_child_standard_streams`, it is not inherited by the child process as file descriptor number *n*; instead, it becomes one of the child's standard streams.

Notes

- Standard streams can be redirected only to pipes.
- If the parent process redefines the DCL DEFINE command, this redefinition is not in effect in a subprocess with user-defined channels. The subprocess always sees the standard DCL DEFINE command.
- It is the responsibility of the parent process to consume all the output written by the child process to `stdout` and `stderr`. Depending on how the subprocess writes to `stdout` and `stderr`—in wait or nowait mode—the subprocess might be placed in LEF state waiting for the reader. For example, DCL writes to SYS$OUTPUT and SYS$ERROR in a wait mode, so a child process executing a DCL command procedure will wait until all the output is read by the parent process.

  Recommendation: Read the pipes associated with the child process' `stdout` and `stderr` in a loop until an EOF message is received, or declare write attention ASTs on these mailboxes.
- The amount of data written to SYS$OUTPUT depends on the verification status of the process (SET VERIFY/NOVERIFY command); the subprocess inherits the verification status of the parent process. It is the caller's responsibility to set the verification status of the parent process to match the expected amount of data written to SYS$OUTPUT by the subprocess.
- Some applications, like DTM, define SYS$ERROR as SYS$OUTPUT. If `stderr` is not redefined by the caller, it is set in the subprocess as the parent's SYS$ERROR, which in this case translates to the parent's SYS$OUTPUT.

  If the caller redefines `stdout` to a pipe and does not redefine `stderr`, output sent to `stderr` goes to the pipe associated with `stdout`, and the amount of data written to this mailbox may be more than expected. Although redefinition of any subset of standard channels is supported, it is always safe to explicitly redefine all of them (or at least `stdout` and `stderr`) to avoid this situation.
- For a child process executing a DCL command procedure, SYS$COMMAND is set to the pipe specified for the child's `stdin` so that the parent process can feed the child requesting data from SYS$COMMAND through the pipe. For DCL command procedures, it is impossible to pass data from the parent to the child by means of the child's SYS$INPUT because for a command procedure, DCL defines SYS$INPUT as the command file itself.

Return Values

x

The number of file descriptors set for the child. This number does not include file descriptors specified as -1 in the call.

-1

Indicates that an invalid file descriptor was specified; `errno` is set to EBADF.

Example

```
parent.c
========

#include <stdio.h>
#include <string.h>
#include <unistd.h>

int decc$set_child_standard_streams(int, int, int);

main()
{
    int fdin[2], fdout[2], fderr[2];
    char msg[] = "parent writing to child's stdin";
    char buf[80];
    int nbytes;

    pipe(fdin);
    pipe(fdout);
    pipe(fderr);

    if ( vfork() == 0 ) {
        decc$set_child_standard_streams(fdin[0], fdout[1], fderr[1]);
      execl( "child", "child" );
    }
    else {
        write(fdin[1], msg, sizeof(msg));
        nbytes = read(fdout[0], buf, sizeof(buf));
        buf[nbytes] = '\0';
        puts(buf);
        nbytes = read(fderr[0], buf, sizeof(buf));
        buf[nbytes] = '\0';
        puts(buf);
    }
}

child.c
=======

#include <stdio.h>
#include <unistd.h>

main()
{
    char msg[] = "child writing to stderr";
    char buf[80];
    int nbytes;

    nbytes = read(0, buf, sizeof(buf));
    write(1, buf, nbytes);
    write(2, msg, sizeof(msg));
}
```

```
child.com
=========

$ read sys$command s
$ write sys$output s
$ write sys$error "child writing to stderr"
```

This example program returns the following for both `child.c` and `child.com`:

```
$ run parent
parent writing to child's stdin
child writing to stderr
```

Note that in order to activate `child.com`, you must explicitly specify `execl("child.com", ...)` in the `parent.c` program.

decc$set_reentrancy

decc$set_reentrancy — Controls the type of reentrancy that reentrant C RTL routines will exhibit.

Format

```
#include <reentrancy.h>
int decc$set_reentrancy (int type);
```

Argument

type

The type of reentrancy desired. Use one of the following values:

- C$C_MULTITHREAD — Designed to be used in conjunction with the DECthreads product. It performs DECthreads locking and never disables ASTs. DECthreads must be available on your system to use this form of reentrancy.
- C$C_AST — Uses the _ _TESTBITSSI (Integrity servers, Alpha) built-in function to perform simple locking around critical sections of RTL code, and it may additionally disable asynchronous system traps (ASTs) in locked regions of code. This type of locking should be used when AST code contains calls to C RTL I/O routines, or when the user application disables ASTs.
- C$C_TOLERANT — Uses the _ _TESTBITSSI (Integrity servers, Alpha) built-in function to perform simple locking around critical sections of RTL code, but ASTs are not disabled. This type of locking should be used when ASTs are used and must be delivered immediately. TOLERANT is the default reentrancy type.
- C$C_NONE — Gives optimal performance in the C RTL, but does absolutely no locking around critical sections of RTL code. It should only be used in a single-threaded environment when there is no chance that the thread of execution will be interrupted by an AST that would call the C RTL.

The reentrancy type can be raised but never lowered. The ordering of reentrancy types from low to high is CC_NONE, CC_TOLERANT, C$C_AST and C$C_ MULTITHREAD. For example, once an application is set to multithread, a call to set the reentrancy to AST is ignored. A call to `decc $set_reentrancy` that attempts to lower the reentrancy type returns a value of -1.

Description

Use the `decc$set_reentrancy` function to change the type of reentrancy exhibited by reentrant routines.

`decc$set_reentrancy` must be called exclusively at the non-AST level.

In an application using DECthreads, DECthreads automatically sets the reentrancy to multithread.

Return Value

type

The type of reentrancy used before this call.

-1

The reentrancy was set to a lower type.

decc$to_vms

decc$to_vms — Converts UNIX style file specifications to OpenVMS file specifications.

Format

```
#include <unixlib.h>
int decc$to_vms (const char *unix_style_filespec, int (*action_routine)
(char *OpenVMS_style_filespec, int type_of_file), int allow_wild,
 int no_directory);
```

Arguments

unix_style_filespec

The address of a null-terminated string containing a name in UNIX style file specification format.

action_routine

The address of a routine called by `decc$to_vms` that accepts the following arguments:

- A pointer to a null-terminated string that is the result of the translation to OpenVMS format.
- An integer that has one of the following values:

Value	Translation
0 (DECC$K_FOREIGN)	A file on a remote system that is not running the OpenVMS or VAXELN operating system.
1 (DECC$K_FILE)	The translation is a file.
2 (DECC$K_DIRECTORY)	The OpenVMS translation of the UNIX style filename is a directory.

These values can be defined symbolically with the symbols DECC$K_FOREIGN, DECC$K_FILE, and DECC$K_DIRECTORY. See the example for more information.

If *action_routine* returns a nonzero value (TRUE), file translation continues. If it returns a 0 value (FALSE), no further file translation takes place.

allow_wild

Either 0 or 1, passed by value. If a 0 is specified, wildcards found in *unix_style_filespec* are not expanded. Otherwise, wildcards are expanded and each one is passed to *action_routine*. Only expanded filenames that correspond to existing OpenVMS files are included.

no_directory

An integer that has one of the following values:

Value	Translation
0	Directory allowed.
1	Prevent expansion of the string as a directory name.
2	Forced to be a directory name.

Description

The `decc$to_vms` function converts the given UNIX style file specification into the equivalent OpenVMS file specification (in all uppercase letters). It allows you to specify UNIX style wildcards, which are translated into a list of corresponding OpenVMS files.

See Section 1.5 for descriptions of the following feature logicals that can affect the behavior of `decc $to_vms`:

DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION
DECC$NO_ROOTED_SEARCH_LISTS

Return Value

x

The number of filenames that result from the specified UNIX style file specification.

Example

```
/* Translate "UNIX" wildcard file names to OpenVMS names.*/
/* Define as a foreign command and provide the name as   */
/* an argument.                                          */

#include <unixlib.h>
#include <stdio.h>
int print_name(char *, int);
int main(int argc, char *argv[])
{
    int number_found;           /* number of files found */

    printf("Translating: %s\n", argv[1]);

    number_found = decc$to_vms(argv[1], print_name, 1, 0);
    printf("%d files found\n", number_found);
}
```

```
/* action routine that prints name and type on each line */

int print_name(char *name, int type)
{
    if (type == DECC$K_DIRECTORY)
        printf("directory: %s\n", name);
    else if (type == DECC$K_FOREIGN)
        printf("remote non-VMS: %s\n", name);
    else
        printf("file:        %s\n", name);

/* Translation continues as long as success status is returned */
    return (1);
}
```

This example shows how to use the `decc$to_vms` routine in VSI C. It takes a UNIX style file specification argument and displays, in OpenVMS file specification format, the name of each existing file that matches it.

decc$translate_vms

decc$translate_vms — Translates OpenVMS file specifications to UNIX style file specifications.

Format

```
#include <unixlib.h>
char *decc$translate_vms (const char *vms_filespec);
```

Argument

vms_filespec

The address of a null-terminated string containing a name in OpenVMS file specification format.

Description

The `decc$translate_vms` function translates the given OpenVMS file specification into the equivalent UNIX style file specification, whether or not the file exists. The translated name string is stored in a thread-specific memory, which is overwritten by each call to `decc$translate_vms` from the same thread.

This function differs from the `decc$from_vms` function, which does the conversion for existing files only.

Return Values

x

The address of a null-terminated string containing a name in UNIX style file specification format.

0

Indicates that the filename is null or syntactically incorrect.

-1

Indicates that the file specification contains an ellipsis (for example, [. . .]a.dat), but is otherwise correct. You cannot translate the OpenVMS ellipsis syntax into a valid UNIX style file specification.

Example

```
/* Demonstrate translation of a "UNIX" name to OpenVMS  */
/* form, define a foreign command, and pass the name as */
/* the argument.                                        */

#include <unixlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    char *ptr;                      /* translation result */

    ptr = decc$translate_vms( argv[1] );

    if ((int) ptr == 0 || (int) ptr == -1)

        printf( "could not translate %s\n", argv[1]);
    else
        printf( "%s is translated to %s\n", argv[1], ptr );
}
```

decc$validate_wchar

decc$validate_wchar — Confirms that its argument is a valid wide character in the current program's locale.

Format

```
#include <unistd.h>
int decc$validate_wchar (wchar_t wc);
```

Argument

wc

Wide character to be validated.

Description

The `decc$validate_wchar` function provides a convenient way to verify whether a specified argument of `wchar_t` type is a valid wide character in the current program's locale.

One reason to call `decc$validate_wchar` is that the `isw*` wide-character classification functions and macros do not validate their argument before dereferencing the `classmask` array describing character properties. Passing an `isw*` function a value that exceeds the maximum wide-character value for the current program's locale can result in an attempt to access memory beyond the allocated `classmask` array.

A standard way to validate a wide character is to call the `wctomb` function, but this way is less convenient because it requires declaring a multibyte character array of sufficient size and passing it to `wctomb`.

Return Values

1

Indicates that the specified wide character is a valid wide character in the current program's locale.

0

Indicates that the specified wide character is not a valid wide character in the current program's locale. `errno` is not set.

decc$write_eof_to_mbx

decc$write_eof_to_mbx — Writes an end-of-file message to the mailbox.

Format

```
#include <unistd.h>
int decc$write_eof_to_mbx (int fd);
```

Argument

fd

File descriptor associated with the mailbox.

Description

The `decc$write_eof_to_mbx` function writes end-of-file message to the mailbox.

For a mailbox that is not a pipe, the `write` function called with an *nbytes* argument value of 0 sends an end-of-file message to the mailbox. For a pipe, however, the only way to write an end-of-file message to the mailbox is to close the pipe.

If the child's standard input is redirected to a pipe through a call to the `decc $set_child_standard_streams` function, the parent process can call `decc $write_eof_to_mbx` for this pipe to send an EOF message to the child. It has the same effect as if the child read the data from a terminal, and Ctrl/Z was pressed.

After a call to `decc$write_eof_to_mbx`, the pipe can be reused for communication with another child, for example. This is the purpose of `decc$write_eof_to_mbx`: to allow reuse of the pipe instead of having to close it just to send an end-of-file message.

Return Values

0

Indicates success.

-1

Indicates failure; `errno` and `vaxc$errno` are set according to the failure status returned by SYS$QIOW.

Example

```
/*      decc$write_eof_to_mbx_example.c         */

#include <errno.h>
#include <stdio.h>
#include <string.h>

#include <fcntl.h>
#include <unistd.h>
#include <unixio.h>

#include <descrip.h>
#include <ssdef.h>
#include <starlet.h>


int decc$write_eof_to_mbx( int );

main()
{
  int status, nbytes, failed = 0;
  int fd, fd2[2];
  short int channel;
  $DESCRIPTOR(mbxname_dsc, "TEST_MBX");
  char c;

  /* first try a mailbox created by SYS$CREMBX        */

status = sys$crembx(0, &channel, 0, 0, 0, 0, &mbxname_dsc, 0, 0);
if ( status != SS$_NORMAL ) {
     printf("sys$crembx failed: %s\n",strerror(EVMSERR, status));
     failed = 1;
  }

 if ( (fd = open(mbxname_dsc.dsc$a_pointer, O_RDWR, 0)) == -1) {
      perror("? open mailbox");
      failed = 1;
  }

  if ( decc$write_eof_to_mbx(fd) == -1 ) {
      perror("?  decc$write_eof_to_mbx to mailbox");
      failed = 1;
  }

  if ( (nbytes = read(fd, &c, 1)) != 0 || errno != 0 ) {
      perror("? read mailbox");
      printf("? nbytes = %d\n", nbytes);
      failed = 1;
  }

  if ( close(fd) == -1 ) {
```

```
      perror("? close mailbox");
      failed = 1;
  }

  /* Now do the same thing with a pipe                */

  errno = 0;           /* Clear errno for consistency */

  if ( pipe(fd2) == -1 ) {
      perror("? opening pipe");
      failed = 1;
  }

  if ( decc$write_eof_to_mbx(fd2[1]) == -1 ) {
      perror("? decc$write_eof_to_mbx to pipe");
      failed = 1;
  }

  if ( (nbytes = read(fd2[0], &c, 1)) != 0 || errno != 0 ) {
      perror("? read pipe");
      printf("? nbytes = %d\n", nbytes);
      failed = 1;
  }

  /* Close both file descriptors involved with the pipe    */

  if ( close(fd2[0]) == -1 ) {
      perror("close(fd2[0])");
      failed = 1;
  }

  if ( close(fd2[1]) == -1 ) {
      perror("close(fd2[1])");
      failed = 1;
  }

  if ( failed )
      puts("?Example program failed");
  else
      puts("Example ran to completion");
}
```

This example program produces the following result:

```
Example ran to completion
```

[w]delch

[w]delch — Delete the character on the specified window at the current position of the cursor. The `delch` function operates on the `stdscr` window.

Format

```
#include <curses.h>
int delch();
int wdelch (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

All of the characters to the right of the cursor on the same line are shifted to the left, and a blank character is appended to the end of the line.

Return Values

OK

Indicates success.

ERR

Indicates an error.

delete

delete — Deletes a file.

Format

```
#include <unixio.h>
int delete (const char *file_spec);
```

Argument

file_spec

A pointer to the string that is an OpenVMS or UNIX style file specification. The file specification can include a wildcard in its version number (but not in any other part of the file spec). So, for example, files of the form *filename*. *txt*;* can be deleted.

Description

If you specify a directory in the filename and it is a search list that contains an error, VSI C for OpenVMS systems interprets it as a file error.

When `delete` is used to delete a symbolic link, the link itself is deleted, not the file to which it refers.

The `remove` and `delete` functions are functionally equivalent in the C RTL.

See also `remove`.

Note

The `delete` routine is not available to C++ programmers because it conflicts with the C++ reserved word `delete`. C++ programmers should use the ANSI/ISO C standard function `remove` instead.

Return Values

0

Indicates success.

nonzero value

Indicates that the operation has failed.

[w]deleteln

[w]deleteln — Delete the line at the current position of the cursor. The `deleteln` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int deleteln();
int wdeleteln (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

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

Return Values

OK

Indicates success.

ERR

Indicates an error.

delwin

delwin — Deletes the specified window from memory.

Format

```
#include <curses.h>
int delwin (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

If the window being deleted contains a subwindow, the subwindow is invalidated. Delete subwindows before deleting their parent. The `delwin` function refreshes all windows covered by the deleted window.

Return Values

OK

Indicates success.

ERR

Indicates an error.

difftime

difftime — Computes the difference, in seconds, between the two times specified by the *time1* and *time2* arguments.

Format

```
#include <time.h>
double difftime (time_t time2, time_t time1);
```

Arguments

time2

A time value of type `time_t`.

time1

A time value of type `time_t`.

Description

The type `time_t` is defined in the `<time.h>` header file as follows:

```
typedef unsigned long int time_t
```

Return Value

n

time2 - *time1* in seconds expressed as a `double`.

dirname

dirname — Reports the parent directory name of a file pathname.

Format

```
#include <libgen.h>
char *dirname (char *path);
```

Function Variants

The `dirname` function has variants named `_dirname32` and `_dirname64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

path

The file pathname.

Description

The `dirname` function takes a pointer to a character string that contains a UNIX pathname and returns a pointer to a string that is a pathname of the parent directory of that file. Trailing slash (/) characters in the path are not counted as part of the path.

This function returns a pointer to the string "."(dot), when the *path* argument:

- Does not contain a slash (/).
- Is a NULL pointer.
- Points to an empty string.

The `dirname` function can modify the string pointed to by the *path* argument.

The `dirname` and `basename` functions together yield a complete pathname. The expression `dirname`(*path*) obtains the pathname of the directory where `basename` (*path*) is found.

See also `basename`.

Return Values

x

A pointer to a string that is the parent directory of the *path* argument.

"."

The *path* argument:

- Does not contain a slash (/).

- Is a NULL pointer.
- Points to an empty string.

Example

Using the `dirname` function, the following example reads a pathname, changes the current working directory to the parent directory, and opens a file.

```
char path [MAXPATHLEN], *pathcopy;
int fd;

fgets(path, MAXPATHLEN, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

div

div — 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

A numerator of type `int`.

denom

A denominator of type `int`.

Description

The type `div_t` is defined in the standard header file `<stdlib.h>` as follows:

```
typedef struct
        {
             int   quot, rem;
        } div_t;
```

dlclose

dlclose — Deallocates the address space for a shared library.

Format

```
#include <dlfcn.h>
```

```
void dlclose (void *handle);
```

Argument

handle

Pointer to the shared library.

Description

The `dlclose` function deallocates the address space allocated by the C RTL for the handle.

There is no way on OpenVMS systems to unload a shareable image dynamically loaded by the LIB$FIND_IMAGE_SYMBOL routine, which is the routine called by the `dlsym` function. In other words, there is no way on OpenVMS systems to release the address space occupied by the shareable image brought into memory by `dlsym`.

dlerror

dlerror — Returns a string describing the last error that occurred from a call to `dlopen`, `dlclose`, or `dlsym`.

Format

```
#include <dlfcn.h>
char *dlerror (void);
```

Return Value

x

A string describing the last error that occurred from a call to `dlopen`, `dlclose`, or `dlsym`.

dlopen

dlopen — Provides an interface to the dynamic library loader to allow shareable images to be loaded and called at run time.

Format

```
#include <dlfcn.h>
void *dlopen (char *pathname, int mode);
```

Arguments

pathname

The name of the shareable image. This name is saved for subsequent use by the `dlsym` function.

mode

This argument is ignored on OpenVMS systems.

Description

The `dlopen` function provides an interface to the dynamic library loader to allow shareable images to be loaded and called at run time.

This function does not load a shareable image but rather saves its *pathname* argument for subsequent use by the `dlsym` function. `dlsym` is the function that actually loads the shareable image through a call to LIB$FIND_IMAGE_SYMBOL.

The *pathname* argument of the `dlopen` function must be the name of the shareable image. This name is passed as-is by the `dlsym` function to the LIB$FIND_IMAGE_SYMBOL routine as the *filename* argument. No *image-name* argument is specified in the call to LIB$FIND_IMAGE_SYMBOL, so default file specification of SYS$SHARE:.EXE is applied to the image name.

The `dlopen` function returns a handle that is used by a `dlsym` or `dlclose` call. If an error occurs, a NULL pointer is returned.

Return Values

x

A handle to be used by a `dlsym` or `dlclose` call.

NULL

Indicates an error.

dlsym

dlsym — Returns the address of the symbol name found in a shareable image.

Format

```
#include <dlfcn.h>
void *dlsym (void *handle, char *name);
```

Arguments

handle

Pointer to the shareable image.

name

Pointer to the symbol name.

Description

The `dlsym` function returns the address of the symbol name found in the shareable image corresponding to *handle*. If the symbol is not found, a NULL pointer is returned.

As of OpenVMS Version 7.3-2, library symbols containing lowercase characters can be loaded using the `dlsym` function. More generally, the functions that dynamically load libraries (`dlopen`, `dlsym`, `dlclose`, `dlerror`) are enhanced to provide the following capabilities:

- Support for libraries with mixed-case symbol names
- Ability to pass a full file path to `dlopen`
- Validation of the specified library name

Return Values

x

Address of the symbol name found.

NULL

Indicates that the symbol was not found.

drand48

drand48 — Generates uniformly distributed pseudo-random-number sequences. Returns 48-bit, nonnegative, double-precision floating-point values.

Format

```
#include <stdlib.h>
double drand48 (void);
```

Description

The `drand48` function generates pseudorandom numbers using the linear congruential algorithm and 48-bit integer arithmetic.

It returns nonnegative, double-precision, floating-point values uniformly distributed over the range of y values such that $0.0 \leq y < 1.0$.

Before you call `drand48`, use either `srand48`, `seed48`, or `lcong48` to initialize the random-number generator. You must initialize prior to invoking the `drand48` function because it stores the last 48-bit X *i* generated into an internal buffer. (Although it is not recommended, constant default initializer values are supplied automatically if the `drand48`, `lrand48`, or `mrand48` functions are called without first calling an initialization function.)

The `drand48` function works by generating a sequence of 48-bit integer values, X *i*, according to the linear congruential formula:

$$Xn+1 = (aXn+c) \bmod m \qquad n \geq 0$$

The argument *m* equals 2^{48}, so 48-bit integer arithmetic is performed. Unless you invoke `lcong48`, the multiplier value *a* and the addend value *c* are:

$$a = 5DEECE66D_{16} = 273673163155_{8}$$
$$c = B_{16} = 13_{8}$$

The values returned by `drand48` are computed by first generating the next 48-bit X *i* in the sequence. Then the appropriate bits, according to the type of returned data item, are copied from the high-order (most significant) bits of X *i* and transformed into the returned value.

See also `srand48`, `seed48`, `lcong48`, `lrand48`, and `mrand48`.

Return Value

n

A nonnegative, double-precision, floating-point value.

dup, dup2

dup, dup2 — Allocate a new descriptor that refers to a file specified by a file descriptor returned by `open`, `creat`, or `pipe`.

Format

```
#include <unistd.h>
int dup (int file_desc1);
int dup2 (int file_desc1, int file_desc2);
```

Arguments

file_desc1

The file descriptor being duplicated.

file_desc2

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

Description

The `dup` function causes a previously unallocated descriptor to refer to its argument, while the `dup2` function causes its second argument to refer to the same file as its first argument.

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

Return Values

n

The new file descriptor.

-1

Indicates that an invalid argument was passed to the function.

[no]echo

[no]echo — Set the terminal so that characters may or may not be echoed on the terminal screen. This mode of single-character input is only supported with Curses.

Format

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

```
void echo (void);
void noecho (void);
```

Description

The `noecho` function may be helpful when accepting input from the terminal screen with `wgetch` and `wgetstr`; it prevents the input characters from being written onto the screen.

ecvt

ecvt — Converts its argument to a null-terminated string of ASCII digits and returns the address of the string. The string is stored in a thread-specific memory location created by the C RTL.

Format

```
#include <stdlib.h>
char *ecvt (double value, int ndigits, int *decpt, int *sign);
```

Arguments

value

An object of type `double` that is converted to a null-terminated string of ASCII digits.

ndigits

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

decpt

The position of the decimal point relative to the first character in the returned string. A negative `int` value means that the decimal point is *decpt* number of spaces to the left of the returned digits (the spaces being filled with zeros). A 0 value means that the decimal point is immediately to the left of the first digit in the returned string.

sign

An integer value that indicates whether the *value* argument is positive or negative. If *value* is negative, the function places a nonzero value at the address specified by *sign*. Otherwise, the function assigns 0 to the address specified by *sign*.

Description

The `ecvt` function converts *value* to a null-terminated string of length *ndigits*, and returns a pointer to it. The resulting low-order digit is rounded to the correct digit for outputting *ndigits* digits in C E-format. The *decpt* argument is assigned the position of the decimal point relative to the first character in the string.

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

The `ecvt`, `fcvt`, and `gcvt` functions represent the following special values specified in the IEEE Standard for floating-point arithmetic:

Value	Representation
Quiet NaN	NaNQ
Signalling NaN	NaNS
+Infinity	Infinity
-Infinity	-Infinity

The sign associated with each of these values is stored into the *sign* argument. In IEEE floating-point representation, a value of 0 (zero) can be positive or negative, as set by the *sign* argument.

See also `gcvt` and `fcvt`.

Return Value

x

The value of the converted string.

encrypt

encrypt — Encrypts a string using the key generated by the `setkey` function.

Format

```
#include <unistd.h>
#include <stdlib.h>
void encrypt (char *block[64], int edflag);
```

Argument

block

A character array of length 64 containing 0s and 1s.

edflag

An integer. If edflag is 0, the argument is encrypted; if nonzero, it is decrypted.

Description

The `encrypt` function encrypts a string using the key generated by the `setkey` function.

The first argument to `encrypt` is a character array of length 64 containing 0s and 1s. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by `setkey`.

The second argument, *edflag*, determines whether the first argument is encrypted or decrypted: if *edflag* is 0, the first argument array is encrypted; if nonzero, it is decrypted.

No value is returned.

See also `crypt` and `setkey`.

Return Value

pointer

Pointer to the encrypted password.

endgrent

endgrent — Closes the group database when processing is complete.

Format

```
#include <grp.h>
void endgrent (void);
```

Description

The `endgrent` function closes the group database.

This function is always successful. No value is returned, and `errno` is not set.

endpwent

endpwent — Closes the user database and any private stream used by `getpwent`.

Format

```
#include <pwd.h>
void endpwent (void);
```

Description

The `endpwent` function closes the user database and any private stream used by `getpwent`.

No value is returned. If an I/O error occurred, the function sets `errno` to EIO.

See also `getpwent`, `getpwuid`, `getpwnam`, and `setpwent`.

endwin

endwin — Clears the terminal screen and frees any virtual memory allocated to Curses data structures.

Format

```
#include <curses.h>
void endwin (void);
```

Description

A program that calls Curses functions must call the `endwin` function before exiting to restore the previous environment of the terminal screen.

erand48

erand48 — Generates uniformly distributed pseudorandom-number sequences. Returns 48-bit nonnegative, double-precision, floating-point values.

Format

```
#include <stdlib.h>
double erand48 (unsigned short int xsubi[3]);
```

Argument

xsubi

An array of three `short int`s, which form a 48-bit integer when concatenated together.

Description

The `erand48` function generates pseudorandom numbers using the linear congruential algorithm and 48-bit integer arithmetic.

It returns nonnegative, double-precision, floating-point values uniformly distributed over the range of y values, such that $0.0 \leq y < 1.0$.

The `erand48` function works by generating a sequence of 48-bit integer values, X *i*, according to the linear congruential formula:

$$Xn+1 = (aXn+c) \bmod m \qquad n >= 0$$

The argument *m* equals 2^{48}, so 48-bit integer arithmetic is performed. Unless you invoke the `lcong48` function, the multiplier value *a* and the addend value *c* are:

$$a = 5DEECE66D_{16} = 273673163155_{8}$$
$$c = B_{16} = 13_{8}$$

The `erand48` function requires that the calling program pass an array as the *xsubi* argument. For the first call, the array must be initialized to the value of the pseudorandom-number sequence. Unlike the `drand48` function, it is not necessary to call an initialization function prior to the first call.

By using different arguments, the `erand48` function allows separate modules of a large program to generate several independent sequences of pseudorandom numbers; for example, the sequence of numbers that one module generates does not depend upon how many times the function is called by other modules.

Return Value

n

A nonnegative, double-precision, floating-point value.

[w]erase

[w]erase — Erases the window by painting it with blanks. The `erase` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int erase();
int werase (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

Both the `erase` and `werase` functions leave the cursor at the current position on the terminal screen after completion; they do not return the cursor to the home coordinates of (0,0).

Return Values

OK

Indicates success.

ERR

Indicates an error.

erf

erf — Returns the error function of its argument.

Format

```
#include <math.h>
double erf (double x);
float erff (float x);
long double erfl (long double x);
double erfc (double x);
float erfcf (float x);
long double erfcl (long double x);
```

Argument

x

A radian expressed as a real number.

Description

The `erf` functions return the error function of x, where `erf`(*x*), `erff`(*x*), and `erfl`(*x*) equal 2/sqrt(π) times the area under the curve e**(-t**2) between 0 and *x*.

The `erfc` functions return (1.0 -`erf`(*x*)). The `erfc`function can result in an underflow as *x* gets large.

Return Values

x

The value of the error function (`erf`) or complementary error function (`erfc`).

NaN

x is NaN; `errno` is set to EDOM.

0

Underflow occurred; `errno` is set to ERANGE.

execl

execl — Passes the name of an image to be activated in a child process. This function is nonreentrant.

Format

```
#include <unistd.h>
int execl (const char *file_spec, const char *arg0, ..., (char *)0);
(ISO POSIX-1)
int execl (char *file_spec, ...); (Compatibility)
```

Arguments

file_spec

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

arg0, ...

A sequence of pointers to null-terminated character strings.

If the POSIX-1 format is used, at least one argument must be present and must point to a string that is the same as the new process filename (or its last component). (This pointer can also be the NULL pointer, but then `execle` would accomplish nothing.) The last pointer must be the NULL pointer. This is also the convention if the compatibility format is used.

Description

To understand how the `exec` functions operate, consider how the OpenVMS system calls any VSI C program, as shown in the following syntax:

```
int main (int argc, char *argv[], char *envp[]);
```

The identifier *argc* is the argument count; *argv* is an array of argument strings. The first member of the array (*argv*[0]) contains the name of the image. The arguments are placed in subsequent elements of the array. The last element of the array is always the NULL pointer.

An `exec` function calls a child process in the same way that the run-time system calls any other VSI C program. The `exec` functions pass the name of the image to be activated in the child; this value is placed in *argv*[0]. However, the functions differ in the way they pass arguments and environment information to the child:

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

If `vfork` was called before invoking an `exec` function, then when the `exec` function completes, control is returned to the parent process at the point of the `vfork` call. If `vfork` was not called, the `exec` function waits until the child has completed execution and then exits the parent process. See `vfork` and Chapter 5 for more information.

Return Value

-1

Indicates failure.

execle

execle — Passes the name of an image to be activated in a child process. This function is nonreentrant.

Format

```
#include <unistd.h>
int execle (char *file_spec, char *arg0, ..., (char *)0,
char *envp[ ]); (ISO POSIX-1)
int execle (char *file_spec, ...); (Compatibility)
```

Arguments

file_spec

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

arg0, ...

A sequence of pointers to null-terminated character strings.

If the POSIX-1 format is used, at least one argument must be present and must point to a string that is the same as the new process filename (or its last component). (This pointer can also be the NULL pointer, but then `execle` would accomplish nothing.) The last pointer must be the NULL pointer. This is also the convention if the compatibility format is used.

envp

An array of strings that specifies the program's environment. Each string in *envp* has the following form:

```
name = value
```

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

- HOME—Your login directory

- TERM—The type of terminal being used
- PATH—The default device and directory
- USER—The name of the user who initiated the process

The last element in *envp* must be the NULL pointer.

When the operating system executes the program, it places a copy of the current environment vector (*envp*) in the external variable environ.

Description

See `execl` for a description of how the `exec` functions operate.

Return Value

-1

Indicates failure.

execlp

execlp — Passes the name of an image to be activated in a child process. This function is nonreentrant.

Format

```
#include <unistd.h>
int execlp (const char *file_name, const char *arg0, ..., (char *)0);
(ISO POSIX-1)
int execlp (char *file_name, ...); (Compatibility)
```

Arguments

file_name

The filename of a new image to be activated in the child process. The device and directory specification for the file is obtained by searching the VAXC$PATH environment name.

argn

A sequence of pointers to null-terminated character strings. By convention, at least one argument must be present and must point to a string that is the same as the new process filename (or its last component).

...

A sequence of pointers to strings. At least one pointer must exist to terminate the list. This pointer must be the NULL pointer.

Description

See `execl` for a description of how the `exec` functions operate.

Return Value

-1

Indicates failure.

execv

execv — Passes the name of an image to be activated in a child process. This function is nonreentrant.

Format

```
#include <unistd.h>
int execv (char *file_spec, char *argv[]);
```

Arguments

file_spec

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

argv

An array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, *argv*[0] must point to a string that is the same as the new process filename (or its last component). *argv* is terminated by a NULL pointer.

Description

See `execl` for a description of how the `exec` functions operate.

Return Value

-1

Indicates failure.

execve

execve — Passes the name of an image to be activated in a child process. This function is nonreentrant.

Format

```
#include <unistd.h>
int execve (const char *file_spec, char *argv[], char *envp[]);
```

Arguments

file_spec

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

argv

An array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, *argv*[0] must point to a string that is the same as the new process filename (or its last component). *argv* is terminated by a NULL pointer.

envp

An array of strings that specifies the program's environment. Each string in *envp* has the following form:

```
name = value
```

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

- HOME—Your login directory
- TERM—The type of terminal being used
- PATH—The default device and directory
- USER—The name of the user who initiated the process

The last element in *envp* must be the NULL pointer.

When the operating system executes the program, it places a copy of the current environment vector (*envp*) in the external variable `environ`.

Description

See `execl` for a description of how the `exec` functions operate.

Return Value

-1

Indicates failure.

execvp

execvp — Passes the name of an image to be activated in a child process. This function is nonreentrant.

Format

```
#include <unistd.h>
int execvp (const char *file_name, char *argv[]);
```

Arguments

file_name

The filename of a new image to be activated in the child process. The device and directory specification for the file is obtained by searching the environment name VAXC$PATH.

argv

An array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, *argv*[0] must point to a string that is the same as the new process filename (or its last component). *argv* is terminated by a NULL pointer.

Description

See `execl` for a description of how the `exec` functions operate.

Return Value

-1

Indicates failure.

exit, _exit

exit, _exit — Terminate execution of the program from which they are called. These functions are nonreentrant.

Format

```
#include <stdlib.h>
void exit (int status);
#include <unistd.h>
void _exit (int status);
```

Argument

status

For non-POSIX behavior, a status value of EXIT_SUCCESS (1), EXIT_FAILURE (2), or a number from 3 to 255, as follows:

- A status value of 0, 1 or EXIT_SUCCESS is translated to the OpenVMS SS$_NORMAL status code to return the OpenVMS success value.
- A status value of 2 or EXIT_FAILURE is translated to an error-level exit status. The status value is passed to the parent process.
- Any other status value is left the same.

For POSIX behavior:

- A status value of 0 is translated to the OpenVMS SS$_NORMAL status code to return the OpenVMS success value.
- Any other status is returned to the parent process as an OpenVMS message symbol with facility set to C, severity set to success, and with the status in the message number field. For more information on the format of message symbols, see "message code" in the *VSI OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

To get POSIX behavior, include `<unistd.h>` and compile with the _POSIX_EXIT feature-test macro set (either with /DEFINE=_POSIX_EXIT, or with `#define _POSIX_EXIT` at the top of your file, before any file inclusions). This behavior is available only on OpenVMS Version 7.0 and higher systems.

Description

If the process was invoked by DCL, the status is interpreted by DCL, and a message is displayed.

If the process was a child process created using `vfork` or an `exec` function, then the child process exits and control returns to the parent. The two functions are identical; the `_exit` function is retained for reasons of compatibility with VAX C.

The `exit` and `_exit` functions make use of the $EXIT system service. If your process is being invoked by the RUN command using any of the hibernation and scheduled wakeup qualifiers, the process might not correctly return to hibernation state when an `exit` or `_exit` call is made.

The C compiler command-line qualifier /[NO]MAIN=POSIX_EXIT can be used to direct the compiler to call `__posix_exit` instead of `exit` when returning from `main`. The default is /NOMAIN.

Beginning with OpenVMS Version 8.3, C RTL contains a fix for the problem in which a call to _exit after a failed `execl` really exits but must not.

In the OpenVMS implementation of `vfork`, a child process is not actually started as it is started on most UNIX systems. However, the C RTL creates some internal data structures intended to mimic child-process functionality (called the "child context").

A bug occurred whereby after a vfork while in the child context, a call to an exec function justifiably fails, then calls `_exit`. On UNIX systems, after the failed exec call, the child process continues to execute. A subsequent call to `_exit` terminates the child. In the OpenVMS implementation, after the failed exec call, the child context terminates. A subsequent call to _exit terminates the parent. The C RTL fix is enabled by a feature logical switch, DECC$EXIT_AFTER_FAILED_EXEC. Enabling this feature logical allows the child context to continue execution.

With DECC$EXIT_AFTER_FAILED_EXEC disabled or not defined, the current behavior remains the default.

Note

EXIT_SUCCESS and EXIT_FAILURE are portable across any ANSI C compiler to indicate success or failure. On OpenVMS systems, they are mapped to OpenVMS condition codes with the severity set to success or failure, respectively. Values in the range of 3 to 255 can be used by a child process to communicate a small amount of data to the parent. The parent retrieves this data using the `wait`, `wait3`, `wait4`, or `waitpid` functions.

exp

exp — Returns the base e raised to the power of the argument.

Format

```
#include <math.h>
double exp (double x);
float expf (float x);
long double expl (long double x);
double expm1 (double x);
float expm1f (float x);
long double expm1l (long double x);
```

Argument

x

A real value.

Description

The `exp` functions compute the value of the exponential function, defined as e** *x*, where e is the constant used as a base for natural logarithms.

The `expm1` functions compute exp(*x*) - 1 accurately, even for tiny *x*.

If an overflow occurs, the `exp` functions return the largest possible floating-point value and set `errno` to ERANGE. The constant HUGE_VAL is defined in the `<math.h>` header file to be the largest possible floating-point value.

Return Values

x

The exponential value of the argument.

HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

0

Underflow occurred; `errno` is set to ERANGE.

NaN

x is NaN; `errno` is set to EDOM.

exp2

exp2 — Returns the value of 2 raised to the power of the argument.

Format

```
#include <math.h>
double exp2 (double x);
float exp2f (float x);
long double exp2l (long double x);
```

Argument

x

A real value.

Description

The `exp2` functions compute the base-2 exponential of *x*.

If an overflow occurs, the `exp` functions return the largest possible floating-point value and set `errno` to ERANGE. The constant HUGE_VAL is defined in the `<math.h>` header file to be the largest possible floating-point value.

Return Values

n

2** *x*.

HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

1

x is +0 or -0; `errno` is set to ERANGE.

0

x is -Inf or underflow occurred; `errno` is set to ERANGE.

x

x is +Inf; `errno` is set to ERANGE.

NaN

x is NaN; `errno` is set to EDOM.

fabs

fabs — Returns the absolute value of its argument.

Format

```
#include <math.h>
double fabs (double x);
float fabsf (float x);
long double fabsl (long double x);
```

Argument

x

A real value.

Return Value

x

The absolute value of the argument.

fchmod

fchmod — Changes file access permissions.

Format

```
#include <stat.h>
int fchmod (int fildes, mode_t mode);
```

Arguments

fildes

An open file descriptor.

mode

The bit pattern that determines the access permissions.

Description

The `fchmod` function is equivalent to the `chmod` function, except that the file whose permissions are changed is specified by a file descriptor (*fildes*) rather than a filename.

Return Values

0

Indicates that the mode is successfully changed.

-1

Indicates that the change attempt has failed.

fchown

fchown — Changes the owner and group of a file.

Format

```
#include <unistd.h>
int fchown (int fildes, uid_t owner, gid_t group);
```

Arguments

fildes

An open file descriptor.

owner

A user ID corresponding to the new owner of the file.

group

A group ID corresponding to the group of the file.

Description

The `fchown` function has the same effect as `chown` except that the file whose owner and group are to be changed is specified by the file descriptor *fildes*.

Return Values

0

Indicates success.

-1

Indicates failure. The function sets `errno` to one of the following values:

The `fchown` function *will* fail if:

- EBADF– The *fildes* argument is not an open file descriptor.
- EPERM– The effective user ID does not match the owner of the file, or the process does not have appropriate privilege.
- EROFS– The file referred to by *fildes* resides on a read-only file system.

The `fchown` function *may* fail if:

- EINVAL– The owner or group ID is not a value supported by the implementation.
- EIO– A physical I/O error has occurred.
- EINTR– The `fchown` function was interrupted by a signal that was intercepted.

fclose

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.

Format

```
#include <stdio.h>
int fclose (FILE *file_ptr);
```

Argument

file_ptr

A pointer to the file to be closed.

Description

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

The `fclose` function tries to write buffered data by using an implicit call to `fflush`.

If the write fails (because the disk is full or the user's quota is exceeded, for example), `fclose` continues executing. It closes the OpenVMS channel, deallocates any buffers, and releases the memory associated with the file descriptor (or FILE pointer). Any buffered data is lost, and the file descriptor (or FILE pointer) no longer refers to the file.

If your program needs to recover from errors when flushing buffered data, it should make an explicit call to `fsync` (or `fflush`) before calling `fclose`.

Return Values

0

Indicates success.

EOF

Indicates that the file control block is not associated with an open file.

fcntl

fcntl — Performs controlling operations on an open file.

Format

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl (int file_desc, int request [,int arg]);
int fcntl (int file_desc, int request [,struct flock *arg]);
```

Arguments

file_desc

An open file descriptor obtained from a successful `open`, `fcntl`, or `pipe` function.

request

The operation to be performed.

arg

A variable that depends on the value of the *request* argument.

For a *request* of F_DUPFD, F_SETFD, or F_SETFL, specify *arg* as an `int`.

For a *request* of F_GETFD and F_GETFL, do not specify *arg*.

For a *request* of F_GETLK, F_SETLK, or F_SETLKW specify *arg* as a pointer to a `flock` structure.

Description

The `fcntl` function performs controlling operations on the open file specified by the *file_desc* argument.

The values for the *request* argument are defined in the header file `<fcntl.h>`, and include the following:

`F_DUPFD`	Returns a new file descriptor that is the lowest numbered available (that is, not already open) file descriptor greater than or equal to the third argument (*arg*) taken as an integer of type `int`.
	
	The new file descriptor refers to the same file as the original file descriptor (*file_desc*). The FD_CLOEXEC flag associated with the new file descriptor is cleared to keep the file open across calls to one of the `exec` functions.
	
	The following two calls are equivalent:
	
	`fid = dup(file_desc);`
	`fid = fcntl(file_desc, F_DUPFD, 0);`
	
	Consider the following call:
	
	`fid = dup2(file_desc, arg);`
	
	It is similar (but not equivalent) to:
	
	`close(arg);`
	`fid = fcntl(file_desc, F_DUPFD, arg);`

`F_GETFD`	Gets the value of the close-on-exec flag associated with the file descriptor *file_desc*. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file. The *arg* argument should not be specified.
`F_SETFD`	Sets the close-on-exec flag associated with *file_desc* to the value of the third argument, taken as type `int`. If the third argument is 0, the file remains open across the `exec` functions, which means that a child process spawned by the `exec` function inherits this file descriptor from the parent. If the third argument is FD_CLOEXEC, the file is closed on successful execution of the next `exec` function, which means that the child process spawned by the `exec` function will not inherit this file descriptor from the parent.
`F_GETFL`	Gets the file status flags and file access modes, defined in `<fcntl.h>`, for the file description associated with *file_desc*. The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in `<fcntl.h>`. File status flags and file access modes are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions. The status flags and access modes supported are O_RDWR, O_RDONLY, O_WRONLY, O_APPEND, O_SYNC, O_DSYNC, and O_NONBLOCK.

F_SETFL	Sets the file status flags, defined in <fcntl.h>, for the file description associated with *file_desc* from the corresponding bits in the third argument, *arg*, taken as type int. Bits corresponding to the file access mode and the file creation flags, as defined in <fcntl.h>, that are set in *arg* are ignored. If any bits in *arg* other than those mentioned here are changed by the application, the result is unspecified. **Note:** The status flags supported are O_APPEND, O_SYNC, O_DSYNC, and O_NONBLOCK. Support for the file status flags is not standard-compliant. The X/Open standard states that "File status flags and file access modes are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions." However, because the append bit is stored in the FCB, all file descriptors using the same FCB are using the same append flag, so that setting this flag with fcntl(F_SETFL) will affect all files sharing the FCB; that is, all files duplicated from the same file descriptor.
Record Locking Requests	
F_GETLK	Gets the first lock that blocks the lock description pointed to by the *arg* parameter, taken as a pointer to type struct flock. The information retrieved overwrites the information passed to the fcntl function in the flock structure. If no lock is found that would prevent this lock from being created, then the structure is left unchanged except for the lock type, which is set to F_UNLCK.
F_SETLK	Sets or clears a file segment lock according to the lock description pointed to by *arg*, taken as a pointer to type struct flock. F_SETLK is used to establish shared locks (F_RDLCK), or exclusive locks (F_WRLCK), as well as remove either type of lock (F_UNLCK). If a shared (read) or exclusive (write) lock cannot beset, the fcntl function returns immediately with a value of -1. An unlock (F_UNLCK) request in which the *l_len* of the flock structure is nonzero and the offset of the last byte of the requested segment is the maximum value for an object of type off_t, when the process has an existing lock in which *l_len* is 0 and which includes the last byte of the requested segment, is treated as a request to unlock from the start of the requested segment with an *l_len* equal to 0. Otherwise, an unlock (F_UNLCK) request attempts to unlock only the requested file.
F_SETLKW	Same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process will wait until it is unblocked. If a signal is received while fcntl is waiting for a region, the function is interrupted, -1 is returned, and errno is set to EINTR.

File Locking

The C RTL supports byte-range file locking using the F_GETLK, F_SETLK, and F_SETLKW commands of the fcntl function, as defined in the X/Open specification. Byte-range file locking is supported across OpenVMS clusters. You can only use offsets that fit into 32-bit unsigned integers.

When a shared lock is set on a segment of a file, other processes on the cluster are able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor was not opened with read access.

An exclusive lock prevents any other process on the cluster from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file descriptor was not opened with write access.

The `flock` structure describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*) and process ID (*l_pid*) of the segment of the file to be affected.

The value of *l_whence* is set to SEEK_SET, SEEK_CUR or SEEK_END, to indicate that the relative offset *l_start* bytes is measured from the start of the file, from the current position, or from the end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. The *l_len* value may be negative (where the definition of `off_t` permits negative values of *l_len*). The *l_pid* field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, the value of *l_whence* becomes SEEK_SET.

If *l_len* is positive, the area affected starts at *l_start* and ends at *l_start* + *l_len* - 1. If *l_len* is negative, the area affected starts at *l_start*+ *l_len* and ends at *l_start* - 1. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. If *l_len* is set to 0 (zero), a lock may be set to always extend to the largest possible value of the file offset for that file. If such a lock also has *l_start* set to 0 (zero) and *l_whence* is set to SEEK_SET, the whole file is locked.

Changing or unlocking a portion from the middle of a larger locked segment leaves a smaller segment at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect.

All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process.

If the *request* argument is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling process to sleep to wait for that lock to become free would cause a deadlock, then the application will hang.

Return Values

n

Upon successful completion, the value returned depends on the value of the *request* argument as follows:

- `F_DUPFD`– Returns a new file descriptor.
- `F_GETFD`– Returns FD_CLOEXEC or 0.
- `F_SETFD`, `F_GETLK`, `F_SETLK`, `F_UNLCK` – Return a value other than -1.

-1

Indicates that an error occurred. The function sets `errno` to one of the following values:

- EACCES– The request argument is F_SETLK; the type of lock (*l_type*) is a shared (F_RDLCK) or exclusive (F_WRLCK) lock, and the segment of a file to be locked is already exclusive-locked by another process; or the type is an exclusive (F_WRLCK) lock and the some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

- EBADF – The *file_desc* argument is not a valid open file descriptor and the *arg* argument is negative or greater than or equal to the per-process limit.

  The *request* parameter is F_SETLK or F_SETLKW, the type of lock (*l_type*) is a shared lock (F_RDLCK), and *file_desc* is not a valid file descriptor open for reading.

  The type of lock (*l_type*) is an exclusive lock (F_WRLCK), and *file_desc* is not a valid file descriptor open for writing.

- EFAULT– The *arg* argument is an invalid address.

- EINVAL – The *request* argument is `F_DUPFD` and *arg* is negative or greater than or equal to OPEN_MAX.

  Either the OPEN_MAX value or the per-process soft descriptor limit is checked.

  An illegal value was provided for the *request* argument.

  The *request* argument is F_GETLK, F_SETLK, or F_SETLKW and the data pointed to by *arg* is invalid, or *file_desc* refers to a file that does not support locking.

- EMFILE– The *request* argument is `F_DUPFD` and too many or OPEN_MAX file descriptors are currently open in the calling process, or no file descriptors greater than or equal to *arg* are available.

  Either the OPEN_MAX value or the per-process soft descriptor limit is checked.

- EOVERFLOW– One of the values to be returned cannot be represented correctly.

  The request argument is F_GETLK, F_SETLK, or F_SETLKW and the smallest or, if *l_len* is nonzero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type `off_t`.

- EINTR– The *request* argument is F_SETLKW, and the function was interrupted by a signal.

- ENOLCK– The *request* argument is F_SETLK or F_SETLKW, and satisfying the lock or unlock request would exceed the configurable system limit of NLOCK_RECORD.

- ENOMEM– The system was unable to allocate memory for the requested file descriptor.

Example

The following code sample shows how to set the close-on-exec flag using the `fcntl` function:

```
#include <unistd.h>
#include <fcntl.h>
...
    int flags;


    flags = fcntl(fd, F_GETFD);
    if (flags == -1)
        /* Handle error */;
    flags |= FD_CLOEXEC;
    if (fcntl(fd, F_SETFD, flags) == -1)
        /* Handle error */;"
```

fcvt

fcvt — Converts its argument to a null-terminated string of ASCII digits and returns the address of the string. The string is stored in a thread-specific location created by the C RTL.

Format

```
#include <stdlib.h>
char *fcvt (double value, int ndigits, int *decpt, int *sign);
```

Arguments

value

An object of type `double` that is converted to a null-terminated string of ASCII digits.

ndigits

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

decpt

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

sign

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

Description

The `fcvt` function converts *value* to a null-terminated string and returns a pointer to it. The resulting low-order digit is rounded to the correct digit for outputting *ndigits* digits in C F-format. The *decpt* argument is assigned the position of the decimal point relative to the first character in the string.

In C F-format, *ndigits* is the number of digits desired after the decimal point. Very large numbers produce a very long string of digits before the decimal point, and *ndigit* of digits after the decimal point. For large numbers, it is preferable to use the `gcvt` or `ecvt` function so that E-format is used.

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

The `ecvt`, `fcvt`, and `gcvt` functions represent the following special values specified in the IEEE Standard for floating-point arithmetic:

Value	Representation
Quiet NaN	NaNQ
Signalling NaN	NaNS
+Infinity	Infinity

Value	Representation
-Infinity	-Infinity

The sign associated with each of these values is stored into the *sign* argument. In IEEE floating-point representation, a value of 0 (zero) can be positive or negative, as set by the *sign* argument.

See also `gcvt` and `ecvt`.

Return Value

x

A pointer to the converted string.

fdim

fdim — Determines the positive difference between its arguments.

Format

```
#include <math.h>
double fdim (double x, double y);
float fdimf (float x, float y);
long double fdiml (long double x, long double y);
```

Argument

x

A real value.

y

A real value.

Description

The `fdim` functions determine the positive difference between their arguments. If *x* is greater than *y*, *x* - *y* is returned. If *x* is less than or equal to *y*, +0 is returned.

Return Values

n

Upon success, the positive difference value.

HUGE_VAL

If *x* - *y* is positive and overflows; `errno` is set to ERANGE.

0

If *x* - *y* is positive and underflows; `errno` is set to ERANGE.

NaN

x or *y* is NaN; `errno` is set to EDOM.

fdopen

fdopen — Associates a file pointer with a file descriptor returned by an `open`, `creat`, `dup`, `dup2`, or `pipe` function.

Format

```
#include <stdio.h>
FILE *fdopen (int file_desc, char *a_mode);
```

Arguments

file_desc

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

a_mode

The access mode indicator. See the `fopen` function for a description. Note that the access mode specified must agree with the mode used to originally open the file. This includes binary/text access mode ("b" mode on `fdopen` and the "ctx=bin" option on `creat` or `open`).

Description

The `fdopen` function allows you to access a file, originally opened by one of the UNIX I/O functions, with Standard I/O functions. Ordinarily, a file can be accessed by either a file descriptor or by a file pointer, but not both, depending on the way you open it. For more information, see Chapters 1 and 2.

Return Values

pointer

Indicates that the operation has succeeded.

NULL

Indicates that an error has occurred.

feof

feof — Tests a file to see if the end-of-file has been reached.

Format

```
#include <stdio.h>
int feof (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Return Values

nonzero integer

Indicates that the end-of-file has been reached.

0

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

feof_unlocked

feof_unlocked — Same as `feof`, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

```
#include <stdio.h>
int feof_unlocked (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The reentrant version of the `feof` function is locked against multiple threads calling it simultaneously. This incurs overhead to ensure integrity of the stream. The unlocked version of this call, `feof_unlocked` can be used to avoid the overhead. The `feof_unlocked` function is functionally identical to the `feof` function, except that it is not required to be implemented in a thread-safe manner. The `feof_unlocked` function can be safely used only within a scope that is protected by the `flockfile` and `funlockfile` functions used as a pair. The caller must ensure that the stream is locked before `feof_unlocked` is used.

See also `flockfile`, `ftrylockfile`, and `funlockfile`.

Return Values

nonzero integer

Indicates end-of-file has been reached.

0

Indicates end-of-file has not been reached.

ferror

ferror — Returns a nonzero integer if an error occurred while reading or writing a file.

Format

```
#include <stdio.h>
int ferror (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

A call to `ferror` continues to return a nonzero integer until the file is closed or until `clearerr` is called.

Return Values

0

Indicates success.

nonzero integer

Indicates that an error has occurred.

ferror_unlocked

ferror_unlocked — Same as `ferror`, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

```
#include <stdio.h>
int ferror_unlocked (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The reentrant version of the `ferror` function is locked against multiple threads calling it simultaneously. This incurs overhead to ensure integrity of the stream. The unlocked version of this call, `ferror_unlocked` can be used to avoid the overhead. The `ferror_unlocked` function is functionally identical to the `ferror` function, except that it is not required to be implemented in a

thread-safe manner. The `ferror_unlocked` function can be safely used only within a scope that is protected by the `flockfile` and `funlockfile` functions used as a pair. The caller must ensure that the stream is locked before `ferror_unlocked` is used.

See also `flockfile`, `ftrylockfile`, and `funlockfile`.

Return Values

0

Indicates success.

nonzero integer

Indicates that an error has occurred.

fflush

fflush — Writes out any buffered information for the specified file.

Format

```
#include <stdio.h>
int fflush (FILE *file_ptr);
```

Argument

file_ptr

A file pointer. If this argument is a NULL pointer, all buffers associated with all currently open files are flushed.

Description

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

The `fflush` function flushes the C RTL buffers. However, RMS has its own buffers. The `fflush` function does not guarantee that the file will be written to disk. See the description of `fsync` for a way to flush buffers to disk.

If the file pointed to by *file_ptr* was opened in record mode and if there is unwritten data in the buffer, then `fflush` always generates a record.

Return Values

0

Indicates that the operation is successful.

EOF

Indicates that the buffered data cannot be written to the file, or that the file control block is not associated with an output file.

ffs

ffs — Finds the index of the first bit set in a string.

Format

```
#include <strings.h>
int ffs (int integer);
```

Argument

integer

The integer to be examined for the first bit set.

Description

The `ffs` function finds the first bit set (beginning with the least significant bit) and returns the index of that bit. Bits are numbered starting at 1 (the least significant bit).

Return Values

x

The index of the first bit set.

0

If *index* is 0.

fgetc

fgetc — Returns the next character from a specified file.

Format

```
#include <stdio.h>
int fgetc (FILE *file_ptr);
```

Argument

file_ptr

A pointer to the file to be accessed.

Description

The `fgetc` function returns the next character from the specified file.

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that uses a faster, inlined version of this function.

See also the `fgetc_unlocked` function and the `getc` macro.

Return Values

x

The returned character.

EOF

Indicates the end-of-file or an error.

fgetc_unlocked

fgetc_unlocked — Same as the `fgetc` function, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

```
#include <stdio.h>
int fgetc_unlocked (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The reentrant version of the `fgetc` function is locked against multiple threads calling it simultaneously. This incurs overhead to ensure integrity of the stream. The unlocked version of this call, `fgetc_unlocked` can be used to avoid the overhead. The `fgetc_unlocked` function is functionally identical to the `fgetc` function, except that `fgetc_unlocked` can be safely used only within a scope that is protected by the `flockfile` and `funlockfile` functions used as a pair. The caller must ensure that the stream is locked before `fgetc_unlocked` is used.

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that uses a faster, inlined version of this function.

See also `getc_unlocked`, `flockfile`, `ftrylockfile`, and `funlockfile`.

Return Values

n

The returned character.

EOF

Indicates the end-of-file or an error.

fgetname

fgetname — Returns the file specification associated with a file pointer.

Format

```
#include <stdio.h>
char *fgetname (FILE *file_ptr, char *buffer, ...);
```

Function Variants

The `fgetname` function has variants named `_fgetname32` and `_fgetname64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

file_ptr

A file pointer.

buffer

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

. . .

An optional additional argument that can be either 1 or 0.If you specify 1, the `fgetname` function returns the file specification in OpenVMS format. If you specify 0, `fgetname` returns the file specification in UNIX style format. If you do not specify this argument, `fgetname` returns the filename according to your current command language interpreter. For more information about UNIX style file specifications, see Section 1.3.3.

Description

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

Return Values

n

The address of the buffer.

0

Indicates an error.

Restriction

The `fgetname` function is specific to the C RTL and is not portable.

fgetpos

fgetpos — Stores the current file position for a given file.

Format

```
#include <stdio.h>
int fgetpos (FILE *stream, fpos_t *pos);
```

Arguments

stream

A file pointer.

pos

A pointer to an implementation-defined structure. The `fgetpos` function fills this structure with information that can be used on subsequent calls to `fsetpos`.

Description

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

Return Values

0

Indicates successful completion.

-1

Indicates that there are errors.

Example

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

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

    /* Open a file for writing.  */
```

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

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

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

    /* Write some data to the file. */

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

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

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

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

    /* Close the file. */

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

}
```

fgets

fgets — Reads a line from the specified file, up to one less than the specified maximum number of characters or up to and including the new-line character, whichever comes first. The function stores the string in *str*.

Format

```
#include <stdio.h>
char *fgets (char *str, int maxchar, FILE *file_ptr);
```

Function Variants

The `fgets` function has variants named `_fgets32` and `_fgets64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

str

A pointer to a character string that is large enough to hold the information fetched from the file.

maxchar

The maximum number of characters to fetch.

file_ptr

A file pointer.

Description

The `fgets` function terminates the line with a null character (\0). Unlike `gets`, `fgets` places the new-line character that terminates the input line into the user buffer if more than *maxchar* characters have not already been fetched.

When the file pointed to by *file_ptr* is opened in record mode, `fgets` treats the end of a record the same as a new-line character, so it reads up to and including a new-line character or to the end of the record.

Return Values

x

Pointer to *str*.

NULL

Indicates the end-of-file or an error. The contents of *str* are undefined if a read error occurs.

Example

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

main()
{
    FILE *fp;
    char c_ptr[130];

    /* Create a dummy data file  */

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

    fprintf(fp, "this is a test\n") ;
    fclose(fp) ;

    /* Open a file with some data -"this is a test"   */

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

```
    fgets(c_ptr, 130, fp);
    puts(c_ptr);        /* Display what fgets got.  */
    fclose(fp);

    delete("file.dat") ;
}
```

fgetwc

fgetwc — Reads the next character from a specified file, and converts it to a wide-character code.

Format

```
#include <wchar.h>
wint_t fgetwc (FILE *file_ptr);
```

Argument

file_ptr

A pointer to the file to be accessed.

Description

Upon successful completion, the `fgetwc` function returns the wide-character code read from the file pointed to by *file_ptr* and converted to type `wint_t`. If the file is at end-of-file, the end-of-file indicator is set, and WEOF is returned. If an I/O read error occurred, then the error indicator is set, and WEOF is returned.

Applications can use `ferror` or `feof` to distinguish between an error condition and an end-of-file condition.

Return Values

x

The wide-character code of the character read.

WEOF

Indicates the end-of-file or an error. If a read error occurs, the function sets `errno` to one of the following:

- EALREADY – An operation is already in progress on the same file.
- EBADF – The file descriptor is not valid.
- EILSEQ – Invalid character detected.

fgetws

fgetws — Reads a line of wide characters from a specified file.

Format

```
#include <wchar.h>
wchar_t *fgetws (wchar_t *wstr, int maxchar, FILE *file_ptr);
```

Function Variants

The `fgetws` function has variants named `_fgetws32` and `_fgetws64` for use with 32-bit and64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

wstr

A pointer to a wide-character string large enough to hold the information fetched from the file.

maxchar

The maximum number of wide characters to fetch.

file_ptr

A file pointer.

Description

The `fgetws` function reads wide characters from the specified file and stores them in the array pointed to by *wstr*. The function reads up to *maxchar*-1 characters or until the new-line character is read, converted, and transferred to *wstr*, or until an end-of-file condition is encountered. The function terminates the line with a null wide character. `fgetws` places the new-line that terminates the input line into the user buffer, unless *maxchar* characters have already been fetched.

Return Values

x

Pointer to *wstr*.

NULL

Indicates the end-of-file or an error. The contents of *wstr* are undefined if a read error occurs. If a read error occurs, the function sets `errno`. For a list of possible `errno` values, see `fgetwc`.

Example

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

main()
```

```
{
    wchar_t wstr[80],
           *ret;
    FILE *fp;

    /* Create a dummy data file  */

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

    fprintf(fp, "this is a test\n") ;
    fclose(fp) ;

   /* Open a test file containing : "this is a test" */

    if ((fp = fopen("file.dat", "r")) == (FILE *) NULL) {
        perror("File open error");
        exit(EXIT_FAILURE);
    }

    ret = fgetws(wstr, 80, fp);
    if (ret == (wchar_t *) NULL) {
        perror("fgetws failure");
        exit(EXIT_FAILURE);
    }

    fputws(wstr, stdout);
    fclose(fp);
    delete("file.dat");
}
```

fileno

fileno — Returns the file descriptor associated with the specified file pointer.

Format

```
#include <stdio.h>
int fileno (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

If you are using version 5.2 or lower of the C compiler, undefine the `fileno` macro:

```
#if defined(fileno)
#undef fileno
#endif
```

Return Values

x

Integer file descriptor.

-1

Indicates an error.

finite

finite — Returns the integer value 1 (TRUE) when its argument is a finite number, or 0 (FALSE) if not.

Format

```
#include <math.h>
int finite (double x);
int finitef (float x);
int double finitel (long double x);
```

Argument

x

A real value.

Description

The `finite` functions return 1 when -Infinity <*x* < +Infinity. They return 0 when | *x*| = Infinity, or *x* is a NaN.

flockfile

flockfile — Locks a `stdio` stream.

Format

```
#include <stdio.h>
void flockfile (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The `flockfile` function locks a `stdio` stream so that a thread can have exclusive use of that stream for multiple I/O operations. Use the `flockfile` function for a thread that wants to ensure that the

output of several `printf` functions, for example, is not garbled by another thread also trying to use `printf`.

File pointers passed are assumed to be valid; `flockfile` will perform locking even on invalid file pointers. Also, the `funlockfile` function will not fail if the calling thread does not own a lock on the file pointer passed.

Matching `flockfile` and `funlockfile` calls can be nested. If the stream has been locked recursively, it will remain locked until the last matching `funlockfile` is called.

All C RTL file-pointer I/O functions lock their file pointers as if calling `flockfile` and `funlockfile`.

See also `ftrylockfile` and `funlockfile`.

floor

floor — Returns the largest integer less than or equal to the argument.

Format

```
#include <math.h>
double floor (double x);
float floorf (float x);
long double floorl (long double x);
```

Argument

x

A real value.

Return Value

n

The largest integer less than or equal to the argument.

fma

fma — Computes (x * y) + z, rounded as one ternary operation.

Format

```
#include <math.h>
double fma (double x, double y, double z);
float fmaf (float x, float y, float z);
long double fmal (long double x, long double y, long double z);
```

Argument

x,y,z

Real values.

Description

The `fma` functions compute (*x* * *y*) + *z*, rounded as one ternary operation: the value is computed as if to infinite precision and rounded once to the result format, according to the rounding mode characterized by the value of FLT_ROUNDS.

Return Values

n

Upon success, (*x* * *y*) + *z*, rounded as one ternary operation.

NaN

x or *y* is NaN; `errno` is set to EDOM.

fmax

fmax — Returns the maximum numeric value of its arguments.

Format

```
#include <math.h>
double fmax (double x, double y);
float fmaxf (float x, float y);
long double fmaxl (long double x, long doubley);
```

Argument

x

A real value.

y

A real value.

Description

The `fmax` functions determine the maximum numeric value of their arguments. NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the numeric value is returned.

Return Values

n

Upon success, the maximum numeric value of the arguments. If just one argument is a NaN, the other argument is returned.

NaN

Both *x* and *y* are NaNs.

fmin

fmin — Returns the minimum numeric value of its arguments.

Format

```
#include <math.h>
double fmin (double x, double y);
float fminf (float x, float y);
long double fminl (long double x, long double y);
```

Argument

x

A real value.

y

A real value.

Description

The `fmin` functions determine the minimum numeric value of their arguments. NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the numeric value is returned.

Return Values

n

Upon success, the minimum numeric value of the arguments. If just one argument is a NaN, the other argument is returned.

NaN

Both *x* and *y* are NaNs.

fmod

fmod — Computes the floating-point remainder.

Format

```
#include <math.h>
double fmod (double x, double y);
```

```
float fmodf (float x, float y);
long double fmodl (long double x, long double y);
```

Arguments

x

A real value.

y

A real value.

Description

The `fmod` functions return the floating-point remainder of the first argument divided by the second. If the second argument is 0, the function returns 0.

Return Values

x

The value *f*, which has the same sign as the argument *x*, such that *x* == *i* * *y* + *f* for some integer *i*, where the magnitude of *f* is less than the magnitude of *y*.

0

Indicates that *y* is 0.

fopen

fopen — Opens a file by returning the address of a FILE structure.

Format

```
#include <stdio.h>
FILE *fopen (const char *file_spec, const char *a_mode); (ANSI C)
FILE *fopen (const char *file_spec, const char *a_mode, ...);
(VSI C Extension)
```

Arguments

file_spec

A character string containing a valid file specification.

a_mode

The access mode indicator. Use one of the following character strings: "r", "w", "a", "e" "r+ , "w+", "rb", "r+b ", "rb+", "wb", "w+b", "wb+", "ab", "a+b", "ab+", or "a+".

These access modes have the following effects:

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

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:

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

...

Optional file attribute arguments. The file attribute arguments are the same as those used in the `creat` function. For more information, see the `creat` function.

Description

If a version of the file exists, a new file created with `fopen` inherits certain attributes from the existing file unless those attributes are specified in the `fopen` call. The following attributes are inherited:

Record format
Maximum record size
Carriage control
File protection

If you specify a directory in the filename and it is a search list that contains an error, VSI C for OpenVMS systems interprets it as a file open error.

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

Return Values

x

File pointer.

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
- Failure to open the specified file

fp_class

fp_class — Determines the class of IEEE floating-point values.

Format

```
#include <math.h>
int fp_class (double x);
int fp_classf (float x);
int fp_classl (long double x);
```

Argument

x

An IEEE floating-point number.

Description

The `fp_class` functions determine the class of the specified IEEE floating-point number, returning a constant from the `<fp_class.h>` header file. They never cause an exception, even for signaling NaNs (Not-a-Number). These functions implement the recommended `class(x)` function in the appendix of the IEEE 754-1985 standard for binary floating-point arithmetic. The constants in `<fp_class.h>` refer to the following classes of values:

FP_SNAN	Signaling NaN (Not-a-Number)
FP_QNAN	Quiet NaN
FP_POS_INF	+Infinity
FP_NEG_INF	-Infinity
FP_POS_NORM	positive normalized
FP_NEG_NORM	negative normalized
FP_POS_DENORM	positive denormalized
FP_NEG_DENORM	negative denormalized
FP_POS_ZERO	+0.0 (positive zero)
FP_NEG_ZERO	-0.0 (negative zero)

Return Value

x

A constant from the `<fp_class.h>` header file.

fpathconf

fpathconf — Retrieves file implementation characteristics.

Format

```
#include <unistd.h>
long int fpathconf (int filedes, int name);
```

Arguments

filedes

An open file descriptor.

name

The configuration attribute to query. If this attribute is not applicable to the file specified by the *filesdes* argument, `fpathconf` returns an error.

Description

The `fpathconf` function allows an application to retrieve the characteristics of operations supported by the file system underlying the file named by the *filesdes* argument. Read, write, or execute permission of the named file is not required, but you must be able to search all directories in the path leading to the file.

Symbolic values for the *name* argument are defined in the `<unistd.h>` header file as follows:

_PC_LINK_MAX	The maximum number of links to the file. If the *filedes* argument refers to a directory, the value returned applies to the directory itself.
_PC_MAX_CANON	The maximum number of bytes in a canonical input line. This is applicable only to terminal devices.
_PC_MAX_INPUT	The number of types allowed in an input queue. This is applicable only to terminal devices.
_PC_NAME_MAX	Maximum number of bytes in a filename (not including a terminating null). The byte range value is between 13 and 255. This is applicable only to a directory file. The value returned applies to filenames within the directory.
_PC_PATH_MAX	Maximum number of bytes in a pathname (not including a terminating null). The value is never larger than 65,535. This is applicable only to a directory file. The value returned is the maximum length of a relative pathname when the specified directory is the working directory.
_PC_PIPE_BUF	Maximum number of bytes guaranteed to be written atomically. This is applicable only to a FIFO. The value returned applies to the referenced object. If the *path* argument refers to a directory, the value returned applies to any FIFO that exists or can be created within the directory.

_PC_CHOWN_RESTRICTED	The value returned applies to any files (other than directories) that exist or can be created within the directory. This is applicable only to a directory file.
_PC_NO_TRUNC	Returns 1 if supplying a component name longer than allowed by NAME_MAX causes an error. Returns 0 (zero) if long component names are truncated. This is applicable only to a directory file.
_PC_VDISABLE	This is always 0 (zero); no disabling character is defined. This is applicable only to a terminal device.

Return Values

x

The resultant value for the configuration attribute specified in the *name* argument.

-1

Indicates an error; `errno` is set to one of the following values:

- EINVAL– The *name* argument specifies an unknown or inapplicable characteristic.
- EBADF– The *filedes* argument is not a valid file descriptor.

fpclassify

fpclassify — Classifies the real floating-point type.

Format

```
#include <math.h>
int fpclassify (real-floating x);
```

Argument

x

Value to classify.

Description

The `fpclassify` macro classifies its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category.

First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.

Return Value

The `fpclassify` macro returns the value of the number classification macro appropriate to the value of its argument.

fprintf

fprintf — Performs formatted output to a specified file.

Format

```
#include <stdio.h>
int fprintf (FILE *file_ptr, const char *format_spec, ...);
```

Arguments

file_ptr

A pointer to the file to which the output is directed.

format_spec

A pointer to a character string that contains the format specification. For more information on format specifications and conversion characters, see Chapter 2.

. . .

Optional expressions whose resultant types correspond to conversion specifications given in the format specification.

If no conversion specifications are given, the output sources can be omitted. Otherwise, the function calls must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources.

Conversion specifications are matched to output sources in left-to-right order. Any excess output sources are ignored.

Example

An example of a conversion specification follows:

```
#include <stdio.h>

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

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

This example outputs the following to the `stdout` file:

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

For a complete description of the format specification and the output source, see Chapter 2.

Return Values

x

The number of bytes written, excluding the null terminator.

Negative value

Indicates an error. The function sets `errno` to one of the following:

- EILSEQ – Invalid character detected.
- EINVAL – Insufficient arguments.
- ENOMEM – Not enough memory available for conversion.
- ERANGE – Floating-point calculations overflow.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This might indicate that conversion to a numeric value failed because of overflow.

The function can also set `errno` to the following as a result of errors returned from the I/O subsystem:

- EBADF – The file descriptor is not valid.
- EIO – I/O error.
- ENOSPC – No free space on the device containing the file.
- ENXIO – Device does not exist.
- EPIPE – Broken pipe.
- ESPIPE – Illegal seek in a file opened for append.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This indicates that an I/O error occurred for which there is no equivalent C error code.

fputc

fputc — Writes a character to a specified file.

Format

```
#include <stdio.h>
int fputc (int character, FILE *file_ptr);
```

Arguments

character

An object of type `int`.

file_ptr

A file pointer.

Description

The `fputc` function writes a single character to the specified file and returns the character.

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that uses a faster, inlined version of this function.

See also the `fputc_unlocked` function and the `putc` macro.

Return Values

x

The character written to the file. Indicates success.

EOF

Indicates an output error.

fputc_unlocked

fputc_unlocked — Same as the `fputc` function, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

```
#include <stdio.h>
int fputc_unlocked (int character, FILE *file_ptr);
```

Arguments

character

The character to be written. An object of type `int`.

file_ptr

A file pointer.

Description

See the `putc_unlocked` macro.

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that uses a faster, inlined version of this function.

See also `flockfile`, `ftrylockfile`, and `funlockfile`.

Return Values

n

The returned character.

EOF

Indicates the end-of-file or an error.

fputs

fputs — Writes a character string to a file without copying the string's null terminator (\0).

Format

```
#include <stdio.h>
int fputs (const char *str, FILE *file_ptr);
```

Arguments

str

A pointer to a character string.

file_ptr

A file pointer.

Description

Unlike `puts`, the `fputs` function does not append a new-line character to the output string.

See also `puts`.

Return Values

Nonnegative value

Indicates success.

EOF

Indicates an error.

fputwc

fputwc — Converts a wide character to its corresponding multibyte value, and writes the result to a specified file.

Format

```
#include <wchar.h>
wint_t fputwc (wint_t wc, FILE *file_ptr);
```

Arguments

wc

An object of type `wint_t`.

file_ptr

A file pointer.

Description

The `fputwc` function writes a wide character to a file and returns the character.

See also `putwc`.

Return Values

x

The character written to the file. Indicates success.

WEOF

Indicates an output error. The function sets `errno` to the following:

- EILSEQ – Invalid wide-character code detected.

The function can also set `errno` to the following as a result of errors returned from the I/O subsystem:

- EBADF – The file descriptor is not valid.
- EIO – I/O error.
- ENOSPC – No free space on the device containing the file.
- ENXIO – Device does not exist.
- EPIPE – Broken pipe.
- ESPIPE – Illegal seek in a file opened for append.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This indicates that an I/O error occurred for which there is no equivalent C error code.

fputws

fputws — Writes a wide-character string to a file without copying the null-terminating character.

Format

```
#include <wchar.h>
int fputws (const wchar_t *wstr, FILE *file_ptr);
```

Arguments

wstr

A pointer to a wide-character string.

file_ptr

A file pointer.

Description

The `fputws` function converts the specified wide-character string to a multibyte character string and writes it to the specified file. The function does not append a terminating null byte corresponding to the null wide-character to the output string.

Return Values

Nonnegative value

Indicates success.

-1

Indicates an error. The function sets `errno`. For a list of the values, see `fputwc`.

fread

fread — 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

A pointer to the location, within memory, where you place the information being read. The type of the object pointed to is determined by the type of the item being read.

size_of_item

The size of the items being read, in bytes.

number_items

The number of items to be read.

file_ptr

A pointer that indicates the file from which the items are to be read.

Description

The type `size_t` is defined in the header file `<stdio.h>` as follows:

```
typedef unsigned int size_t
```

The reading begins at the current location in the file. The items read are placed in storage beginning at the location given by the first argument. You must also specify the size of an item, in bytes.

If the file pointed to by *file_ptr* was opened in record mode, `fread` will read *size_of_item* multiplied by *number_items* bytes from the file. That is, it does not necessarily read *number_items* records.

Return Values

n

The number of bytes read divided by *size_of_item*.

0

Indicates the end-of-file or an error.

free

free — Makes available for reallocation the area allocated by a previous `calloc`, `malloc`, or `realloc` call.

Format

```
#include <stdlib.h>
void free (void *ptr);
```

Argument

ptr

The address returned by a previous call to `malloc`, `calloc`, or `realloc`. If *ptr* is a NULL pointer, no action occurs.

Description

The ANSI C standard defines `free` as not returning a value; therefore, the function prototype for `free` is declared with a return type of `void`. However, since a `free` can fail, and since previous versions of the C RTL have declared `free` to return an `int`, the implementation of `free` does return 0 on success and -1 on failure.

freeifaddrs

freeifaddrs — Frees network interface addresses.

Format

```
#include <ifaddrs.h>
void freeifaddrs(struct ifaddrs *ifp);
```

Argument

ifp

A pointer to the linked list of `ifaddrs` structures to be freed.

Description

The `freeifaddrs` function frees the dynamically allocated data returned by the `getifaddrs` function. *ifp* is the address returned by a previous call to `getifaddrs`. If *ifp* is a NULL pointer, no action occurs.

freopen

freopen — 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

A pointer to a string that contains a valid OpenVMS or UNIX style file specification. After the function call, the given file pointer is associated with this file.

a_mode

The access mode indicator. See the `fopen` function for a description.

file_ptr

A file pointer.

...

Optional file attribute arguments. The file attribute arguments are the same as those used in the `creat` function.

Description

The `freopen` function is typically used to associate one of the predefined names `stdin`, `stdout`, or `stderr` with a file. For more information about these predefined names, see Chapter 2.

Return Values

file_ptr

The file pointer, if `freopen` is successful.

NULL

Indicates an error.

frexp

frexp — Calculates the fractional and exponent parts of a floating-point value.

Format

```
#include <math.h>
double frexp (double value, int *eptr);
float frexpf (float value, int *eptr);
long double frexpl (long double value, int *eptr);
```

Arguments

value

A floating-point number of type `double`, `float`, or `long double`.

eptr

A pointer to an `int` where `frexp` places the exponent.

Description

The `frexp` functions break the floating-point number (*value*) into a normalized fraction and an integral power of 2, as follows:

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

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

Example

```
#include <math.h>

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

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

}
```

In this example,

`frexp`

converts the value 16 to .5 * 2^{5}. The example produces the following output:

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

|*value*| = Infinity or NaN is an invalid argument.

Return Values

x

The fractional part of *value*.

0

Both parts of the result are 0.

NaN

If *value* is NaN, NaN is returned, `errno` is set to EDOM, and the value of * *eptr* is unspecified.

value

If |*value* | = Infinity, *value* is returned, `errno` is set to EDOM, and the value of * *eptr* is unspecified.

fscanf

fscanf — Performs formatted input from a specified file, interpreting it according to the format specification.

Format

```
#include <stdio.h>
int fscanf (FILE *file_ptr, const char *format_spec, ...);
```

Arguments

file_ptr

A pointer to the file that provides input text.

format_spec

A pointer to a character string that contains the format specification. For more information on conversion characters, see Chapter 2.

. . .

Optional expressions whose results correspond to conversion specifications given in the format specification.

If no conversion specifications are given, you can omit the input pointers. Otherwise, the function calls must have exactly as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input pointers.

Conversion specifications are matched to input sources in left-to-right order. Excess input pointers, if any, are ignored.

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);
}
```

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

```
4 17
```

The example conversion specification produces the following result:

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

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

Return Values

x

The number of successfully matched and assigned input items.

EOF

Indicates that the end-of-file was encountered or a read error occurred. If a read error occurs, the function sets `errno` to one of the following:

- EILSEQ – Invalid character detected.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This can indicate that conversion to a numeric value failed due to overflow.

The function can also set `errno` to the following as a result of errors returned from the I/O subsystem:

- EBADF – The file descriptor is not valid.
- EIO – I/O error.
- ENXIO – Device does not exist.
- EPIPE – Broken pipe.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This indicates that an I/O error occurred for which there is no equivalent C error code.

fseek

fseek — 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

A file pointer.

offset

The offset, specified in bytes.

direction

An integer indicating the position to which the *offset* is added to calculate the new position. The new position is the beginning of the file if *direction* is SEEK_SET, the current value of the file position indicator if *direction* is SEEK_CUR, or end-of-file if *direction* is SEEK_END.

Description

The `fseek` function can position a fixed-length record-access file with no carriage control or a stream-access file on any byte offset, but can position all other files only on record boundaries.

The available Standard I/O functions position a variable-length or VFC record file at its first byte, at the end-of-file, or on a record boundary. Therefore, the arguments given to `fseek` must specify any of the following:

- The beginning or end of the file
- A 0 offset from the current position (an arbitrary record boundary)
- The position returned by a previous, valid `ftell` call

See the `fgetpos` and `fsetpos` functions for a portable way to seek to arbitrary locations with these types of record files.

Caution

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

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

See also `open`, `creat`, `dup`, `dup2`, and `lseek`.

Return Values

0

Indicates successful seeks.

-1

Indicates improper seeks.

fseeko

fseeko — Positions the file to the specified byte offset in the file. Equivalent to `fseek`.

Format

```
#include <stdio.h>
int fseeko (FILE *file_ptr, off_t offset, int direction);
```

Arguments

file_ptr

A file pointer.

offset

The offset, specified in bytes. The `off_t` data type is either a 32-bit or 64-bit integer. The 64-bit interface allows for file sizes greater than 2 GB, and can be selected at compile time by defining the _LARGEFILE feature-test macro as follows:

```
CC/DEFINE=_LARGEFILE
```

direction

An integer indicating the position to which the *offset* is added to calculate the new position. The new position is the beginning of the file if *direction* is SEEK_SET, the current value of the file position indicator if *direction* is SEEK_CUR, or end-of-file if *direction* is SEEK_END.

Description

The `fseeko` function is identical to the `fseek` function, except that the *offset* argument is of type `off_t` instead of `long int`.

fsetpos

fsetpos — Sets the file position indicator for a given file.

Format

```
#include <stdio.h>
int fsetpos (FILE *stream, const fpos_t *pos);
```

Arguments

stream

A file pointer.

pos

A pointer to an implementation-defined structure. The `fgetpos` function fills this structure with information that can be used on subsequent calls to `fsetpos`.

Description

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

Return Values

0

Indicates success.

-1

Indicates an error.

fstat

fstat — Accesses information about the file specified by the file descriptor.

Format

```
#include <stat.h>
int fstat (int file_desc, struct stat *buffer);
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `fstat` function that is equivalent to the behavior before OpenVMS Version 7.0.

Arguments

file_desc

A file descriptor.

buffer

A pointer to a structure of type `stat_t`, which is defined in the `<stat.h>` header file. The argument receives information about that particular file. The members of the structure pointed to by *buffer* are:

Member	Type	Definition
st_dev	`dev_t`	Pointer to a physical device name
st_ino[3]	`ino_t`	Three words to receive the file ID
st_mode	`mode_t`	File “mode” (prot, dir, …)
st_nlink	`nlink_t`	For UNIX system compatibility only
st_uid	`uid_t`	Owner user ID
st_gid	`gid_t`	Group member: from st_uid
st_rdev	`dev_t`	UNIX system compatibility – always 0

Member	Type	Definition
st_size	off_t	File size, in bytes. For st_size to report a correct value, you need to flush both the CRTL and RMS buffers.
st_atime	time_t	File access time; always the same as st_mtime
st_mtime	time_t	Last modification time
st_ctime	time_t	File creation time
st_fab_rfm	char	Record format
st_fab_rat	char	Record attributes
st_fab_fsz	char	Fixed header size
st_fab_mrs	unsigned	Record size

The types dev_t, ino_t, off_t, mode_t, nlink_t, uid_t, gid_t, and time_t,are defined in the <stat.h> header file. However, when compiling for compatibility (/DEFINE=_DECC_V4_SOURCE), only dev_t, ino_t, and off_t are defined.

The off_t data type is either a 32-bit or 64-bit integer. The 64-bit interface allows for file sizes greater than 2 GB,and can be selected at compile time by defining the _LARGEFILE feature-test macro as follows:

```
CC/DEFINE=_LARGEFILE
```

As of OpenVMS Version 7.0, times are given in seconds since the Epoch (00:00:00 GMT, January 1, 1970).

The st_mode structure member is the status information mode and is defined in the <stat.h> header file. The st_mode bits follow:

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

Description

The fstat function does not work on remote network files.

Be aware that for the `stat_t` structure member `st_size` to report a correct value, you need to flush both the C RTL and RMS buffers.

Note

(Integrity servers, Alpha) On OpenVMS Alpha and Integrity server systems, the `stat`, `fstat`, `utime`, and `utimes` functions have been enhanced to take advantage of the new file-system support for POSIX compliant file timestamps.

This support is available only on ODS-5 devices on OpenVMS Alpha and Integrity servers systems beginning with a version of OpenVMS Alpha after Version 7.3.

Before this change, the `stat` and `fstat` functions were setting the values of the st_ctime, st_mtime, and st_atime fields based on the following file attributes:

st_ctime - ATR$C_CREDATE (file creation time)
st_mtime - ATR$C_REVDATE (file revision time)
st_atime - was always set to st_mtime because no support for file access time was available

Also, for the file-modification time, `utime` and `utimes` were modifying the ATR$C_REVDATE file attribute, and ignoring the file-access-time argument.

After the change, for a file on an ODS-5 device, the `stat` and `fstat` functions set the values of the st_ctime, st_mtime, and st_atime fields based on the following new file attributes:

st_ctime - ATR$C_ATTDATE (last attribute modification time)
st_mtime - ATR$C_MODDATE (last data modification time)
st_atime - ATR$C_ACCDATE (last access time)

If ATR$C_ACCDATE is zero, as on an ODS-2 device, the `stat` and `fstat` functions set st_atime to st_mtime.

For the file-modification time, the `utime` and `utimes` functions modify both the ATR$C_REVDATE and ATR$C_MODDATE file attributes. For the file-access time, these functions modify the ATR$C_ACCDATE file attribute. Setting the ATR$C_MODDATE and ATR$C_ACCDATE file attributes on an ODS-2 device has no effect.

For compatibility, the old behavior of `stat`, `fstat`, `utime`, and `utimes` remains the default, regardless of the kind of device.

The new behavior must be explicitly enabled at run time by defining the DECC$EFS_FILE_TIMESTAMPS logical name to "ENABLE" before invoking the application. Setting this logical does not affect the behavior of `stat`, `fstat`, `utime` and `utimes` for files on an ODS-2 device.

Return Values

0

Indicates successful completion.

-1

Indicates an error other than a protection violation.

-2

Indicates a protection violation.

fstatvfs

fstatvfs — Gets information about a device containing the specified file.

Format

```
#include <statvfs.h>
int fstatvfs (int filedes, struct statvfs *buffer);
```

Arguments

filedes

File descriptor obtained from a successful `open` or `fcntl` function call.

buffer

Pointer to a `statvfs` structure to hold the returned information.

Description

The `fstatvfs` function returns descriptive information about the device containing the specified file. Read, write, or execute permission of the specified file is not required. The returned information is in the format of a `statvfs` structure, which is defined in the `<statvfs.h>` header file and contains the following members:

`unsigned long f_bsize` - Preferred block size.
`unsigned long f_frsize` - Fundamental block size.
`fsblkcnt_t f_blocks` - Total number of blocks in units of `f_frsize`.
`fsblkcnt_t f_bfree` - Total number of free blocks. If `f_bfree` would assume a meaningless value due to the misreporting of free block count by $GETDVI for a DFS disk, then `f_bfree` is set to the maximum block count.
`fsblkcnt_t f_bavail` - Number of free blocks available. Set to the unused portion of the caller's disk quota.
`fsfilcnt_t f_files` - Total file (inode) count.
`fsfilcnt_t f_ffree` - Free file (inode) count. For OpenVMS systems, this value is calculated as freeblocks/clustersize.
`fsfilcnt_t f_favail` - Free file (inode) count nonprivileged. Set to `f_ffree`.
`unsigned long f_fsid` - File system identifier. This identifier is based on the allocation-class device name. This gives a unique value based on device, as long as the device is locally mounted.
`unsigned long f_flag` - Bit mask representing one or more of the following flags:

- `ST_RONLY` - The volume is read-only.
- `ST_NOSUID` - The volume has protected subsystems enabled.

`unsigned long f_namemax` - Maximum length of a filename.
`char f_basetype[64]` - Device-type name.
`char f_fstr[64]` - Logical volume name.

`char __reserved[64]` - Media type name.

Upon successful completion, `fstatvfs` returns 0 (zero). Otherwise, it returns -1 and sets `errno` to indicate the error.

See also `statvfs`.

Return Value

0

Successful completion.

-1

Indicates an error. `errno` is set to one of the following:

- EBADF - The file descriptor parameter contains an invalid value.
- EIO - An I/O error occurred while reading the device.
- EINTR - A signal was intercepted during execution of the function.
- EOVERFLOW - One of the values to be returned cannot be represented correctly in the structure pointed to by *buffer*.

fsync

fsync — Flushes data all the way to the disk.

Format

```
#include <unistd.h>
int fsync (int fd);
```

Argument

fd

A file descriptor corresponding to an open file.

Description

The `fsync` function behaves much like the `fflush` function. The primary difference between the two is that `fsync` flushes data all the way to the disk while `fflush` flushes data only as far as the underlying RMS buffers. Also, with `fflush`, you can flush all buffers at once; with `fsync` you cannot.

Return Values

0

Indicates successful completion.

-1

Indicates an error.

ftell

ftell — Returns the current byte offset to the specified stream file.

Format

```
#include <stdio.h>
long int ftell (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The `ftell` function measures the byte offset from the beginning of the file.

For variable-length files, VFC files, or any file with carriage-control attributes, if the file is opened in record mode, then `ftell` returns the starting position of the current record, not the current byte offset.

When using record files, the `ftell` function ignores any characters that have been pushed back using either `ungetc` or `ungetwc`. This behavior does not occur if stream files are being used.

For a portable way to measure the exact offset for any type of file, see the `fgetpos` function.

Return Values

n

The current offset.

EOF

Indicates an error.

ftello

ftellos — Returns the current byte offset to the specified stream file. This function is equivalent to `ftell`.

Format

```
#include <stdio.h>
off_t ftello (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The `ftello` function is identical to the `ftell` function, except that the return value is of type `off_t` instead of `long int`.

The `off_t` data type is either a 64-bit or 32-bit integer. The 64-bit interface allows for file sizes greater than 2 GB, and can be selected at compile time by defining the _LARGEFILE feature-test macro as follows:

```
CC/DEFINE=_LARGEFILE
```

ftime

ftime — Returns the elapsed time since 00:00:00, January 1, 1970, in the structure pointed at by *timeptr*.

Format

```
#include <timeb.h>
int ftime (struct timeb *timeptr);
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `ftime` function that is equivalent to the behavior before OpenVMS Version 7.0.

Argument

timeptr

A pointer to the structure `timeb_t`.

Description

The typedef `timeb_t` refers to the following structure defined in the `<timeb.h>` header file:

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

The member `time` gives the time in seconds.

The member `millitm` gives the fractional time in milliseconds.

After a call to `ftime`, the `timezone` and `dstflag` members of the `timeb` structure have the values of the global variables `timezone` and `dstflag`, respectively. See the description of the `tzset` function for `timezone` and `dstflag` global variables.

Return Values

0

Successful execution. The `timeb_t` structure is filled in.

-1

Indicates an error. Failure might indicate that the system's time-differential factor (that is, the difference between the system time and UTC time) is not set correctly.

If the value of the SYS$TIMEZONE_DIFFERENTIAL logical is wrong, the function fails with `errno` set to EINVAL.

ftok

ftok — Generates a standard interprocess communication key that is usable in subsequent calls to `semget`.

Format

```
#include <ipc.h>
key_t ftok (const char *path_name, int project_id);
```

Argument

path_name

the pathname of an existing file that is accessible to the process.

project_id

a value that uniquely identifies a project.

Description

The `ftok` function returns a key, based on the *path_name* and *project_id* parameters, that is usable in subsequent calls to the `semget` function. The `ftok` function returns the same key for all paths that name the same file, when called with the same *project_id* parameter. Different keys are returned for the same file if different *project_id* parameters are used, or if paths are used that name different files existing on the same file system at the same time. If a file named by *path_name* is removed and recreated with the same name, the `ftok` function may return a different key than the original one.

Only the low order 8 bits of *project_id* are significant. The behavior of `ftok` is unspecified if these bits are 0.

For maximum portability, *project_id* must be a single-byte character.

Upon successful completion, the `ftok` function returns a key. Otherwise, it returns the value `(key_t)-1` and sets `errno` to indicate the error.

Return Values

n

Upon successful completion, the `ftok` function returns a key.

(key_t) -1

Indicates an error. The function sets `errno` to:

- EACCESS– Search permission is denied for a component of the *path_name* parameter.

ftruncate

ftruncate — Truncates a file to a specified length.

Format

```
#include <unistd.h>
int ftruncate (int filedes, off_t length);
```

Arguments

filedes

The descriptor of a file that must be open for writing.

length

The new length of the file, in bytes. The `off_t` data type is either a 32-bit or 64-bit integer. The 64-bit interface allows for file sizes greater than 2 GB, and can be selected at compile time by defining the _LARGEFILE feature-test macro as follows:

```
CC/DEFINE=_LARGEFILE
```

Description

The `ftruncate` function truncates a file at the specified position. For record files, the position must be a record boundary. Also, the files must be local, regular files.

If the file was previously larger than *length*, extra data is lost. If the file was previously shorter than *length*, bytes between the old and new lengths are read as zeros.

Return Values

0

Indicates success.

-1

An error occurred; `errno` is set to indicate the error.

ftrylockfile

ftrylockfile — Acquires ownership of a `stdio (FILE*)` object.

Format

```
#include <stdio.h>
int ftrylockfile (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The `ftrylockfile` function is used by a thread to acquire ownership of a `stdio (FILE*)` object, if the object is available. The `ftrylockfile` function is a non-blocking version of `flockfile`.

The `ftrylockfile` function returns zero for success and nonzero to indicate that the lock cannot be acquired.

See also `flockfile` and `funlockfile`.

Return Values

0

Indicates success.

nonzero

Indicates the lock cannot be acquired.

ftw

ftw — Walks a file tree.

Format

```
#include <ftw.h>
int ftw (const char *path, int(*function)(const char *,
const struct stat *, int), int depth);
```

Arguments

path

The directory hierarchy to be searched.

function

The function to be invoked for each file in the directory hierarchy.

depth

The maximum number of directory streams or file descriptors, or both, available for use by `ftw`. This argument should be in the range of 1 to OPEN_MAX.

Description

The `ftw` function recursively searches the directory hierarchy that descends from the directory specified by the *path* argument. The *path* argument can be specified in OpenVMS style or UNIX style.

For each file in the hierarchy, `ftw` calls the function specified by the *function* argument, passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a `stat` structure containing information about the file, and an integer.

The integer identifies the file type. Possible values, defined in `<ftw.h>` are:

FTW_F	Regular file.
FTW_D	Directory.
FTW_DNR	Directory that cannot be read.
FTW_NS	A file on which `stat` could not successfully be executed.

If the integer is FTW_DNR, then the files and subdirectories contained in that directory are not processed.

If the integer is FTW_NS, then the `stat` structure contents are meaningless. For example, a file in a directory for which you have read permission but not execute (search) permission can cause the *function* argument to pass FTW_NS.

The `ftw` function finishes processing a directory before processing any of its files or subdirectories.

The `ftw` function continues the search until:

- The directory hierarchy specified by the *path* argument is completed.
- An invocation of the function specified by the *function* argument returns a nonzero value.
- An error (such as an I/O error) is detected within the `ftw` function.

Because the `ftw` function is recursive, it is possible for it to terminate with a memory fault because of stack overflow when applied to very deep file structures.

The `ftw` function uses the `malloc` function to allocate dynamic storage during its operation. If `ftw` is forcibly terminated, as with a call to `longjmp` from the function pointed to by the *function* argument, `ftw` has no chance to free that storage. It remains allocated.

A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the *function* argument return a nonzero value the next time it is called.

Notes

- The `ftw` function is reentrant; make sure that the function supplied as argument function is also reentrant.

- The C RTL supports a standard-compliant definition of the `stat` structure and associated definitions. To use them, compile your application with the _USE_STD_STAT feature-test macro defined. See the `<stat.h>` header file on your system for more information.
- The `ftw` function supports UNIX style path name specifications. For more information about UNIX style directory specifications, see Section 1.3.3.

See also `malloc`, `longjump`, and `stat`.

Return Values

0

Indicates success.

x

Indicates that the function specified by the *function* argument stops its search, and returns the value that was returned by the function.

-1

Indicates an error; `errno` is set to one of the following values:

- EACCES– Search permission is denied for any component of the *path* argument or read permission is denied for the *path* argument.
- ENAMETOOLONG– The length of the path string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while [_POSIX_NO_TRUNC] is in effect.
- ENOENT– The *path* argument points to the name of a file that does not exist or points to an empty string.
- ENOMEM– There is insufficient memory for this operation.

Also, if the function pointed to by the *function* argument encounters an error, `errno` can be set accordingly.

funlockfile

funlockfile — Unlocks a `stdio` stream.

Format

```
#include <stdio.h>
void funlockfile (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The `funlockfile` function unlocks a `stdio` stream, causing the thread that had been holding the lock to relinquish exclusive use of the stream.

File pointers passed are assumed to be valid; `flockfile` will perform locking even on invalid file pointers. Also, the `funlockfile` function will not fail if the calling thread does not own a lock on the file pointer passed.

Matching `flockfile` and `funlockfile` calls can be nested. If the stream has been locked recursively, it will remain locked until the last matching `funlockfile` is called.

All C RTL file-pointer I/O functions lock their file pointers as if calling `flockfile` and `funlockfile`.

See also `flockfile` and `ftrylockfile`.

fwait

fwait — Waits for I/O on a specific file to complete.

Format

```
#include <stdio.h>
int fwait (FILE *fp);
```

Argument

fp

A file pointer corresponding to an open file.

Description

The `fwait` function is used primarily to wait for completion of pending asynchronous I/O.

Return Values

0

Indicates successful completion.

-1

Indicates an error.

fwide

fwide — Determines and sets the orientation of a stream.

Format

```
#include <wchar.h>
```

```
int fwide (FILE *stream, int mode);
```

Arguments

stream

A file pointer.

mode

A value that specifies the desired orientation of the stream.

Description

The `fwide` function determines the orientation of the stream pointed to by *stream* and sets the orientation of a nonoriented stream according to the *mode* argument in the following way:

If the *mode* argument is:	**Then the fwide function:**
greater than zero	makes the stream wide-oriented.
less than zero	makes the stream byte-oriented.
zero	does not alter the orientation of the stream.

If the orientation of the stream has already been set, `fwide` does not alter it. Because no error status is defined for `fwide`, the calling application should check `errno` if `fwide` returns a 0.

Return Values

> 0

After the call, the stream is wide-oriented.

< 0

After the call, the stream is byte-oriented.

0

After the call, the stream has no orientation or a stream argument is invalid; the function sets `errno`.

fwprintf

fwprintf — Writes output to the stream under control of the wide-character format string.

Format

```
#include <wchar.h>
int fwprintf (FILE *stream, const wchar_t *format, ...);
```

Arguments

stream

A file pointer.

format

A pointer to a wide-character string containing the format specifications. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

. . .

Optional expressions whose resultant types correspond to conversion specifications given in the format specification.

If no conversion specifications are given, the output sources can be omitted. Otherwise, the function calls must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources.

Conversion specifications are matched to output sources in left-to-right order. Any excess output sources are ignored.

Description

The `fwprintf` function writes output to the stream pointed to by *stream* under control of the wide-character string pointed to by *format*, which specifies how to convert subsequent arguments to output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated, but are otherwise ignored. The `fwprintf` function returns when it encounters the end of the format string.

The *format* argument is composed of zero or more directives that include:

- Ordinary wide characters (not the percent sign (%))
- Conversion specifications

Return Values

n

The number of wide characters written.

Negative value

Indicates an error. The function sets `errno` to one of the following:

- EILSEQ – Invalid character detected.
- EINVAL – Insufficient arguments.
- ENOMEM – Not enough memory available for conversion.
- ERANGE – Floating-point calculations overflow.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This might indicate that conversion to a numeric value failed because of overflow.

The function can also set `errno` to the following as a result of errors returned from the I/O subsystem:

- EBADF – The file descriptor is not valid.
- EIO – I/O error.
- ENOSPC – No free space on the device containing the file.
- ENXIO – Device does not exist.
- EPIPE – Broken pipe.
- ESPIPE – Illegal seek in a file opened for append.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This indicates that an I/O error occurred for which there is no equivalent C error code.

Example

The following example shows how to print a date and time in the form "Sunday, July 3, 10:02", followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
#include <wchar.h>
/*...*/
wchar_t *weekday, *month; /* pointers to wide-character strings */
int day, hours, min;
fwprintf(stdout, L"%ls, %ls %d, %.2d:%.2d\n",
   weekday, month, day, hour, min);
fwprintf(stdout, L"pi = %.5f\n", 4 * atan(1.0));
```

fwrite

fwrite — Writes a specified number of items to the file.

Format

```
#include <stdio.h>
size_t fwrite (const void *ptr, size_t size_of_item, size_t number_items,
FILE *file_ptr);
```

Arguments

ptr

A pointer to the memory location from which information is being written. The type of the object pointed to is determined by the type of the item being written.

size_of_item

The size, in bytes, of the items being written.

number_items

The number of items to be written.

file_ptr

A file pointer that indicates the file to which the items are being written.

Description

The type `size_t` is defined in the header file `<stdio.h>` as follows:

```
typedef unsigned int size_t
```

The writing begins at the current location in the file. The items are written from storage beginning at the location given by the first argument. You must also specify the size of an item, in bytes.

If the file pointed to by *file_ptr* is a record file, the `fwrite` function outputs at least *number_items* records, each of length *size_of_item*.

Return Value

x

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

fwscanf

fwscanf — Reads input from the stream under control of the wide-character format string.

Format

```
#include <wchar.h>
int fwscanf (FILE *stream, const wchar_t *format, ...);
```

Arguments

stream

A file pointer.

format

A pointer to a wide-character string containing the format specification. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

. . .

Optional expressions whose results correspond to conversion specifications given in the format specification. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

If no conversion specifications are given, you can omit the input pointers. Otherwise, the function calls must have exactly as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input pointers.

Conversion specifications are matched to input sources in left-to-right order. Excess input pointers, if any, are ignored.

Description

The `fwscanf` function reads input from the stream pointed to by *stream* under the control of the wide-character string pointed to by *format*. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated, but otherwise ignored.

The format is composed of zero or more directives that include:

- One or more white-space wide characters.
- An ordinary wide character (neither a percent (%)) nor a white-space wide character).
- Conversion specifications.

Each conversion specification is introduced by the wide character %.

If the stream pointed to by the stream argument has no orientation, `fwscanf` makes the stream wide-oriented.

Return Values

n

The number of input items assigned, sometimes fewer than provided for, or even zero, in the event of an early matching failure.

EOF

Indicates an error; input failure occurs before any conversion.

gcvt

gcvt — Converts its argument to a null-terminated string of ASCII digits and returns the address of the string.

Format

```
#include <stdlib.h>
char *gcvt (double value, int ndigit, char *buffer);
```

Function Variants

The `gcvt` function has variants named `_gcvt32` and `_gcvt64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

value

An object of type `double` that is converted to a null-terminated string of ASCII digits.

ndigit

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

buffer

A storage location to hold the converted string.

Description

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

The `ecvt`, `fcvt`, and `gcvt` functions represent the following special values specified in the IEEE Standard for floating-point arithmetic:

Value	Representation
Quiet NaN	NaNQ
Signalling NaN	NaNS
+Infinity	Infinity
-Infinity	-Infinity

The sign associated with each of these values is stored into the *sign* argument. In IEEE floating-point representation, a value of 0 (zero) can be positive or negative, as set by the *sign* argument.

See also `fcvt` and `ecvt`.

Return Value

x

The address of the buffer.

getc

getc — Returns the next character from a specified file.

Format

```
#include <stdio.h>
int getc (FILE *file_ptr);
```

Argument

file_ptr

A pointer to the file to be accessed.

Description

The `getc` macro returns the next byte from the input stream specified by the *file_ptr* parameter and moves the file pointer, if defined, ahead one byte in the input stream.

Since `getc` is a macro, a file pointer argument with side effects (for example, `getc (*f++)`) might be evaluated incorrectly. In such a case, use the `fgetc` function instead. See the `fgetc` function.

See also `getc_unlocked`.

Return Values

n

The returned character.

EOF

Indicates the end-of-file or an error.

getc_unlocked

getc_unlocked — Same as `getc`, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

```
#include <stdio.h>
int getc_unlocked (FILE *file_ptr);
```

Argument

file_ptr

A file pointer.

Description

The reentrant version of the `getc` macro is locked against multiple threads calling it simultaneously. This incurs overhead to ensure integrity of the stream. The unlocked version of this call, `getc_unlocked` can be used to avoid the overhead. The `getc_unlocked` macro is functionally identical to the `getc` macro, except that it is not required to be implemented in a thread-safe manner. The `getc_unlocked` macro can be safely used only within a scope that is protected by the `flockfile` and `funlockfile` functions used as a pair. The caller must ensure that the stream is locked before `getc_unlocked` is used.

Since `getc_unlocked` is a macro, a file pointer argument with side effects might be evaluated incorrectly. In such a case, use the `fgetc_unlocked` function instead.

See also `flockfile`, `ftrylockfile`, and `funlockfile`.

Return Values

n

The returned character.

EOF

Indicates the end-of-file or an error.

[w]getch

[w]getch — Get a character from the terminal screen and echo it on the specified window. The `getch` function echoes the character on the `stdscr` window.

Format

```
#include <curses.h>
char getch();
char wgetch (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

The `getch` and `wgetch` functions refresh the specified window before fetching a character. For more information, see the `scrollok` function.

Return Values

x

The returned character.

ERR

Indicates that the function makes the screen scroll illegally.

getchar

getchar — Reads a single character from the standard input (`stdin`).

Format

```
#include <stdio.h>
int getchar (void);
```

Description

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

See also `getchar_unlocked`.

Return Values

x

The next character from `stdin`, converted to `int`.

EOF

Indicates the end-of-file or an error.

getchar_unlocked

getchar_unlocked — Same as `getchar`, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

```
#include <stdio.h>
int getchar_unlocked (void);
```

Description

The reentrant version of the `getchar` function is locked against multiple threads calling it simultaneously. This incurs overhead to ensure integrity of the input stream. The unlocked version of this call, `getchar_unlocked` can be used to avoid the overhead. The `getchar_unlocked` function is functionally identical to the `getchar` function, except that it is not required to be implemented in a thread-safe manner. The `getchar_unlocked` function can be safely used only within a scope that is protected by the `flockfile` and `funlockfile` functions used as a pair. The caller must ensure that the stream is locked before `getchar_unlocked` is used.

See also `flockfile`, `ftrylockfile`, and `funlockfile`.

Return Values

x

The next character from `stdin`, converted to `int`.

EOF

Indicates the end-of-file or an error.

getclock

getclock — Gets the current value of the systemwide clock.

Format

```
#include <timers.h>
int getclock (int clktyp, struct timespec *tp);
```

Arguments

clktyp

The type of systemwide clock.

tp

Pointer to a `timespec` structure space where the current value of the systemwide clock is stored.

Description

The `getclock` function sets the current value of the clock specified by *clktyp* into the location pointed to by *tp*.

The *clktyp* argument is given as a symbolic constant name, as defined in the `<timers.h>` header file. Only the `TIMEOFDAY` symbolic constant, which specifies the normal time-of-day clock to access for systemwide time, is supported.

For the clock specified by `TIMEOFDAY`, the value returned by this function is the elapsed time since the Epoch. The Epoch is referenced to 00:00:00 UTC (Coordinated Universal Time) 1 Jan 1970.

The `getclock` function returns a `timespec` structure, which is defined in the `<timers.h>` header file as follows:

```
struct  timespec {

 unsigned long  tv_sec   /* Elapsed time in seconds since the Epoch*/
 long           tv_nsec  /* Elapsed time as a fraction of a second */
                         /* since the Epoch (in nanoseconds)       */

};
```

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to one of the following values:

- EINVAL– The *clktyp* argument does not specify a known systemwide clock.

  Or, the value of SYS$TIMEZONE_DIFFERENTIAL logical is wrong.

- EIO– An error occurred when the systemwide clock specified by the *clktyp* argument was accessed.

getcwd

getcwd — Returns a pointer to the file specification for the current working directory.

Format

```
#include <unistd.h>
char *getcwd (char *buffer, size_t size); (ISO POSIX-1)
char *getcwd (char *buffer, unsigned int size, ...); (VSI C Extension)
```

Function Variants

The `getcwd` function has variants named `_getcwd32` and `_getcwd64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

buffer

Pointer to a character string large enough to hold the directory specification.

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

size

The length of the directory specification to be returned.

. . .

An optional argument that can be either 1 or 0. If you specify 1, the directory specification is returned in OpenVMS format. If you specify 0, the directory specification (pathname) is returned in UNIX style format. If you omit this argument, `getcwd` returns the filename according to your current command-language interpreter (CLI). For more information about UNIX style directory specifications, see Section 1.3.3.

Return Values

x

A pointer to the file specification.

NULL

Indicates an error.

get[w]delim

get[w]delim — Read characters from an input stream until the specified delimiter character is encountered.

Format

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

```
ssize_t getdelim (char **lineptr, size_t *n, int delimiter, FILE *stream);
ssize_t getwdelim (wchar_t **lineptr, size_t *n, wint_t delimiter,
FILE *stream);
```

Function Variants

The `getdelim` function has variants named `_getdelim32` and `_getdelim64` for use with 32-bit and 64-bit pointer sizes, respectively.

The `getwdelim` function has variants named `_getwdelim32` and `_getwdelim64` for use with 32-bit and 64-bit pointer sizes, respectively.

Arguments

lineptr

A pointer to the initial buffer or to a null pointer.

n

A pointer to the size of the initial buffer.

delimiter

The delimiter character.

stream

Valid input stream.

Description

The `getdelim` and `getwdelim` functions work like `getline` and `getwline`, respectively, except that a line delimiter other than newline can be specified as the delimiter argument. As with `getline` and `getwline` a delimiter character is not added if one was not present in the input before end of file was reached.

Return Value

n

On success, `getdelim` and `getwdelim` return the number of characters read, including the delimiter character, but not including the terminating null byte.

getdtablesize

getdtablesize — Gets the total number of file descriptors that a process can have open simultaneously.

Format

```
#include <unistd.h>
int getdtablesize (void);
```

Description

The `getdtablesize` function returns the total number of file descriptors that a process can have open simultaneously. Each process is limited to a fixed number of open file descriptors.

The number of file descriptors that a process can have open is the minimum of the following:

- C RTL open file limit – 65535 on OpenVMS Alpha and Integrity servers.
- SYSGEN CHANNELCNT parameter – permanent I/O channel count.
- Process open file quota FILLM parameter – number of open files that can be opened by a process at one time.

Return Values

x

The number of file descriptors that a process can have open simultaneously.

-1

Indicates an error.

getegid

getegid — With POSIX IDs disabled, this function is equivalent to `getgid` and returns the group number from the user identification code (UIC). With POSIX IDs enabled, this function returns the effective group ID of the calling process.

Format

```
#include <unistd.h>
gid_t getegid (void);
```

Description

The `getegid` function can be used with POSIX style identifiers (IDs) or with UIC-based identifiers.

POSIX style IDs are supported on OpenVMS Version 7.3-2 and higher.

With POSIX style IDs disabled, the `getegid` and `getgid` functions are equivalent and return the group number from the current UIC. For example, if the UIC is [313,031], 313 is the group number.

With POSIX style IDs enabled, `getegid` returns the effective group ID of the calling process, and `getgid` returns the real group ID of the calling process. The real group ID is specified at login time. The effective group ID is more transient, and determines additional access permission during execution of a *set-group-ID* process. It is for such processes that the `getgid` function is most useful.

The `getegid` function is always successful; no return value is reserved to indicate an error.

To enable/disable POSIX style IDs, see Section 1.6.

See also `geteuid` and `getuid`.

Return Value

x

The effective group ID (POSIX IDs enabled), or the group number from the UIC (POSIX IDs disabled).

getenv

getenv — 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);
```

Argument

name

One of the following values:

- HOME—Your login directory
- TERM—The type of terminal being used
- PATH—The default device and directory
- ***.?)**

  USER—The name of the user who initiated the process
- Logical name or command-language interpreter (CLI) symbolic name
- An environment variable set with `setenv` or `putenv`

The case of the specified *name* is important.

Description

In certain situations, the `getenv` function attempts to perform a logical name translation on the user-specified argument:

1. If the argument to `getenv` does not match any of the environment strings present in your environment array, `getenv` attempts to translate your argument as a logical name by searching the logical name tables indicated by the LNM$FILE_DEV logical, as is done for file processing.

   `getenv` first does a case-sensitive lookup. If that fails, it does a case-insensitive lookup. In most instances, logical names are defined in uppercase, but `getenv` can also find logical names that include lowercase letters.

   `getenv` does not perform iterative logical name translation.

2. If the logical name is a search list with multiple equivalence values, the returned value points to the first equivalence value. For example:

```
$ DEFINE A B,C
ptr = getenv("A");
```

   A returns a pointer to " B".

3. If no logical name exists, `getenv` attempts to translate the argument string as a CLI symbol. If it succeeds, it returns the translated symbol text. If it fails, the return value is NULL.

   `getenv` does not perform iterative CLI translation.

If your CLI is the DEC/Shell, the function does not attempt a logical name translation since Shell environment symbols are implemented as DCL symbols.

Notes

- In OpenVMS Version 7.1, a cache of OpenVMS environment variables (that is, logical names and DCL symbols) was added to the `getenv` function to avoid the library making repeated calls to translate a logical name or to obtain the value of a DCL symbol. By default, the cache is disabled. If your application does not need to track changes in OpenVMS environment variables that can occur during its execution, the cache can be enabled by enabling the DECC$ENABLE_GETENV_CACHE logical before invoking the application.
- Do not use the `setenv`, `getenv`, and `putenv` functions to manipulate symbols and logicals. Instead use the OpenVMS library calls `lib$set_logical`, `lib$get_logical`, `lib$set_symbol`, and `lib$get_symbol`. The * `env` functions deliberately provide UNIX behavior, and are not a substitute for these OpenVMS runtime library calls.

  OpenVMS DCL symbols, not logical names, are the closest analog to environment variables on UNIX systems. While `getenv` is a mechanism to retrieve either a logical name or a symbol, it maintains an internal cache of values for use with `setenv` and subsequent `getenv` calls. The `setenv` function does not write or create DCL symbols or OpenVMS logical names.

  This is consistent with UNIX behavior. On UNIX systems, `setenv` does not change or create any symbols that will be visible in the shell after the program exits.

Return Values

x

Pointer to an array containing the translated symbol. An equivalence name is returned at index zero.

NULL

Indicates that the translation failed.

geteuid

geteuid — With POSIX IDs disabled, this function is equivalent to `getuid` and returns the member number (in OpenVMS terms) from the user identification code (UIC). With POSIX IDs enabled, this function returns the effective user ID.

Format

```
#include <unistd.h>
uid_t geteuid (void);
```

Description

The `geteuid` function can be used with POSIX style identifiers (IDs) or with UIC-based identifiers.

POSIX style IDs are supported on OpenVMS Version 7.3-2 and higher.

With POSIX style IDs disabled (the default), the `geteuid` and `getuid` functions are equivalent and return the member number from the current UIC as follows:

- For programs compiled with the _VMS_V6_SOURCE feature-test macro or programs that do not include the `<unistd.h>` header file, the `getuid` and `geteuid` functions return the member number of the OpenVMS UIC. For example, if the UIC is [313,31], then the member number, 31, is returned.
- For programs compiled without the _VMS_V6_SOURCE feature-test macro that do include the `<unistd.h>` header file, the full UIC is returned in decimal after converting the octal representation to decimal. For example, if the UIC is [313, 31] then 13303833 is returned. (13303833 = 25 + 203 * 65536; Octal 31 = 25 decimal; Octal 313 = 203 decimal.)

With POSIX style IDs enabled, `geteuid` returns the effective user ID of the calling process, and `getuid` returns the real user ID of the calling process.

To enable/disable POSIX style IDs, see Section 1.6.

See also `getegid` and `getgid`.

Return Value

x

The effective user ID (POSIX IDs enabled), or the member number from the current UIC or the full UIC (POSIX IDs disabled).

getgid

getgid — With POSIX IDs disabled, this function is equivalent to `getegid` and returns the group number from the user identification code (UIC). With POSIX IDs enabled, this function returns the real group ID.

Format

```
#include <unistd.h>
gid_t getgid (void);
```

Description

The `getgid` function can be used with POSIX style identifiers or with UIC-based identifiers.

POSIX style IDs are supported on OpenVMS Version 7.3-2 and higher.

With POSIX style IDs disabled (the default), the `getegid` and `getgid` functions are equivalent and return the group number from the current UIC. For example, if the UIC is [313,031], 313 is the group number.

With POSIX style IDs enabled, `getegid` returns the effective group ID of the calling process, and `getgid` returns the real group ID of the calling process. The real group ID is specified at login time. The effective group ID is more transient, and determines additional access permission during execution of a *set-group-ID* process. It is for such processes that the `getgid` function is most useful.

To enable/disable POSIX style IDs, see Section 1.6.

See also `geteuid` and `getuid`.

Return Value

x

The real group ID (POSIX IDs enabled), or the group number from the current UIC (POSIX IDs disabled).

getgrent

getgrent — Gets a group database entry.

Format

```
#include <grp.h>
struct group *getgrent (void);
```

Description

The `getgrent` function returns the next group in the sequential search, returning a pointer to a structure containing the broken-out fields of an entry in the group database.

When first called, `getgrent` returns a pointer to a `group` structure containing the first entry in the group database. Thereafter, it returns a pointer to the next `group` structure in the group database, so successive calls can be used to search the entire database.

If an end-of-file or an error is encountered on reading, `getgrent` returns a NULL pointer and sets `errno`.

Return Values

x

Pointer to a `group` structure, if successful.

NULL

Indicates that an error occurred. The function sets `errno` to one of the following values:

- EACCES– The user process does not have appropriate privileges enabled to access the user authorization file.
- EINTR– A signal was intercepted during the operation.
- EIO– Indicates that an I/O error occurred.
- EMFILE– OPEN_MAX file descriptors are currently open in the calling process.
- ENFILE– The maximum allowable number of files is currently open in the system.

getgrent_r

getgrent_r — Gets a group database entry (reentrant).

Format

```
#include <grp.h>
int getgrent_r(struct group *grp, char *buffer, size_t bufsize,
struct group **result);
```

Function Variant

The `getgrent_r` function has a variant named `__getgrent_r64` for use with 64-bit pointers. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

grp

The storage area to hold the retrieved `group` structure.

buffer

A pointer to a temporary buffer that the function can use during the operation to store the longest group entry in the database.

bufsize

The length of the temporary buffer (in characters).

result

A pointer to the retrieved `group` structure.

Description

The `getgrent_r` function is the reentrant version of `getgrent`. The `getgrent_r` function returns a pointer to a structure containing the broken-out fields of a record in the group database.

When first called, `getgrent_r` returns a pointer to a group structure containing the first entry in the group database. Thereafter, it returns a pointer to the next group structure in the group database, so successive calls can be used to search the entire database.

It updates the group structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* argument, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the _SC_GETGR_R_SIZE_MAX parameter of the `sysconf` function.

If the requested entry is not found or an error is encountered, a NULL pointer is returned at the location pointed to by *result*.

Return Values

0

On success, *result* is a pointer to the `struct group`.

x

On error, the function returns an error value, and *result* is NULL.

getgrgid

getgrgid — Gets a group database entry for a group ID.

Format

```
#include <types.h>
#include <grp.h>
struct group *getgrgid (gid_t gid);
```

Argument

gid

The group ID of the group for which the group database entry is to be retrieved.

Description

The `getgrgid` function searches the group database for an entry with a matching *gid* and returns a pointer to the `group` structure containing the matching entry.

Return Values

x

Pointer to a valid `group` structure containing a matching entry.

NULL

An error occurred.

Note: The return value points to a static area that is overwritten by subsequent calls to `getgrent`, `getgrgid`, or `getgrnam`.

On error, the function sets `errno` to one of the following values:

- EACCES– The user process does not have appropriate privileges enabled to access the user authorization file.
- EIO – An I/O error has occurred.
- EINTR– A signal was intercepted during `getgrgid`.
- EMFILE– OPEN_MAX file descriptors are currently open in the calling process.
- ENFILE– The maximum allowable number of files is currently open in the system.

Applications checking for error situations must set `errno` to 0 before calling `getgrgid`. If `errno` is set on return, an error has occurred.

getgrgid_r

getgrgid_r — Gets a group database entry for a group ID.

Format

```
#include <types.h>
#include <grp.h>
int getgrgid_r (gid_t gid, struct group *grp, char *buffer, size_t bufsize,
struct group **result);
```

Arguments

gid

The group ID of the group for which the group database entry is to be retrieved.

grp

The storage area to hold the retrieved `group` structure.

buffer

The working buffer that is able to hold the longest group entry in the database.

bufsize

The length, in characters, of *buffer*.

result

Upon successful return, *result* points to the retrieved `group` structure.

Upon unsuccessful return, *result* is set to NULL.

Description

The `getgrgid_r` function updates the `group` structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching *gid*. Storage referenced by the `group` structure is allocated from the memory provided with the *buffer* argument, which is *bufsize* characters in size. The maximum size needed for this buffer

can be determined with the _SC_GETGR_R_SIZE_MAX parameter of the `sysconf` function. On error or if the requested entry is not found, a NULL pointer is returned at the location pointed to by *result*.

Return Values

0

Successful completion.

x

On error, the function sets the return value to one of the following:

- EACCES– The user process does not have appropriate privileges enabled to access theuser authorization file.
- EIO – An I/O error has occurred.
- EINTR– A signal was intercepted during `getgrgid`.
- EMFILE– OPEN_MAX file descriptors are currently open in the calling process.
- ENFILE– The maximum allowable number of files is currently open in the system.
- ERANGE– Insufficient storage was supplied through the *buffer* and *bufsize* arguments to contain the data to be referenced by the resulting `group` structure.

getgrnam

getgrnam — Gets a group database entry for a name.

Format

```
#include <types.h>
#include <grp.h>
struct group *getgrnam (const char *name);
```

Argument

name

The group name of the group for which the group database entry is to be retrieved.

Description

The `getgrnam` function searches the group database for an entry with a matching *name*, and returns a pointer to the `group` structure containing the matching entry.

Return Values

x

Pointer to a valid `group` structure containing a matching entry.

NULL

Indicates an error.

Note: The return value points to a static area which is overwritten by subsequent calls to `getgrent`, `getgrgid`, or `getgrnam`.

On error, the function sets the return value to one of the following:

- EACCES– The user process does not have appropriate privileges enabled to access theuser authorization file.
- EIO – An I/O error has occurred.
- EINTR– A signal was intercepted during `getgrnam`.
- EMFILE– OPEN_MAX file descriptors are currently open in the calling process.
- ENFILE– The maximum allowable number of files is currently open in the system.

Applications wishing to check for error situations should set `errno` to 0 before calling `getgrnam`. If `errno` is set on return, an error occurred.

getgrnam_r

getgrnam_r — Gets a group database entry for a name.

Format

```
#include <types.h>
#include <grp.h>
int getgrnam_r (const char *name, struct group *grp, char *buffer,
size_t bufsize, struct group **result);
```

Arguments

name

The group name of the group for which the group database entry is to be retrieved.

grp

The storage area to hold the retrieved `group` structure.

buffer

The working buffer that is able to hold the longest group entry in the database.

bufsize

The length, in characters, of *buffer*.

result

Upon successful return, *result* points to the retrieved `group` structure.

Upon unsuccessful return, *result* is set to NULL.

Description

The `getgrnam_r` function updates the `group` structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching *name*. Storage referenced by the `group` structure is allocated from the memory provided with the *buffer* argument, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the _SC_GETGR_R_SIZE_MAX parameter of the `sysconf` function. On error or if the requested entry is not found, a NULL pointer is returned at the location pointed to by *result*.

Return Values

0

Successful completion.

x

On error, the function sets the return value to one of the following:

- EACCES– The user process does not have appropriate privileges enabled to access the user authorization file.
- EIO – An I/O error has occurred.
- EINTR– A signal was intercepted during `getgrnam`.
- EMFILE– OPEN_MAX file descriptors are currently open in the calling process.
- ENFILE– The maximum allowable number of files is currently open in the system.
- ERANGE– Insufficient storage was supplied through the *buffer* and *bufsize* arguments to contain the data to be referenced by the resulting `group` structure.

getgroups

getgroups — Gets the current supplementary group IDs of the calling process.

Format

```
#include <unistd.h>
int getgroups (int gidsetsize, gid_t grouplist[]);
```

Arguments

gidsetsize

Indicates the number of entries that can be stored in the array pointed to by the *grouplist* parameter.

grouplist

Points to the array in which the supplementary group IDs of the process are stored. The effective group ID of the process is not returned by the `getgroups` function if it is not also a supplementary group ID of the calling process.

Description

The `getgroups` function gets the current supplementary group IDs of the calling process. The list is stored in the array pointed to by the *grouplist* parameter. The *gidsetsize* parameter indicates the number of entries that can be stored in this array.

The `getgroups` function never returns more IDs than the value indicated by the `sysconf` parameter _SC_NGROUPS_MAX.

See also `getgid` and `setsid`.

Return Value

n

The number of elements stored in the array pointed to by the grouplist parameter.

-1

Indicates failure. `errno` might be set to one of the following values:

- EFAULT– The *gidsetsize* and *grouplist* parameters specify an array that is partially or completely outside of the allocated address space of the process.
- EINVAL– The *gidsetsize* parameter is nonzero and smaller than the number of supplementary group IDs.

getifaddrs

getifaddrs — Gets interface addresses.

Format

```
#include <sys/socket.h>
#include <ifaddrs.h>
int getifaddrs(struct ifaddrs **ifap);
```

Function Variants

The `getifaddrs` function has variants named `_getifaddrs32` and `_getifaddrs64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

ifap

The address of a location where the function can store a pointer to a linked list of `ifaddrs` structures that contain the data related to the network interfaces on the local machine.

Description

The `getifaddrs` function creates a linked list of structures describing the network interfaces, one for each network interface on the host machine. The `getifaddrs` function stores a reference to a linked

list of the network interfaces on the local machine in the memory referenced by *ifap*. The list consists of `ifaddrs` structures, as defined in the include file `<ifaddrs.h>`. The `ifaddrs` structure contains the following entries:

```
struct    ifaddrs  *ifa_next;       /* Pointer to next struct */
char               *ifa_name;       /* Interface name */
u_int               ifa_flags;      /* Interface flags */
struct    sockaddr *ifa_addr;       /* Interface address */
struct    sockaddr *ifa_netmask;    /* Interface netmask */
struct    sockaddr *ifa_broadaddr;  /* Interface broadcast address */
struct    sockaddr *ifa_dstaddr;    /* P2P interface destination */
void               *ifa_data;       /* unused */
```

The data returned by `getifaddrs` is dynamically allocated and should be freed using `freeifaddrs` when no longer needed.

Return Values

0

Successful completion.

-1

Indicates an error. `errno` contains the error.

getitimer

getitimer — Returns the value of interval timers.

Format

```
#include <time.h>
int getitimer (int which, struct itimerval *value);
```

Arguments

which

The type of interval timer. The C RTL supports only ITIMER_REAL.

value

Pointer to an `itimerval` structure whose members specify a timer interval and the time left to the end of the interval.

Description

The `getitimer` function returns the current value for the timer specified by the *which* argument in the structure pointed to by *value*.

A timer value is defined by the `itimerval` structure:

```
struct itimerval {
        struct  timeval it_interval;
```

```
            struct  timeval it_value;
    };
```

The following table lists the values for the `itimerval` structure members:

itimerval Member Value	Meaning
`it_interval` = 0	Disables a timer after its next expiration and assumes `it_value` is nonzero.
`it_interval` = nonzero	Specifies a value used in reloading `it_value` when the timer expires.
`it_value` = 0	Disables a timer.
`it_value` = nonzero	Indicates the time to the next timer expiration.

Time values smaller than the resolution of the system clock are rounded up to this resolution.

The C RTL provides each process with one interval timer, defined in the `<time.h>` header file as ITIMER_REAL. This timer decrements in real time and delivers a SIGALRM signal when the timer expires.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to EINVAL (The *value* argument specified a time that was too large to handle.)

get[w]line

get[w]line — Read characters from an input stream.

Format

```
#include <stdio.h>
ssize_t getline (char **lineptr, size_t *n, FILE *stream);
ssize_t getwline (wchar_t **lineptr, size_t *n, FILE *stream);
```

Function Variants

The `getline` function has variants named `_getline32` and `_getline64` for use with 32-bit and 64-bit pointer sizes, respectively.

The `getwline` function has variants named `_getwline32` and `_getwline64` for use with 32-bit and 64-bit pointer sizes, respectively.

Arguments

lineptr

A pointer to the initial buffer or to a null pointer.

n

A pointer to the size of the initial buffer.

stream

Valid input stream.

Description

The `getline` and `getwline` read an entire line from *stream*, storing the address of the buffer, which contains the text into **lineptr*. The buffer is null-terminated and includes the newline characterqa if one was found.

If **lineptr* is NULL, then `getline` will allocate a buffer for storing the line, which should be freed by the user program. (In this case, the value in *n* is ignored.)

Alternatively, before calling `getline`, **lineptr* can contain a pointer to a `malloc` allocated buffer *n* bytes in size. If the buffer is not large enough to hold the line, `getline` resizes it with `realloc`, updating **lineptr* and *n* as necessary.

Return Value

n

On success, `getline` and `getwline` return the number of characters read, including the delimiter character, but not including the terminating null byte.

getlogin

getlogin — Gets the login name.

Format

```
#include <unistd.h>
char *getlogin (void);
int *getlogin_r (char *name, size_t namesize);
```

Description

The `getlogin` function returns the login name of the user associated with the current session. If `getlogin` returns a non-null pointer, then that pointer points to the name that the user logged in under, even if there are several login names with the same user ID.

The `getlogin_r` function is the reentrant version of `getlogin`. Upon successful completion, `getlogin_r` returns 0 and puts the name associated by the login activity with the controlling terminal of the current process in the character array pointed to by *name*. The array is *namesize* characters long and should have space for the name and the terminating null character. The maximum size of the login name is LOGIN_NAME_MAX.

If `getlogin_r` is successful, *name* points to the name the user used at login, even if there are several login names with the same user ID.

Return Values

x

Upon successful completion, `getlogin` returns a pointer to a null-terminated string in a static buffer.

0

Indicates successful completion of `getlogin_r`.

NULL

Indicates an error; `errno` is set.

getname

getname — Returns the file specification associated with a file descriptor.

Format

```
#include <unixio.h>
char *getname (int file_desc, char *buffer, ...);
```

Function Variants

The `getname` function has variants named `_getname32` and `_getname64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

file_desc

A file descriptor.

buffer

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

. . .

An optional argument that can be either 1 or 0. If you specify 1, the `getname` function returns the file specification in OpenVMS format. If you specify 0, the `getname` function returns the file specification in UNIX style format. If you omit this argument, the `getname` function returns the filename according to your current command-language interpreter (CLI). For more information about UNIX style file specifications, see Section 1.3.3.

Description

The `getname` function places the file specification into the area pointed to by *buffer* and returns that address. The area pointed to by *buffer* should be an array large enough to contain a fully qualified file specification (the maximum length is 256 characters).

Return Values

x

The address passed in the *buffer* argument.

0

Indicates an error.

getopt

getopt — A command-line parser that can be used by applications that follow UNIX command-line conventions.

Format

```
#include <unistd.h> (X/Open, POSIX-1)
#include <stdio.h> (X/Open, POSIX-2)
int getopt (int argc, char * const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

Arguments

argc

The argument count as passed to `main`.

argv

The argument array as passed to `main`.

optstring

A string of recognized option characters. If a character is followed by a colon, the option takes an argument.

Description

The variable `optind` is the index of the next element of the *argv* vector to be processed. It is initialized to 1 by the system, and it is updated by `getopt` when it finishes with each element of *argv*. When an element of *argv* contains multiple option characters, it is unspecified how `getopt` determines which options have already been processed.

The `getopt` function returns the next option character (if one is found) from *argv* that matches a character in *optstring*, if there is one that matches. If the option takes an argument, `getopt` sets the variable `optarg` to point to the option-argument as follows:

- If the option was the last character in the string pointed to by an element of *argv*, then `optarg` contains the next element of *argv*, and `optind` is incremented by 2. If the resulting value of `optind` is not less than *argc*, `getopt` returns an error, indicating a missing option-argument.

- Otherwise, `optarg` points to the string following the option character in that element of *argv*, and `optind` is incremented by 1.

If one of the following is true, `getopt` returns -1 without changing `optind`:

argv[`optind`] is a NULL pointer
* *argv*[`optind`] is not the character –
argv[`optind`] points to the string "–"

If *argv*[`optind`] points to the string "– –" `getopt` returns -1 after incrementing `optind`.

If `getopt` encounters an option character not contained in *optstring*, the question-mark character (?) is returned.

If `getopt` detects a missing argument, the colon character (:) is returned if the first character of *optstring* is a colon; otherwise, a question-mark character is returned.

In either of the previous two cases, `getopt` sets the variable `optopt` to the option character that caused the error. If the application has not set the variable `opterr` to 0 and the first character of *optstring* is not a colon, `getopt` also prints a diagnostic message to `stderr`.

Return Values

x

The next option character specified on the command line.

A colon is returned if `getopt` detects a missing argument and the first character of *optstring* is a colon.

A question mark is returned if `getopt` encounters an option character not in *optstring* or detects a missing argument and the first character of *optstring* is not a colon.

-1

When all command-line options are parsed.

Example

The following example shows how you might process the arguments for a utility that can take the mutually exclusive options *a* and *b* and the options *f* and *o*, both of which require arguments:

```
#include <unistd.h>

int main (int argc, char *argv[ ])
{
          int c;
          int bflg, aflg, errflg;
          char *ifile;
          char *ofile;
          extern char *optarg;
          extern int optind, optopt;
          .
          .
          .
          while ((c = getopt(argc, argv, ":abf:o:)) != -1) {
```

```
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                               errflg++;
                        else {
                               bflg++;
                               bproc();
                        }

                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case ':':      /* -f or -o without operand */
                        fprintf (stderr,
                         "Option -%c requires an operand\n"' optopt);
                        errflg++;
                        break;
                case '?':
                        fprintf (stderr,
                                "Unrecognized option -%c\n"' optopt);
                        errflg++;
                }
        }
        if (errflg) {
                fprintf (stderr, "usage: ...");
                exit(2);
        }
        for ( ; optind < argc; optind++)  {
                if (access(argv[optind], R_OK)) {
        .
        .
        .
}
```

This sample code accepts any of the following as equivalent:

```
cmd -ao arg path path
cmd -a -o arg path path
cmd -o arg -a path path
cmd -a -o arg - path path
cmd -a -oarg path path
cmd -aoarg path path
```

getpagesize

getpagesize — Gets the system page size.

Format

```
#include <unistd.h>
int getpagesize (void);
```

Description

The `getpagesize` function returns the number of bytes in a page. The system page size is useful for specifying arguments to memory management system calls.

The page size is a system page size and is not necessarily the same as the underlying hardware page size.

Return Value

x

Always indicates success. Returns the number of bytes in a page.

getpgid

getpgid — Gets the process group ID for a process.

Format

```
#include <unistd.h>
pid_t getpgid (pid_t pid);
```

Argument

pid

The process ID for which the group ID is being requested.

Description

The `getpgid` function returns the process group ID of the process specified by *pid*. If *pid* is 0, the `getpgid` function returns the process group ID of the calling process.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

Return Values

x

The process group ID of the session leader of the specified process.

(pid_t) -1

Indicates an error. The function sets `errno` to one of the following values:

- EPERM – The process specified by *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.

- ESRCH– There is no process with a process ID of *pid*.
- EINVAL– The value of *pid* is invalid.

getpgrp

getpgrp — Gets the process group ID of the calling process.

Format

```
#include <unistd.h>
pid_t getpgrp (void);
```

Description

The `getpgrp` function returns the process group ID of the calling process.

The `getpgrp` function is always successful, and no return value is reserved to indicate an error.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

Return Values

x

The process group ID of the calling process.

getpid

getpid — Returns the process ID of the current process.

Format

```
#include <unistd.h>
pid_t getpid (void);
```

Return Value

x

The process ID of the current process.

getppid

getppid — Returns the parent process ID of the calling process.

Format

```
#include <unistd.h>
pid_t getppid (void);
```

Return Values

x

The parent process ID.

0

Indicates that the calling process does not have a parent process.

getpwent

getpwent — Accesses user entry information in the user database, returning a pointer to a `passwd` structure.

Format

```
#include <pwd.h>
struct passwd *getpwent (void);
```

Function Variants

The `getpwent` function has variants named `__32_getpwent` and `__64_getpwent` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Description

The `getpwent` function returns a pointer to a structure containing fields whose values are derived from an entry in the user database. Entries in the database are accessed sequentially by `getpwent`. When first called, `getpwent` returns a pointer to a `passwd` structure containing the first entry in the user database. Thereafter, it returns a pointer to a `passwd` structure containing the next entry in the user database. Successive calls can be used to search the entire user database.

The `passwd` structure is defined in the `<pwd.h>` header file as follows:

pw_name	The name of the user.
pw_uid	The ID of the user.
pw_gid	The group ID of the principle group of the user.
pw_dir	The home directory of the user.
pw_shell	The initial program for the user.

If an end-of-file or an error is encountered on reading, `getpwent` returns a NULL pointer.

Because `getpwent` accesses the user authorization file (SYSUAF) directly, the process must have appropriate privileges enabled or the function will fail.

Notes

All information generated by the `getpwent` function is stored in a per-thread static area and is overwritten on subsequent calls to the function.

Password file entries that are too long are ignored.

Return Values

x

Pointer to a `passwd` structure, if successful.

NULL

Indicates an end-of-file or error occurred. The function sets `errno` to one of the following values:

- EIO– Indicates that an I/O error occurred or the user does not have appropriate privileges enabled to access the user authorization file (SYSUAF).
- EMFILE– OPEN_MAX file descriptors are currently open in the calling process.
- ENFILE– The maximum allowable number of files is currently open in the system.

getpwnam, getpwnam_r

getpwnam, getpwnam_r — The `getpwnam` function returns information about a user database entry for the specified *name*. The `getpwnam_r` function is a reentrant version of `getpwnam`.

Format

```
#include <pwd.h>
struct passwd *getpwnam (const char *name); (ISO POSIX-1)
struct passwd *getpwnam (const char *name, ...); (VSI C Extension)
int getpwnam_r (const char *name, struct passwd *pwd, char *buffer,
size_t bufsize, struct passwd **result (ISO POSIX-1),
int getpwnam_r (const char *name, struct passwd *pwd, char *buffer,
size_t bufsize, struct passwd **result, ...); (VSI C Extension),
```

Function Variants

The `getpwnam` and `getpwnam_r` functions have variants named `__32_getpwnam`, `_getpwnam_r32` and `__64_getpwnam`, `_getpwnam_r64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

name

The name of the user for which the attributes are to be read.

pwd

The address of a `passwd` structure into which the function writes its results.

buffer

A working buffer for the *result* argument that is able to hold the largest entry in the `passwd` structure. Storage referenced by the `passwd` structure is allocated from the memory provided with the *buffer* argument, which is *bufsize* characters in length.

bufsize

The length of the character array that *buffer* points to.

result

Upon successful return, is set to *pwd*. Upon unsuccessful return, the result is set to NULL.

. . .

An optional argument that can be either 1 or 0. If you specify 1, the directory specification is returned in OpenVMS format. If you specify 0, the directory specification (pathname) is returned in UNIX style format. If you omit this argument, the function returns the directory specification according to your current command-language interpreter. For more information about UNIX style directory specifications, see Section 1.3.3.

Description

The `getpwnam` function searches the user database for an entry with the specified *name*. The function returns the first user entry in the database with the `pw_name` member of the `passwd` structure that matches the *name* argument.

The `passwd` structure is defined in the `<pwd.h>` header file as follows:

pw_name	The user's login name.
pw_uid	The numerical user ID.
pw_gid	The numerical group ID.
pw_dir	The home directory of the user.
pw_shell	The initial program for the user.

Note

All information generated by the `getpwnam` function is stored in a per-thread static area and is overwritten on subsequent calls to the function.

The `getpwnam_r` function is the reentrant version of `getpwnam`. The `getpwnam_r` function updates the `passwd` structure pointed to by *pwd* and stores a pointer to that structure at the location pointed to by *result*. The structure will contain an entry from the user database that matches the specified *name*. Storage referenced by the structure is allocated from the memory provided with the *buffer* argument, which is *bufsize* characters in length. The maximum size needed for this buffer can be determined with the _SC_GETPW_R_SIZE_MAX parameter of the `sysconf` function. On error or if the requested entry is not found, a NULL pointer is returned at the location pointed to by *result*.

Applications wishing to check for error situations should set `errno` to 0 before calling `getpwnam`. If `getpwnam` returns a NULL pointer and `errno` is nonzero, an error occurred.

Return Values

x

`getpwnam` returns a pointer to a valid `passwd` structure, if a matching entry is found.

NULL

`getpwnam` returns NULL if an error occurred or the specified entry was not found. `errno` is set to indicate the error. The `getpwnam` function may fail if:

- EIO– An I/O error has occurred.
- EINTR– A signal was intercepted during `getpwnam`.
- EMFILE– OPEN_MAX file descriptors are currently open in the calling process.
- ENFILE– The maximum allowable number of files is currently open in the system.

0

When successful, `getpwnam_r` returns 0 and stores a pointer to the updated `passwd` structure at the location pointed to by *result*.

0

When unsuccessful (on error or if the requested entry is not found), `getpwnam_r` returns 0 and stores a NULL pointer at the location pointed to by *result*. The `getpwnam_r` function may fail if:

- ERANGE– Insufficient storage was supplied through *buffer* and *bufsize* to contain the data to be referenced by the resulting `passwd` structure.

Example

When building a sample program with /def=_USE_STD_STAT, you can observe the following:

- When the DECC$POSIX_STYLE_UID logical is enabled:
  - For a system, that supports POSIX style identifiers:

    - getpwnam_r API reads information from the TCP/IP proxy database and fills UID and GID with values from the TCP/IP proxy database.

    - getgrgid_r API returns gr_name and gr_mem from the right's database associated with GID returned by getpwnam_r API.

  - System with no support for POSIX style identifiers, getpwnam_r fills GID and UID with SYSGEN parameters as "DEFUID" and "DEFGID".
- When the DECC$POSIX_STYLE_UID logical is not defined:

  getpwnam function returns information about a user database entry for the specified name, which is specified in SYSUAF.DAT

```
#include <unistd>  // getuid()
#include <pwd>     // getpwuid_r()
#include <grp>
#include <errno.h>
#include <stdio.h>
#include <string.h>

main()
```

```
{

struct passwd pwd2;
const unsigned int PWD_BUFF_SIZE = 1024; const unsigned int GRP_BUFF_SIZE =
 1024;

struct passwd *p_passwd;
struct passwd *result;
struct group *grpresult;
struct group grp;
char pwdBuffer[PWD_BUFF_SIZE],*name;
char grpBuffer[GRP_BUFF_SIZE];
char buf[PWD_BUFF_SIZE];

gid_t gid;
uid_t uid;

int status;
p_passwd = getpwnam("user1");
uid=p_passwd->pw_uid;
gid=p_passwd->pw_gid;

printf("User id is %u\n", uid);
printf("Group id is %u\n", gid);

status = getpwnam_r("user1", &pwd2, pwdBuffer, PWD_BUFF_SIZE, &result);

gid = pwd2.pw_gid;

status = getgrgid_r(gid, &grp, grpBuffer, GRP_BUFF_SIZE, &grpresult);

gid=grp.gr_gid; name=grp.gr_name;

strcpy(name,grp.gr_name);

printf("Group id is %u\n", gid);
printf("Group name is %s\n", name);

}
```

Running the example program with `/def=_USE_STD_STAT` produces the following result:

- When the DECC$POSIX_STYLE_UID logical is NOT enabled, prints uid as 11010118 (result of 65536*168+ 70) and gid as 168 with group name as RTL.
- When the DECC$POSIX_STYLE_UID logical is enabled and POSIX style identifiers are supported, prints uid as 70, gid as 168 with group name as FOR_POSIX_TEST (retrieved from TCP/IP proxy database).
- When the DECC$POSIX_STYLE_UID logical is enabled, but POSIX style identifiers are not supported, prints uid as DEFUID, gid as DEFGID with group name as invalid buffer.

getpwuid, getpwuid_r

getpwuid, getpwuid_r — The `getpwuid` function returns information about a user database entry for the specified *uid*. The `getpwuid_r` function is a reentrant version of `getpwuid`.

Format

```
#include <pwd.h>
struct passwd *getpwuid (uid_t uid); (ISO POSIX-1)
struct passwd *getpwuid (uid_t uid, . . .); (VSI C Extension)
int getpwuid_r (uid_t uid, struct passwd *pwd, char *buffer,
size_t bufsize, struct passwd **result); (ISO POSIX-1)
int getpwuid_r (uid_t uid, struct passwd *pwd, char *buffer,
size_t bufsize, struct passwd **result, ...); (VSI C Extension)
```

Function Variants

The `getpwuid` and `getpwuid_r` functions have variants named `__32_getpwuid`, `_getpwuid_r32` and `__64_getpwuid`, `_getpwuid_r64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

uid

The user ID (UID) for which the attributes are to be read.

pwd

The location where the retrieved `passwd` structure is to be placed.

buffer

A working buffer for the *result* argument that is able to hold the entry in the `passwd` structure. Storage referenced by the `passwd` structure is allocated from the memory provided with the *buffer* argument, which is *bufsize* characters in size.

bufsize

The length of the character array that *buffer* points to.

result

Upon successful return, *result* is set to *pwd*. Upon unsuccessful return, *result* is set to NULL.

. . .

An optional argument that can be either 1 or 0. If you specify 1, the directory specification is returned in OpenVMS format. If you specify 0, the directory specification (pathname) is returned in UNIX style format. If you omit this argument, the function returns the directory specification according to your current command-language interpreter. For more information about UNIX style directory specifications, see Section 1.3.3.

Description

The `getpwuid` function searches the user database for an entry with the specified *uid*. The function returns the first user entry in the database with a `pw_uid` member of the `passwd` structure that matches the *uid* argument.

The `passwd` structure is defined in the `<pwd.h>` header file as follows:

pw_name	The user's login name.
pw_uid	The numerical user ID.
pw_gid	The numerical group ID.
pw_dir	The home directory of the user.
pw_shell	The initial program for the user.

Note

All information generated by the `getpwuid` function is stored in a per-thread static area and is overwritten on subsequent calls to the function.

The `getpwuid_r` function is the reentrant version of `getpwuid`. The `getpwuid_r` function updates the `passwd` structure pointed to by *pwd* and stores a pointer to that structure at the location pointed to by *result*. The structure will contain an entry from the user database with a matching *uid*. Storage referenced by the structure is allocated from the memory provided with the *buffer* argument, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the _SC_GETPW_R_SIZE_MAX parameter of the `sysconf` function. On error or if the requested entry is not found, a NULL pointer is returned at the location pointed to by *result*.

Applications wishing to check for error situations should set `errno` to 0 before calling `getpwuid`. If `getpwuid` returns a NULL pointer and `errno` is nonzero, an error occurred.

See also `getuid` to know how UIC is represented.

Return Values

x

`getpwuid` returns a pointer to a valid `passwd` structure, if a matching entry is found.

NULL

`getpwuid` returns NULL if an error occurred or a matching entry was not found. `errno` is set to indicate the error. The `getpwuid` function may fail if:

- EIO– An I/O error has occurred.
- EINTR– A signal was intercepted during `getpwnam`.
- EMFILE– OPEN_MAX file descriptors are currently open in the calling process.
- ENFILE– The maximum allowable number of files is currently open in the system.

0

When successful, `getpwuid_r` returns 0 and stores a pointer to the updated `passwd` structure at the location pointed to by *result*.

0

When unsuccessful (on error or if the requested entry is not found), `getpwuid_r` returns 0 and stores a NULL pointer at the location pointed to by *result*. The `getpwuid_r` function may fail if:

- ERANGE– Insufficient storage was supplied through *buffer* and *bufsize* to contain the data to be referenced by the resulting `passwd` structure.

getrusage

getrusage — Gets information about resource utilization.

Format

```
#include <sys/resource.h>
int getrusage(int who, struct rusage *r_usage);
```

Arguments

who

The process which you want to get the resource usage for.

r_usage

A pointer to an object of type `struct rusage` in which the function can store the resource usage information.

Description

The `getrusage` function provides measures of the resources used by the current process or its terminated and waited-for child processes.

If the value of the *who* argument is RUSAGE_SELF, information is returned about resources used by the current process. If the value of the *who* argument is RUSAGE_CHILDREN, information is returned about resources used by the terminated and waited-for children of the current process. If the child is never waited for, the resource information for the child process is discarded and not included in the resource information provided by `getrusage`.

Currently, only getting elapsed user time (`ru_utime`) and maximum resident memory (`ru_maxrss`) is supported.

Return Values

0

Successful completion.

-1

Indicated an error. `errno` contains the error.

gets

gets — Reads a line from the standard input (`stdin`).

Format

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

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

Function Variants

The `gets` function has variants named `_gets32` and `_gets64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

str

A pointer to a character string that is large enough to hold the information fetched from `stdin`.

Description

The new-line character (\n) that ends the line is replaced by the function with an ASCII null character (\0).

When `stdin` is opened in record mode, `gets` treats the end of a record the same as a new-line character and, therefore, reads up to and including a new-line character or to the end of the record.

Return Values

x

A pointer to the *str* argument.

NULL

Indicates that an error has occurred or that the end-of-file was encountered before a new-line character was encountered. The contents of *str* are undefined if a read error occurs.

getsid

getsid — Gets the process group ID of the session leader.

Format

```
#include <unistd.h>
pid_t getsid (pid_t pid);
```

Argument

pid

The process ID of the process whose session leader process group ID is being requested.

Description

The `getsid` function obtains the process group ID of the process that is the session leader of the process specified by *pid*. If *pid* is (pid_t) 0, it specifies the calling process.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

Return Values

x

The process group ID of the session leader of the specified process.

(pid_t) -1

Indicates an error. The function sets `errno` to one of the following values:

- EPERM – The process specified by *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of the session leader of that process from the calling process.
- ESRCH– There is no process with a process ID of *pid*.

[w]getstr

[w]getstr — Get a string from the terminal screen, store it in the variable *str*, and echo it on the specified window. The `getstr` function works on the `stdscr` window.

Format

```
#include <curses.h>
int getstr (char *str);
int wgetstr (WINDOW *win, char *str);
```

Arguments

win

A pointer to the window.

str

Must be large enough to hold the character string fetched from the window.

Description

The `getstr` and `wgetstr` functions refresh the specified window before fetching a string. The new-line terminator is stripped from the fetched string. For more information, see the `scrollok` function.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the screen scroll illegally.

gettimeofday

gettimeofday — Gets the date and time.

Format

```
#include <time.h>
int gettimeofday (struct timeval *tp, void *tzp);
```

Arguments

tp

Pointer to a `timeval` structure, defined in the `<time.h>` header file.

tzp

A NULL pointer. If this argument is not a NULL pointer, it is ignored.

Description

The `gettimeofday` function gets the current time (expressed as seconds and microseconds) since 00::00 Coordinated Universal Time, January 1, 1970. The current time is stored in the `timeval` structure pointed to by the *tp* argument.

The *tzp* argument is intended to hold time-zone information set by the kernel. However, because the OpenVMS kernel does not set time-zone information, the *tzp* argument should be NULL. If it is not NULL, it is ignored. This function is supported for compatibility with BSD programs.

If the value of the SYS$TIMEZONE_DIFFERENTIAL logical is wrong, the function fails with `errno` set to EINVAL.

Return Values

0

Indicates success.

-1

An error occurred. `errno` is set to indicate the error.

getuid

getuid — With POSIX IDs disabled, this function is equivalent to `geteuid` and returns the member number (in OpenVMS terms) from the user identification code (UIC). With POSIX IDs enabled, returns the real user ID.

Format

```
#include <unistd.h>
```

```
uid_t getuid (void);
```

Description

The `getuid` function can be used with POSIX style identifiers or with UIC-based identifiers.

POSIX style IDs are supported on OpenVMS Version 7.3-2 and higher.

With POSIX style IDs disabled (the default), the `geteuid` and `getuid` functions are equivalent and return the member number from the current UIC as follows:

- For programs compiled with the _VMS_V6_SOURCE feature-test macro or programs that do not include the `<unistd.h>` header file, the `getuid` and `geteuid` functions return the member number of the OpenVMS UIC. For example, if the UIC is [313,31], then the member number, 31, is returned.
- For programs compiled without the _VMS_V6_SOURCE feature-test macro that do include the `<unistd.h>` header file, the full UIC is returned in decimal after converting the octal representation to decimal. For example, if the UIC is [313, 31] then 13303833 is returned. (13303833 = 25 + 203 * 65536; Octal 31 = 25 decimal; Octal 313 = 203 decimal.)

With POSIX style IDs enabled, `geteuid` returns the effective user ID of the calling process, and `getuid` returns the real user ID of the calling process.

To enable/disable POSIX style IDs, see Section 1.6.

See also `getegid` and `getgid`.

Return Value

x

The real user ID (POSIX IDs enabled), or the member number from the current UIC or the full UIC (POSIX IDs disabled).

getw

getw — Returns characters from a specified file.

Format

```
#include <stdio.h>
int getw (FILE *file_ptr);
```

Argument

file_ptr

A pointer to the file to be accessed.

Description

The `getw` function returns the next four characters from the specified input file as an `int`.

Return Values

x

The next four characters, in an `int`.

EOF

Indicates that the end-of-file was encountered during the retrieval of any of the four characters and all four characters were lost. Since EOF is an acceptable integer, use `feof` and `ferror` to check the success of the function.

getwc

getwc — Reads the next character from a specified file, and converts it to a wide-character code.

Format

```
#include <wchar.h>
wint_t getwc (FILE *file_ptr);
```

Argument

file_ptr

A pointer to the file to be accessed.

Description

Since `getwc` is implemented as a macro, a file pointer argument with side effects (for example `getwc (*f++)`) might be evaluated incorrectly. In such a case, use the `fgetwc` function instead. See the `fgetwc` function.

Return Values

n

The returned character.

WEOF

Indicates the end-of-file or an error. If an error occurs, the function sets `errno`. For a list of the values set by this function, see `fgetwc`.

getwchar

getwchar — Reads a single wide character from the standard input (`stdin`).

Format

```
#include <wchar.h>
wint_t getwchar (void);
```

Description

The `getwchar` function is identical to `fgetwc(stdin)`.

Return Values

x

The next character from `stdin`, converted to `wint_t`.

WEOF

Indicates the end-of-file or an error. If an error occurs, the function sets `errno`. For a list of the values set by this function, see `fgetwc`.

getyx

getyx — Puts the (*y*, *x*) coordinates of the current cursor position on *win* in the variables *y* and *x*.

Format

```
#include <curses.h>
getyx (WINDOW *win, int y, int x);
```

Arguments

win

Must be a pointer to the window.

y

Must be a valid lvalue.

x

Must be a valid lvalue.

glob

glob — Returns a list of existing files for a user supplied pathname (with optional wildcards).

Format

```
#include <glob.h>
int glob (const char *pattern, int flags, int (*errfunc)(const char *epath,
int eerrno), glob_t *pglob);
```

Function Variants

The `glob` function has variants named `_glob32` and `_glob64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

pattern

The pattern string to match with accessible files and pathnames. This pattern can have wildcards.

flags

Controls the customizable behavior of the `glob` function.

errfunc

An optional function that, if specified, is called when the `glob` function detects an error condition, or if not specified, is NULL.

epath

First argument of the optional *errfunc* function, *epath* is the pathname that failed because a directory could not be opened or read.

eerrno

Second argument of the optional *errfunc* function, *eerrno* is the `errno` value from a failure specified by the *epath* argument as set by the `opendir`, `readdir`, or `stat` functions.

pglob

Pointer to a `glob_t` structure that returns the matching accessible existing filenames. The structure is allocated by the caller. The array of structures containing the located filenames that match the *pattern* argument are stored by the `glob` function into the structure. The last entry is a NULL pointer.

The structure type `glob_t` is defined in the `<glob.h>` header file and includes at least the following members:

```
size_t   gl_pathc    //Count of paths matched by pattern.
char **  gl_pathv    //Pointer to a list of matched pathnames.
size_t   gl_offs     //Slots to reserve at the beginning of gl_pathv.
```

Description

The `glob` function constructs a list of accessible files that match the *pattern* argument.

The `glob` function operates in one of two modes: UNIX mode or OpenVMS mode.

You can select UNIX mode explicitly by enabling the feature logical DECC$GLOB_UNIX_STYLE, which is disabled by default.

The `glob` function defaults to OpenVMS mode unless one of the following conditions is met (in which case `glob` uses UNIX mode):

- The DECC$GLOB_UNIX_STYLE is enabled.
- The DECC$FILENAME_UNIX_ONLY feature logical is enabled.
- The glob function checks the specified pattern for pathname indications, such as directory delimiters, and determines it to be a UNIX style pathname.

OpenVMS mode

This mode allows an OpenVMS programmer to give an OpenVMS style pattern to the `glob` function and get expected OpenVMS style output. The OpenVMS style pattern is what a user would expect from DCL commands or as input to the SYS$PARSE and SYS$SEARCH system routines.

In this mode, you can use any of the expected OpenVMS wildcards (see the OpenVMS documentation for additional information).

OpenVMS mode does not support the UNIX wildcard ?, or [] pattern matching. OpenVMS users expect [] to be available as directory delimiters.

Some additional behavior differences between OpenVMS mode and UNIX mode:

- OpenVMS mode outputs full file specifications, not relative ones, as in UNIX mode.
- The GLOB_MARK flag is ignored in OpenVMS mode because it is not meaningful to append a slash (/) to a directory on OpenVMS.

For example:

```
Sample pattern input      Sample output

[.SUBDIR1]A.TXT           DEV:[DIR.SUBDIR1]A.TXT;1
[.SUB*]%.*                DEV:[DIR.SUBDIR1]A.TXT;1
```

UNIX mode

You can enable this mode explicitly with:

```
$ DEFINE DECC$GLOB_UNIX_STYLE ENABLE
```

UNIX mode is also enabled if the DECC$FILENAME_UNIX_ONLY feature logical is set, or if the `glob` function determines that the specified pattern looks like a UNIX style pathname.

In UNIX mode, the `glob` function follows the X/Open specification where possible.

For example:

```
Sample pattern input      Sample output

./a/b/c                   ./a/b/c
./?/b/*                   ./a/b/c
[a-c]                     c
```

Standard Description

The `glob` function matches all accessible pathnames against this pattern and develops a list of all pathnames that match. To have access to a pathname, the `glob` function requires search permission on every component of a pathname except the last, and read permission on each directory of any filename component of the *pattern* argument.

The `glob` function stores the number of matched pathnames and a pointer to a list of pointers to pathnames in the *pglob* argument. The pathnames are sorted, based on the setting of the LC_COLLATE category in the current locale. The first pointer after the last pathname is NULL. If the pattern does not match any pathnames, the returned number of matched pathnames is 0.

It is the caller's responsibility to create the structure pointed to by the *pglob* argument. The `glob` function allocates other space as needed. The `globfree` function frees any space associated with the *pglob* argument as a result of a previous call to the `glob` function.

The *flags* argument is used to control the behavior of the `glob` function. The *flags* value is the bitwise inclusive OR (|) of any of the following constants, which are defined in the `<glob.h>` header file:

GLOB_APPEND	Appends pathnames located with this call to any pathnames previously located.
GLOB_DOOFFS	Uses the `gl_offs` structure to specify the number of NULL pointers to add to the beginning of the `gl_pathv` component of the *pglob* argument.
GLOB_ERR	Causes the `glob` function to return when it encounters a directory that it cannot open or read. If the GLOB_ERR flag is not set, the `glob` function continues to find matches if it encounters a directory that it cannot open or read.
GLOB_MARK	Specifies that each pathname that is a directory should have a slash (/) appended. GLOB_MARK is ignored in OpenVMS mode because it is not meaningful to append a slash to a directory on OpenVMS systems.
GLOB_NOCHECK	If the *pattern* argument does not match any pathname, then the `glob` function returns a list consisting only of the *pattern* argument, and the number of matched pathnames is 1.
GLOB_NOESCAPE	If the GLOB_NOESCAPE flag is set, a backslash (\) cannot be used to escape meta characters.

The GLOB_APPEND flag can be used to append a new set of pathnames to those found in a previous call to the `glob` function. The following rules apply when two or more calls to the `glob` function are made with the same value of the *pglob* argument, and without intervening calls to the `globfree` function:

- If the application sets the GLOB_DOOFFS flag in the first call to the `glob` function, then it is also set in the second call, and the value of the `gl_offs` field of the *pglob* argument is not modified between the calls.
- If the application did not set the GLOB_DOOFFS flag in the first call to the `glob` function, then it is not set in the second call.
- After the second call, *pglob* -> gl_pathvpoints to a list containing the following:
  - Zero or more NULLs, as specified by the GLOB_DOOFFS flag and *pglob* -> `gl_offs`.
  - Pointers to the pathnames that were in the *pglob* -> `gl_pathv` list before the call, in the same order as after the first call to the `glob` function.
  - Pointers to the new pathnames generated by the second call, in the specified order.
- The count returned in the *pglob* -> `gl_offs` argument is the total number of pathnames from the two calls.
- The application should not modify the *pglob* -> `gl_pathc` or *pglob* -> `gl_pathv` fields between the two calls.

On successful completion, the `glob` function returns a value of 0 (zero). The *pglob* -> `gl_pathc` field returns the number of matched pathnames and the *pglob* -> `gl_pathv` field contains a pointer to a NULL-terminated list of matched and sorted pathnames. If the number of matched pathnames in the *pglob* -> `gl_pathc` argument is 0 (zero), the pointer in the *pglob* -> `gl_pathv` argument is undefined.

If the `glob` function terminates because of an error, the function returns one of the nonzero constants GLOB_ABORTED, GLOB_NOMATCH, or GLOB_NOSPACE, defined in the `<glob.h>` header file. In this case, the *pglob* argument values are still set as defined above.

If, during the search, a directory is encountered that cannot be opened or read and the *errfunc* argument value is not NULL, the `glob` function calls *errfunc* with the two arguments *epath* and *eerno*:

epath—The pathname that failed because a directory could not be opened or read.
eerno—The `errno` value from a failure specified by the *epath* argument as set by the `opendir`, `readdir`, or `stat` functions.

If *errfunc* is called and returns nonzero, or if the GLOB_ERR flag is set in *flags*, the `glob` function stops the scan and returns GLOB_ABORTED after setting the *pglob* argument to reflect the pathnames already scanned. If GLOB_ERR is not set and either *errfunc* is NULL or *errfunc* returns zero, the error is ignored.

No `errno` values are returned.

See also `globfree`, `readdir`, and `stat`.

Return Values

0

Successful completion.

GLOB_ABORTED

The scan was stopped because GLOB_ERROR was set or *errfunc* returned a nonzero value.

GLOB_NOMATCH

The pattern does not match any existing pathname, and GLOB_NOCHECK was not set in *flags*.

GLOB_NOSPACE

An attempt to allocate memory failed.

globfree

globfree — Frees any space associated with the *pglob* argument resulting from a previous call to the `glob` function.

Format

```
#include <glob.h>
void globfree (glob_t *pglob);
```

Function Variants

The `globfree` function has variants named `_globfree32` and `_globfree64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

pglob

Pointer to a previously allocated `glob_t` structure.

Description

The `globfree` function frees any space associated with the *pglob* argument resulting from a previous call to the `glob` function. The `globfree` function returns no value.

gmtime, gmtime_r

gmtime, gmtime_r — Converts time units to the broken-down UTC time.

Format

```
#include <time.h>
struct tm *gmtime (const time_t *timer);
struct tm *gmtime_r (const time_t *timer, struct tm *result); (ISO POSIX-1)
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `gmtime_r` function that is equivalent to the behavior before OpenVMS Version 7.0.

Arguments

timer

Points to a variable that specifies a time value in seconds since the Epoch.

result

A pointer to a `tm` structure where the result is stored.

The `tm` structure is defined in the `<time.h>` header, and is also shown in Table 36 in the description of `localtime`.

Description

The `gmtime` and `gmtime_r` functions convert the time (in seconds since the Epoch) pointed to by *timer* into a broken-down time, expressed as Coordinated Universal Time (UTC), and store it in a `tm` structure.

The difference between the `gmtime_r` and `gmtime` functions is that the former puts the result into a user-specified `tm` structure where the result is stored. The latter puts the result into thread-specific static memory allocated by the C RTL, and which is overwritten by subsequent calls to `gmtime`; you must make a copy if you want to save it.

On success, `gmtime` returns a pointer to the `tm` structure; `gmtime_r` returns its second argument. On failure, these functions return the NULL pointer.

Note

Generally speaking, UTC-based time functions can affect in-memory time-zone information, which is processwide data. However, if the system time zone remains the same during the execution of the application (which is the common case) and the cache of timezone files is enabled (which is the default), then the `_r` variant of the time functions `asctime_r`, `ctime_r`, `gmtime_r` and `localtime_r`, is both thread-safe and AST-reentrant.

If, however, the system time zone can change during the execution of the application or the cache of timezone files is not enabled, then both variants of the UTC-based time functions belong to the third class of functions, which are neither thread-safe nor AST-reentrant.

Return Values

x

Pointer to a `tm` structure.

NULL

Indicates an error; `errno` is set to the following value:

- EINVAL– The *timer* argument is NULL.

gsignal

gsignal — Generates a specified software signal, which invokes the action routine established by a `signal`, `ssignal`, or `sigvec` function.

Format

```
#include <signal.h>
int gsignal (int sig [, int sigcode]);
```

Arguments

sig

The signal to be generated.

sigcode

An optional signal code. For example, signal SIGFPE—the arithmetic trap signal—has 10 different codes, each representing a different type of arithmetic trap.

The signal codes can be represented by mnemonics or numbers. The arithmetic trap codes are represented by the numbers 1 to 10, but the SIGILL codes are represented by the numbers 0 to 2. The code values are defined in the `<signal.h>` header file. See Table 4.4 for a list of signal mnemonics,codes, and corresponding OpenVMS exceptions.

Description

Calling the `gsignal` function has one of the following results:

- If `gsignal` specifies a *sig* argument that is outside the range defined in the `<signal.h>` header file, then `gsignal` returns 0 and sets `errno` to EINVAL.
- If `signal`, `ssignal`, or `sigvec` establishes SIG_DFL (default action) for the signal, then `gsignal` does not return. The image is exited with the OpenVMS error code corresponding to the signal.
- If `signal`, `ssignal`, or `sigvec` establishes SIG_IGN (ignore signal) as the action for the signal, then `gsignal` returns its argument, *sig*.
- `signal`, `ssignal`, or `sigvec` must be used to establish an action routine for the signal. That function is called and its return value is returned by `gsignal`.

See Chapter 4 for more information.

See also `raise`, `signal`, `ssignal`, and `sigvec`.

Return Values

0

Indicates a *sig* argument that is outside the range defined in the `<signal.h>` header file; `errno` is set to EINVAL.

sig

Indicates that SIG_IGN (ignore signal) has been established as the action for the signal.

x

Indicates that `signal`, `ssignal`,or `sigvec` has established an action function for the signal. That function is called, and its return value is returned by `gsignal`.

hypot

hypot — Returns the length of the hypotenuse of a right triangle.

Format

```
#include <math.h>
double hypot (double x, double y);
float hypotf (float x, float y);
long double hypotl (long double x, long double y);
```

Arguments

x

A real value.

y

A real value.

Description

The `hypot` functions return the length of the hypotenuse of a right triangle, where *x* and *y* represent the perpendicular sides of the triangle. The length is calculated as:

`sqrt(x^2 + y^2)`

On overflow, the return value is undefined, and `errno` is set to ERANGE.

Return Values

x

The length of the hypotenuse.

HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

0

Underflow occurred; `errno` is set to ERANGE.

NaN

x or *y* is NaN; `errno` is set to EDOM.

iconv

iconv — Converts characters coded in one codeset to characters coded in another codeset.

Format

```
#include <iconv.h>
size_t iconv (iconv_t cd, const char **inbuf,size_t *inbytesleft,
char **outbuf, size_t *outbytesleft);
```

Arguments

cd

A conversion descriptor. This is returned by a successful call to `iconv_open`.

inbuf

A pointer to a variable that points to the first character in the input buffer.

inbytesleft

Initially, this argument is a pointer to a variable that indicates the number of bytes to the end of the input buffer (*inbuf*). When the conversion is completed, the variable indicates the number of bytes in *inbuf* not converted.

outbuf

A pointer to a variable that points to the first available byte in the output buffer. The output buffer contains the converted characters.

outbytesleft

Initially, this argument is a pointer to a variable that indicates the number of bytes to the end of the output buffer (*outbuf*). When the conversion is completed, the variable indicates the number of bytes left in *outbuf*.

Description

The `iconv` function converts characters in the buffer pointed to by *inbuf* to characters in another code set. The resulting characters are stored in the buffer pointed to by *outbuf*. The conversion type is specified by the conversion descriptor *cd*. This descriptor is returned from a successful call to `iconv_open`.

If an invalid character is found in the input buffer, the conversion stops after the last successful conversion. The variable pointed to by *inbytesleft* is updated to reflect the number of bytes in the input buffer that are not converted. The variable pointed to by *outbytesleft* is updated to reflect the number of bytes remaining in the output buffer.

Return Values

x

Number of nonidentical conversions performed. Indicates successful conversion. In most cases, 0 is returned.

size_t -1

Indicates an error condition. The function sets `errno` to one of the following:

- EBADF – The *cd* argument is not a valid conversion descriptor.
- EILSEQ – The conversion stops when an invalid character detected.
- E2BIG – The conversion stops because of insufficient space in the output buffer.
- EINVAL – The conversion stops because of an incomplete character at the end of the input buffer.

iconv_close

iconv_close — Deallocates a specified conversion descriptor and the resources allocated to the descriptor.

Format

```
#include <iconv.h>
int iconv_close (iconv_t cd);
```

Argument

cd

The conversion descriptor to be deallocated. A conversion descriptor is returned by a successful call to `iconv_open`.

Return Values

0

Indicates that the conversion descriptor was successfully deallocated.

-1

Indicates an error occurred. The function sets `errno` to one of the following:

- EBADF – The *cd* argument is not a valid conversion descriptor.
- EVMSERR – Nontranslatable OpenVMS error occur. `vaxc$errno` contains the VMS error code.

iconv_open

iconv_open — Allocates a conversion descriptor for a specified codeset conversion.

Format

```
#include <iconv.h>
iconv_t iconv_open (const char *tocode, const char *fromcode);
```

Arguments

tocode

The name of the codeset to which characters are converted.

fromcode

The name of the source codeset. See Chapter 11 for information on obtaining a list of currently available codesets or for details on adding new codesets.

Return Values

x

A conversion descriptor. Indicates the call was successful. This descriptor is used in subsequent calls to `iconv`

iconv_t-1

Indicates an error occurred. The function sets `errno` to one of the following:

- EMFILE – The process does not have enough I/O channels to open a file.
- ENOMEM – Insufficient space is available.
- EINVAL – The conversion specified by *fromcode* and *tocode* is not supported.
- EVMSERR – Nontranslatable OpenVMS error occur. `vaxc$errno` contains the OpenVMS error code. A value of SS$_BADCHKSUM in `vaxc$errno` indicates that a conversion table file was found, but its contents is corrupted. A value of SS$_IDMISMATCH in `vaxc$errno` indicates that the conversion table file version does not match the version of the C Run-Time Library.

Example

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


int main()
{
  /* Declare variables to be used                       */

  char fromcodeset[30];
  char tocodeset[30];
  int iconv_opened;
  iconv_t iconv_struct;       /* Iconv descriptor      */

  /* Initialize variables                               */

  sprintf(fromcodeset, "DECHANYU");
  sprintf(tocodeset, "EUCTW");
  iconv_opened = FALSE;

  /* Attempt to create a conversion descriptor for the */
  /* codesets specified. If the return value from      */
  /* iconv_open is -1 then an error has occurred.      */
  /*  Check the value of errno.                        */

  if ((iconv_struct = iconv_open(tocodeset, fromcodeset))
     == (iconv_t) - 1) {

      /* Check the value of errno                       */

      switch (errno) {
      case EMFILE:
      case ENFILE:
          printf("Too many iconv conversion files open\n");
          break;

      case ENOMEM:
          printf("Not enough memory\n");
          break;
```

```
      case EINVAL:
          printf("Unsupported conversion\n");
          break;

      default:
          printf("Unexpected error from iconv_open\n");
          break;
     }
  }
  else

      /* Successfully allocated a conversion descriptor   */

      iconv_opened = TRUE;

  /*  Was a conversion descriptor allocated               */

   if (iconv_opened) {

      /* Attempt to deallocate the conversion descriptor. */
      /* If iconv_close returns -1 then an error has      */
      /* occurred.                                        */

      if (iconv_close(iconv_struct) == -1) {

          /* An error occurred. Check the value of errno  */

          switch (errno) {
          case EBADF:
              printf("Conversion descriptor is invalid\n");
              break;
          default:
              printf("Unexpected error from iconv_close\n");
              break;
          }
      }
  }
  return (EXIT_FAILURE);
}
```

ilogb

ilogb — Returns the exponent part of its argument.

Format

```
#include <math.h>
int ilogb (double x);
int ilogbf (float x);
int ilogbl (long double x);
```

Argument

x

A real value.

Description

The `ilogb` functions return the exponent part of their argument *x*. Formally, the return value is the integral part of logr |*x*| as a signed integral value, for nonzero *x*, where r is the radix of the machine's floating-point arithmetic, which is the value of FLT_RADIX defined in `<float.h>`.

Return Values

n

Upon success, the exponent part of *x* as a signed integer value. These functions are equivalent to calling the corresponding `logb` function and casting the returned value to type `int`.

[w]inch

[w]inch — Return the character at the current cursor position on the specified window without making changes to the window. The `inch` function acts on the `stdscr` window.

Format

```
#include <curses.h>
char inch();
char winch (WINDOW *win);
```

Argument

win

A pointer to the window.

Return Values

x

The returned character.

ERR

Indicates an input error.

index

index — Searches for a character in a string.

Format

```
#include <strings.h>
char *index (const char *s, int c);
```

Function Variants

The `index` function has variants named `_index32` and `_index64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s

The string to search.

c

The character to search for.

Description

The `index` function is identical to the `strchr` function, and is provided for compatibility with some UNIX implementations.

initscr

initscr — Initializes the terminal-type data and all screen functions. You must call `initscr` before using any of the curses functions.

Format

```
#include <curses.h>
void initscr (void);
```

Description

The OpenVMS Curses version of the `initscr` function clears the screen before doing the initialization. The BSD-based Curses version does not.

initstate

initstate — Initializes random-number generators.

Format

```
#include <stdlib.h>
char *initstate (unsigned int seed, char *state, int size);
```

Arguments

seed

An initial seed value.

state

Pointer to an array of state information.

size

The size of the state information array.

Description

The `initstate` function initializes random-number generators. It lets you initialize, for future use, a state array passed as an argument. The size, in bytes, of the state array is used by the `initstate` function to decide how sophisticated a random-number generator to use; the larger the state array, the more random the numbers.

Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Amounts less than 8 bytes generate an error, while other amounts are rounded down to the nearest known value.

The *seed* argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The `initstate` function returns a pointer to the previous state information array.

Once you initialize a state, the `setstate` function allows rapid switching between states. The array defined by the *state* argument is used for further random-number generation until the `initstate` function is called or the `setstate` function is called again. The `setstate` function returns a pointer to the previous state array.

After initialization, you can restart a state array at a different point in one of two ways:

- Use the `initstate` function with the desired *seed* argument, *state* array, and *size* of the array.
- Use the `setstate` function with the desired state, followed by the `srandom` function with the desired *seed*. The advantage of using both functions is that you do not have to save the state array size once you initialize it.

See also `setstate`, `srandom`, and `random`.

Return Values

x

A pointer to the previous state array information.

0

Indicates an error. Call made with less than 8 bytes of state information. Further specified in the global `errno`.

[w]insch

[w]insch — Insert a character at the current cursor position in the specified window. The `insch` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int insch (char ch);
int winsch (WINDOW *win, char ch);
```

Arguments

win

A pointer to the window.

ch

The character to be inserted.

Description

After the character is inserted, each character on the line shifts to the right, and the last character in the line is deleted. For more information, see the `scrollok` function.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the screen scroll illegally.

[w]insertln

[w]insertln — Insert a line above the line containing the current cursor position. The `insertln` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int insertln();
int winsertln (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

The current line and every line below it shifts down, and the bottom line disappears. The inserted line is blank and the current (*y*, *x*) coordinates remain the same. For more information, see the `scrollok`function.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the screen scroll illegally.

[w]insstr

[w]insstr — Insert a string at the current cursor position in the specified window. The `insstr` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int insstr (char *str);
int winsstr (WINDOW *win, char *str);
```

Arguments

win

A pointer to the window.

str

A pointer to the string to be inserted.

Description

Each character after the string shifts to the right, and the last character disappears. These functions are specific to VSI C for OpenVMS systems and are not portable.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the screen scroll illegally. For more information, see the `scrollok` function.

isalnum

isalnum — Indicates if a character is classed either as alphabetic or as a digit in the program's current locale.

Format

```
#include <ctype.h>
int isalnum (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If alphanumeric.

0

If not alphanumeric.

isalpha

isalpha — Indicates if a character is classed as an alphabetic character in the program's current locale.

Format

```
#include <ctype.h>
int isalpha (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If alphabetic.

0

If not alphabetic.

isapipe

isapipe — Indicates if a specified file descriptor is associated with a pipe.

Format

```
#include <unixio.h>
int isapipe (int file_desc);
```

Argument

file_desc

A file descriptor.

Description

For more information about pipes, see Chapter 5.

Return Values

1

Indicates an association with a pipe.

0

Indicates no association with a pipe.

-1

Indicates an error (for example, if the file descriptor is not associated with an open file).

isascii

isascii — Indicates if a character is an ASCII character.

Format

```
#include <ctype.h>
int isascii (int character);
```

Argument

character

An object of type `char`.

Return Values

nonzero

If ASCII.

0

If not ASCII.

isatty

isatty — Indicates if a specified file descriptor is associated with a terminal.

Format

```
#include <unistd.h>
int isatty (int file_desc);
```

Argument

file_desc

A file descriptor.

Return Values

1

If the file descriptor is associated with a terminal.

0

Indicates an error (for example, if the file descriptor is not associated with an open file) and `errno` is set to indicate the error.

isblank

isblank — Checks if a character is blank.

Format

```
#include <ctype.h>
int isblank (int c);
```

Argument

c

Character to be checked.

Description

The `isblank` function tests for any character that is a standard blank character or is one of a locale-specific set of characters for which `isspace` is true and that is used to separate words within a line of text.

The standard blank characters are the space character (' ') and the horizontal tab character ('\t').

In the "C" locale, `isblank` returns true only for the standard blank characters.

See also `iswblank`.

Return Values

nonzero

If a blank character.

0

If not a blank character.

iscntrl

iscntrl — Indicates if a character is classed as a control character in the program's current locale.

Format

```
#include <ctype.h>
int iscntrl (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a control character.

0

If not a control character.

isdigit

isdigit — Indicates if a character is classed as a digit in the program's current locale.

Format

```
#include <ctype.h>
int isdigit (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a decimal digit.

0

If not a decimal digit.

isgraph

isgraph — Indicates if a character is classed as a graphic character in the program's current locale.

Format

```
#include <ctype.h>
int isgraph (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a graphic character.

0

If not a graphic character.

isgreater

isgreater — Determines whether *x* is greater than *y*.

Format

```
#include <math.h>
int isgreater (x, y);
```

Arguments

x, y

Values to be compared.

Description

The function fails if one of the arguments is NaN. This causes an exception. To avoid this, C99 defines the macro `isgreater (x,y)`, which determines (x) > (y) without an exception if *x* or *y* is NaN.

The macro is guaranteed to evaluate the arguments only once. The arguments must be of real floating-point type.

Note

Do not pass integer values as arguments to this macro because the arguments will not be promoted to real floating-point types.

Return Values

The `isgreater (x,y)` macro returns the result of the relational comparison. It returns 0 if either argument is a NaN.

isgreaterequal

isgreaterequal — Determines whether *x* is greater than or equal to *y*.

Format

```
#include <math.h>
int isgreaterequal (x, y);
```

Arguments

x, y

Values to be compared.

Description

The function fails if one of the arguments is NaN. This causes an exception. To avoid this, C99 defines the macro `isgreaterequal (x, y)`, which determines (x) >= (y) without an exception if *x* or *y* is NaN.

The macro is guaranteed to evaluate the arguments only once. The arguments must be of real floating-point type.

Note

Do not pass integer values as arguments to this macro because the arguments will not be promoted to real floating-point types.

Return Values

The `isgreaterequal (x, y)` macro returns the result of the relational comparison. It returns 0 if either argument is a NaN.

isless

isless — Determines whether *x* is less than *y*.

Format

```
#include <math.h>
int isless (x, y);
```

Arguments

x, y

Values to be compared.

Description

The function fails if one of the arguments is NaN. This causes an exception. To avoid this, C99 defines the macro `isless (x, y)`, which determines (x) < (y) without an exception if *x* or *y* is NaN.

The macro is guaranteed to evaluate the arguments only once. The arguments must be of real floating-point type.

Note

Do not pass integer values as arguments to this macro because the arguments will not be promoted to real floating-point types.

Return Values

The `isless (x, y)` macro returns the result of the relational comparison. It returns 0 if either argument is a NaN.

islessequal

islessequal — Determines whether *x* is less than or equal to *y*.

Format

```
#include <math.h>
int islessequal (x, y);
```

Arguments

x, y

Values to be compared.

Description

The function fails if one of the arguments is NaN. This causes an exception. To avoid this, C99 defines the macro `islessequal (x, y)`, which determines (x) <= (y) without an exception if *x* or *y* is NaN.

The macro is guaranteed to evaluate the arguments only once. The arguments must be of real floating-point type.

Note

Do not pass integer values as arguments to this macro because the arguments will not be promoted to real floating-point types.

Return Values

The `islessequal (x, y)` macro returns the result of the relational comparison. It returns 0 if either argument is a NaN.

islessgreater

islessgreater — Determines whether *x* is less than or greater than *y*.

Format

```
#include <math.h>
int islessgreater (x, y);
```

Arguments

x, y

Values to be compared.

Description

The function fails if one of the arguments is NaN. This causes an exception. To avoid this, C99 defines the macro `islessgreater (x, y)`, which determines (x) < (y) || (x) > (y) without an exception if *x* or *y* is NaN.

Note

The `islessgreater (x, y)` macro is not equivalent to `x!=y` because that expression is true if *x* or *y* is NaN.

The macro is guaranteed to evaluate the arguments only once. The arguments must be of real floating-point type.

Note

Do not pass integer values as arguments to this macro because the arguments will not be promoted to real floating-point types.

Return Values

The `islessgreater (x, y)` macro returns the result of the relational comparison. It returns 0 if either argument is a NaN.

islower

islower — Indicates if a character is classed as a lowercase character in the program's current locale.

Format

```
#include <ctype.h>
int islower (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a lowercase alphabetic character.

0

If not a lowercase alphabetic character.

isnan

isnan — Tests for a NaN. Returns 1 if the argument is NaN; 0 if not.

Format

```
#include <math.h>
int isnan (double x);
int isnanf (float x);
int isnanl (long double x);
```

Argument

x

A real value.

Description

The `isnan` functions return the integer value 1 (TRUE) if *x* is NaN (the IEEE floating point reserved not-a-number value); otherwise, they return the value 0 (FALSE).

isprint

isprint — Indicates if a character is classed as a printing character in the program's current locale.

Format

```
#include <ctype.h>
int isprint (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a printing character.

0

If not a printing character.

ispunct

ispunct — Indicates if a character is classed as a punctuation character in the program's current locale.

Format

```
#include <ctype.h>
int ispunct (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a punctuation character.

0

If not a punctuation character.

isspace

isspace — Indicates if a character is classed as white space in the program's current locale; that is, if it is an ASCII space, tab (horizontal or vertical), carriage-return, form-feed, or new-line character.

Format

```
#include <ctype.h>
int isspace (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a white-space character.

0

If not a white-space character.

isunordered

isunordered — Determines whether *x* or *y* are unordered values.

Format

```
#include <math.h>
int isunordered (x, y);
```

Arguments

x, y

Values to check whether they are unordered.

Description

The function fails if one of the arguments is NaN. This causes an exception. To avoid this, C99 defines the macro `isunordered (x, y)`, which determines whether the arguments are unordered, that is, whether at least one of the arguments is a NaN.

The macro is guaranteed to evaluate the arguments only once. The arguments must be of real floating-point type.

Note

Do not pass integer values as arguments to this macro because the arguments will not be promoted to real floating-point types.

Return Values

1

If either *x* or *y* is NaN.

0

Otherwise.

isupper

isupper — Indicates if a character is classed as an uppercase character in the program's current locale.

Format

```
#include <ctype.h>
int isupper (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If an uppercase alphabetic character.

0

If not an uppercase alphabetic character.

iswalnum

iswalnum — Indicates if a wide character is classed either as alphabetic or as a digit in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswalnum (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *character* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If alphanumeric.

0

If not alphanumeric.

iswalpha

iswalpha — Indicates if a wide character is classed as an alphabetic character in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswalpha (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If alphabetic.

0

If not alphabetic.

iswblank

iswblank — Checks if a wide character is blank.

Format

```
#include <wctype.h>
int iswblank (wint_t wc);
```

Argument

wc

Wide character to be checked.

Description

The `iswblank` function tests for any wide character that is a standard blank wide character or is one of a locale-specific set of wide characters for which `iswspace` is true and that is used to separate words within a line of text.

The standard blank wide characters are the space character (L' ') and the horizontal tab character (L'\t').

In the "C" locale, `iswblank` returns true only for the standard blank characters.

See also `isblank`.

Return Values

nonzero

If a blank character.

0

If not a blank character.

iswcntrl

iswcntrl — Indicates if a wide character is classed as a control character in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswcntrl (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a control character.

0

If not a control character.

iswctype

iswctype — Indicates if a wide character has a specified property.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswctype (wint_t wc, wctype_t wc_prop);
```

Arguments

wc

An object of type `wint_t`. The value of *wc* must be representable as a valid wide-character code in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

wc_prop

A valid property name in the current locale. This is set up by calling the `wctype` function.

Description

The `iswctype` function tests whether *wc* has the character-class property *wc_prop*. Set *wc_prop* by calling the `wctype` function.

See also `wctype`.

Return Values

nonzero

If the character has the property *wc_prop*.

0

If the character does not have the property *wc_prop*.

Example

```
#include <locale.h>
#include <wchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

/* This test will set up the "upper" character class using      */
/* wctype() and then verify whether the characters 'a' and 'A'  */
/* are members of this class                                    */

#include <stdlib.h>

main()
{

    wchar_t w_char1,
            w_char2;
    wctype_t ret_val;
```

```
    char *char1 = "a";
    char *char2 = "A";

    ret_val = wctype("upper");

    /* Convert char1 to wide-character format - w_char1 */

    if (mbtowc(&w_char1, char1, 1) == -1) {
       perror("mbtowc");
        exit(EXIT_FAILURE);
    }

    if (iswctype((wint_t) w_char1, ret_val))
        printf("[%C] is a member of the character class upper\n",
                w_char1);
    else
     printf("[%C] is not a member of the character class upper\n",
              w_char1);

    /* Convert char2 to wide-character format - w_char2 */

    if (mbtowc(&w_char2, char2, 1) == -1) {
        perror("mbtowc");
        exit(EXIT_FAILURE);
    }

    if (iswctype((wint_t) w_char2, ret_val))
        printf("[%C] is a member of the character class upper\n",
                w_char2);
    else
     printf("[%C] is not a member of the character class upper\n",
                w_char2);
}
```

Running the example program produces the following result:

```
[a] is not a member of the character class upper
[A] is a member of the character class upper
```

iswdigit

iswdigit — Indicates if a wide character is classed as a digit in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswdigit (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a decimal digit.

0

If not a decimal digit.

iswgraph

iswgraph — Indicates if a wide character is classed as a graphic character in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswgraph (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a graphic character.

0

If not a graphic character.

iswlower

iswlower — Indicates if a wide character is classed as a lowercase character in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswlower (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a lowercase character.

0

If not a lowercase character.

iswprint

iswprint — Indicates if a wide character is classed as a printing character in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswprint (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a printing character.

0

If not a printing character.

iswpunct

iswpunct — Indicates if a wide character is classed as a punctuation character in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswpunct (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a punctuation character.

0

If not a punctuation character.

iswspace

iswspace — Indicates if a wide character is classed as a space character in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int iswspace (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a white-space character.

0

If not a white-space character.

iswupper

iswupper — Indicates if a wide character is classed as an uppercase character in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
```

```
#include <wchar.h> (XPG4)
int iswupper (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If an uppercase character.

0

If not an uppercase character.

iswxdigit

iswxdigit — Indicates if a wide character is a hexadecimal digit (0 to 9, A to F, or a to f) in the program's current locale.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
 int iswxdigit (wint_t wc);
```

Argument

wc

An object of type `wint_t`. The value of *wc* must be representable as a `wchar_t` in the current locale, or must equal the value of the macro WEOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a hexadecimal digit.

0

If not a hexadecimal digit.

isxdigit

isxdigit — Indicates if a character is a hexadecimal digit (0 to 9, A to F, or a to f) in the program's current locale.

Format

```
#include <ctype.h>
int isxdigit (int character);
```

Argument

character

An object of type `int`. The value of *character* must be representable as an `unsigned char` in the current locale, or must equal the value of the macro EOF. If it has any other value, the behavior is undefined.

Return Values

nonzero

If a hexadecimal digit.

0

If not a hexadecimal digit.

j0, j1, jn

j0, j1, jn — Compute Bessel functions of the first kind.

Format

```
#include <math.h>
double j0 (double x);
float j0f (float x);
long double j0l (long double x);
double j1 (double x);
float j1f (float x);
long double j1l (long double x);
double jn (int n, double x);
float jnf (int n, float x);
long double jnl (int n, long double x);
```

Arguments

x

A real value.

n

An integer.

Description

The `j0` functions return the value of the Bessel function of the first kind of order 0.

The `j1` functions return the value of the Bessel function of the first kind of order 1.

The `jn` functions return the value of the Bessel function of the first kind of order *n*.

The `j1` and `jn` functions can result in an underflow as *x* gets small. The largest value of *x* for which this occurs is a function of *n*.

Return Values

x

The relevant Bessel value of *x* of the first kind.

0

The value of the *x* argument is too large, or underflow occurred; `errno` is set to ERANGE.

NaN

x is NaN; `errno` is set to EDOM.

jrand48

jrand48 — Generates uniformly distributed pseudorandom-number sequences. Returns 48-bit signed, long integers.

Format

```
#include <stdlib.h>
long int jrand48 (unsigned short int xsubi[3]);
```

Argument

xsubi

An array of three `short int`s that form a 48-bit integer when concatenated together.

Description

The `jrand48` function generates pseudorandom numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The function returns signed long integers uniformly distributed over the range of *y* values, such that $-2^{31} \leq y < 2^{31}$.

The function works by generating a sequence of 48-bit integer values, X *i*, according to the linear congruential formula:

```
Xn+1 = (aXn+c)mod m        n >= 0
```

The argument *m* equals 2^{48}, so 48-bit integer arithmetic is performed. Unless you invoke the `lcong48` function, the multiplier value *a* and the addend value *c* are:

$a = 5DEECE66D_{16} = 273673163155_{8}$

$c = B_{16} = 13_8$

The `jrand48` function requires that the calling program pass an array as the *xsubi* argument, which for the first call must be initialized to the initial value of the pseudorandom-number sequence. Unlike the `drand48` function, it is not necessary to call an initialization function prior to the first call.

By using different arguments, `jrand48` allows separate modules of a large program to generate several independent sequences of pseudorandom numbers. For example, the sequence of numbers that one module generates does not depend upon how many times the function is called by other modules.

Return Value

n

Signed, long integers uniformly distributed over the range $-2^{31} \leq y < 2^{31}$.

kill

kill — Sends a signal to the process specified by a process ID.

Format

```
#include <signal.h>
int kill (int pid, int sig);
```

Arguments

pid

The process ID.

sig

The signal code.

Description

The `kill` function is restricted to C and C++ programs that include the `main` function.

The `kill` function sends a signal to a process, as if the process had called `raise`. If the signal is not trapped or ignored by the target program, the program exits.

OpenVMS VAX and Alpha implement different rules about what process you are allowed to send signals to. A program always has privileges to send a signal to a child started with `vfork/exec`. For other processes, the results are determined by the OpenVMS security model for your system.

Because of an OpenVMS restriction, the `kill` function cannot deliver a signal to a target process that runs an image installed with privileges.

Unless you have system privileges, the sending and receiving processes must have the same user identification code (UIC).

On OpenVMS systems before Version 7.0, `kill` treats a signal value of 0 as if SIGKILL were specified.

For OpenVMS Version 7.0 and higher systems, if you include `<stdlib.h>` and compile with the _POSIX_EXIT feature-test macro set, then:

- If the signal value is 0, `kill` validates the process ID but does not send any signals.
- If the process ID is not valid, `kill` returns -1 and sets `errno` to ESRCH.

Return Values

0

Indicates that `kill` was successfully queued.

-1

Indicates errors. The receiving process may have a different UIC and you are not a system user, or the receiving process does not exist.

l64a

l64a — Converts a long integer to a character string.

Format

```
#include <stdlib.h>
char *l64a (long l);
```

Argument

l

A long integer that is to be converted to a character string.

Description

The `a64l` and `l64a` functions are used to maintain numbers stored in base-64 ASCII characters:

- `a64l` converts a character string to a long integer.
- `l64a` converts a long integer to a character string.

Each character used to store a long integer represents a numeric value from 0 through 63. Up to six characters can be used to represent a long integer.

The characters are translated as follows:

- A period (.) represents 0.
- A slash (/) represents 1.
- The numbers 0 through 9 represent 2 through 11.
- Uppercase letters A through Z represent 12 through 37.

- Lowercase letters a through z represent 38 through 63.

The `l64a` function takes a long integer and returns a pointer to a corresponding base-64 notation of the least significant 32 bits.

The value returned by `l64a` is a pointer to a thread-specific buffer whose contents are overwritten on subsequent calls from the same

See also `a64l`.

Return Value

x

Upon successful completion, a pointer to the corresponding base-64 ASCII character-string notation. If the *l* parameter is 0, `l64a` returns a pointer to an empty string.

labs

labs — Returns the absolute value of an integer as a `long int`.

Format

```
#include <stdlib.h>
long int labs (long int j);
```

Argument

j

A value of type `long int`.

lchown

lchown — Changes the user and group ownership of the specified file.

Format

```
#include <unistd.h>
int lchown (const char *file_path, uid_t file_owner, gid_t file_group);
```

Arguments

file_path

The name of the file for which you want to change the owner and group IDs.

file_owner

The new user ID for the file.

file_group

The new group ID for the file.

Description

The `lchown` function changes the owner and/or group of the specified file (*file_path*). If the file is a symbolic link, the owner of the symbolic link is modified (in contrast to `chown` which would modify the file that the symbolic link points to).

See also `symlink`, `unlink`, `readlink`, `realpath`, and `lstat`.

Return Values

0

Successful completion.

-1

Indicates an error. `errno` is set to any `errno` value returned by `chown`.

lcong48

lcong48 — Initializes a 48-bit uniformly distributed pseudorandom-number sequence.

Format

```
#include <stdlib.h>
void lcong48 (unsigned short int param[7]);
```

Argument

param

An array that in turn specifies the initial X *i*, the multiplier value *a*, and the addend value *c*.

Description

The `lcong48` function generates pseudorandom numbers using the linear congruential algorithm and 48-bit integer arithmetic.

You can use `lcong48` to initialize the random number generator before you call any of the following functions:

```
drand48
lrand48
mrand48
```

The `lcong48` function specifies the initial X *i* value, the multiplier value *a*, and the addend value *c*.The *param* array elements specify the following:

param[0-2]	X *i*
param[3-5]	Multiplier *a* value

param[6]	16-bit addend *c* value

After `lcong48` has been called, a subsequent call to either `srand48` or `seed48` restores the standard *a* and *c* as specified previously.

The `lcong48` function does not return a value.

See also `drand48`, `lrand48`, `mrand48`, `srand48`, and `seed48`.

ldexp

ldexp — Returns its first argument multiplied by 2 raised to the power of its second argument; that is, $x(2^n)$.

Format

```
#include <math.h>
double ldexp (double x, int n);
float ldexp (float x, int n);
long double ldexp (long double x, int n);
```

Arguments

x

A base value of type `double`, `float`, or `long double` that is to be multiplied by 2^n.

n

The integer exponent value to which 2 is raised.

Return Values

$x(2^n)$

The first argument multiplied by 2 raised to the power of the second argument.

0

Underflow occurred; `errno` is set to ERANGE.

HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

NaN

x is NaN; `errno` is set to EDOM.

ldiv

ldiv — 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

A numerator of type `long int`.

denom

A denominator of type `long int`.

Description

The type `ldiv_t` is defined in the `<stdlib.h>` header file as follows:

```
typedef struct
        {
             long   quot, rem;
        } ldiv_t;
```

See also `div`.

leaveok

leaveok — Signals Curses to leave the cursor at the current coordinates after an update to the window.

Format

```
#include <curses.h>
leaveok (WINDOW *win, bool boolf);
```

Arguments

win

A pointer to the window.

boolf

A Boolean TRUE or FALSE value. If *boolf* is TRUE, the cursor remains in place after the last update and the coordinate setting on *win* changes accordingly. If *boolf* is FALSE, the cursor moves to the currently specified (*y*, *x*) coordinates of *win*.

Description

The `leaveok` function defaults to moving the cursor to the current coordinates of win. The `bool` type is defined in the `<curses.h>` header file as follows:

```
#define bool int
```

lgamma

lgamma — Computes the logarithm of the gamma function.

Format

```
#include <math.h>
double lgamma (double x);
float lgammaf (float x);
long double lgammal (long double x);
```

Argument

x

A real number. *x* cannot be 0, a negative integer, or Infinity.

Description

The `lgamma` functions return the logarithm of the absolute value of gamma of *x*, or ln(|G(*x*)|), where G is the gamma function.

The sign of gamma of *x* is returned in the external integer variable `signgam`. The *x* argument cannot be 0, a negative integer, or Infinity.

Return Values

x

The logarithmic gamma of the *x* argument.

-HUGE_VAL

The *x* argument is a negative integer; `errno` is set to ERANGE.

NaN

The *x* argument is NaN; `errno` is set to EDOM.

0

Underflow occurred; `errno` is set to ERANGE.

HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

link

link — Creates a new link (directory entry) for an existing file. This function is supported only on volumes that have hard link counts enabled.

Format

```
#include <unistd.h>
link (const char *path1, const char *path2);
```

Arguments

path1

Pointer to a pathname naming an existing file.

path2

Pointer to a pathname naming the new directory entry to be created.

Description

The `link` function atomically creates a new link for the existing file, and the link count of the file is incremented by one.

The `link` function can be used on directory files.

If `link` fails, no link is created and the link count of the file remains unchanged.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EEXIST– The link named by *path2* exists.
- EFTYPE– Wildcards appear in either *path1* or *path2*.
- EINVAL– One or both arguments specify a syntactically invalid pathname.
- ENAMETOOLONG– The length of *path1* or *path2* exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
- EXDEV– The link named by *path2* and the file named by *path1* are on different devices.

llrint

llrint — Rounds to the nearest integer value, using the current rounding direction, and casts to `long long` integer.

Format

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

```
long long int llrint (double x);
long long int llrintf (float x);
long long int llrintl (long double x);
```

Argument

x

Value to round.

Description

The `llrint` functions round *x* to the nearest integer value, using the current rounding direction, and return it as a value of type `long long int`.

If the rounded value is outside the range of the return type, the numeric result is unspecified and a *domain error* or *range error* may occur.

See also `lrint` for an equivalent function that returns a `long int`.

Return Value

n

On success, the function returns the rounded integer value.

llround

llround — Rounds to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction, and casts to `long long` integer.

Format

```
#include <math.h>
long long int llround (double x);
long long int llroundf (float x);
long long int llroundl (long double x);
```

Argument

x

Value to round.

Description

The `llround` functions round *x* to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction, and return it as a value of type `long long int`.

If the rounded value is outside the range of the return type, the numeric result is unspecified and a *domain error* or *range error* may occur.

See also `lround` for an equivalent function that returns a `long int`.

Return Value

n

On success, the function returns the rounded integer value.

localeconv

localeconv — Sets the members of a structure of type `struct lconv` with values appropriate for formatting numeric quantities according to the rules of the current locale.

Format

```
#include <locale.h>
struct lconv *localeconv (void);
```

Description

The `localeconv` function returns a pointer to the `lconv` structure defined in the `<locale.h>` header file. This structure should not be modified by the program. It is overwritten by calls to `localeconv`, or by calls to the `setlocale` function that change the LC_NUMERIC, LC_MONETARY, or LC_ALL categories.

The members of the structure are:

Member	Description
`char` *decimal_point	The radix character.
`char` *thousands_sep	The character used to separate groups of digits.
`char` *grouping	The string that defines how digits are grouped in nonmonetary values.
`char` *int_curr_symbol	The international currency symbol.
`char` *currency_symbol	The local currency symbol.
`char` *mon_decimal_point	The radix character used to format monetary values.
`char` *mon_thousands_sep	The character used to separate groups of digits in monetary values.
`char` *mon_grouping	The string that defines how digits are grouped in a monetary value.
`char` *positive_sign	The string used to indicate a nonnegative monetary value.
`char` *negative_sign	The string used to indicate a negative monetary value.
`char` int_frac_digits	The number of digits displayed after the radix character in a monetary value formatted with the international currency symbol.
`char` frac_digits	The number of digits displayed after the radix character in a monetary value.
`char` p_cs_precedes	For positive monetary values, this is set to 1 if the local or international currency symbol precedes the number, and it is set to 0 if the symbol succeeds the number.
`char` p_sep_by_space	For positive monetary values, this is set to 0 if there is no space between the currency symbol and the number. It is set to 1 if there is a space, and it is set to 2 if there is a space between the symbol and the sign string.

Member	Description
`char` n_cs_precedes	For negative monetary values, this is set to 1 if the local or international currency symbol precedes the number, and it is set to 0 if the symbol succeeds the number.
`char` n_sep_by_space	For negative monetary values, this is set to 0 if there is no space between the currency symbol and the number. It is set to 1 if there is a space, and it is set to 2 if there is a space between the symbol and the sign string.
`char` p_sign_posn	An integer used to indicate where the positive_sign string should be placed for a nonnegative monetary quantity.
`char` n_sign_posn	An integer used to indicate where the negative_sign string should be placed for a negative monetary quantity.

Members of the structure of type `char*` are pointers to strings, any of which (except decimal_point) can point to "", indicating that the associated value is not available in the current locale or is zero length. Members of the structure of type `char` are positive numbers, any of which can be CHAR_MAX, indicating that the associated value is not available in the current locale. CHAR_MAX is defined in the `<limits.h>` header file.

Be aware that the value of the CHAR_MAX macro in the `<limits.h>` header depends on whether the program is compiled with the /UNSIGNED_CHAR qualifier:

- Use the CHAR_MAX macro as an indicator of a nonavailable value in the current locale only if the program is compiled *without* /UNSIGNED_CHAR (/NOUNSIGNED_CHAR is the default).
- If the program is compiled *with* /UNSIGNED_CHAR, use the SCHAR_MAX macro instead of the CHAR_MAX macro.

In /NOUNSIGNED_CHAR mode, the values of CHAR_MAX and SCHAR_MAX are the same; therefore, comparison with SCHAR_MAX gives correct results regardless of the /[NO]UNSIGNED_CHAR mode used.

The members grouping and mon_grouping point to a string that defines the size of each group of digits when formatting a number. Each group size is separated by a semicolon (;). For example, if grouping points to the string 5;3 and the thousands_sep character is a comma (,), the number 123450000 would be formatted as 1,234,50000.

The elements of grouping and mon_grouping are interpreted as follows:

Value	Interpretation
CHAR_MAX	No further grouping is performed.
0	The previous element is to be used repeatedly for the remainder of the digits.
other	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values of p_sign_posn and n_sign_posn are interpreted as follows:

Value	Interpretation
0	Parentheses surround the number and currency symbol.

Value	Interpretation
1	The sign string precedes the number and currency symbol.
2	The sign string succeeds the number and currency symbol.
3	The sign string immediately precedes the number and currency symbol.
4	The sign string immediately succeeds the number and currency symbol.

Return Value

x

Pointer to the `lconv` structure.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <locale.h>
#include <string.h>

/* The following test program will set up the British English   */
/* locale, and then extract the International Currency symbol   */
/* and the International Fractional Digits fields for this      */
/* locale and print them.                                       */

int main()
{
    /* Declare variables                                        */

    char *return_val;
    struct lconv *lconv_ptr;

    /* Load a locale                                            */

    return_val = (char *) setlocale(LC_ALL, "en_GB.iso8859-1");

    /* Did the locale load successfully?                        */

    if (return_val == NULL) {

        /* It failed to load the locale                         */
       printf("ERROR : The locale is unknown");
        exit(EXIT_FAILURE);
    }

    /*  Get the lconv structure from the locale                 */

    lconv_ptr = (struct lconv *) localeconv();

    /* Compare the international currency symbol string with an */
    /* empty string. If they are equal, then the international  */
    /* currency symbol is not defined in the locale.            */

    if (strcmp(lconv_ptr->int_curr_symbol, "")) {
        printf("International Currency Symbol = %s\n",
```

```
                lconv_ptr->int_curr_symbol);
    }
    else {
        printf("International Currency Symbol =");
        printf("[Not available in this locale]\n");
    }

    /* Compare International Fractional Digits with CHAR_MAX.   */
    /* If they are equal, then International Fractional Digits  */
    /* are not defined in this locale.                          */

    if ((unsigned char) (lconv_ptr->int_frac_digits) != CHAR_MAX) {
        printf("International Fractional Digits = %d\n",
               lconv_ptr->int_frac_digits);
    }
    else {
        printf("International Fractional Digits =");
        printf("[Not available in this locale]\n");
    }
}
```

Running the example program produces the following result:

```
International Currency Symbol = GBP
International Fractional Digits = 2
```

localtime, localtime_r

localtime, localtime_r — Convert a time value to broken-down local time.

Format

```
#include <time.h>
struct tm *localtime (const time_t *timer);
struct tm *localtime_r (const time_t *timer, struct tm *result);
(ISO POSIX-1)
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `localtime_r` function that is equivalent to the behavior before OpenVMS Version 7.0.

Arguments

timer

A pointer to a time in seconds since the Epoch. You can generate this time by using the `time` function or you can supply a time.

result

A pointer to a `tm` structure where the result is stored. The `tm` structure is defined in the `<time.h>` header file, and is also shown in Table 36.

Description

The `localtime` and `localtime_r` functions convert the time (in seconds since the Epoch) pointed to by *timer* into a broken-down time, expressed as a local time, and store it in a `tm` structure.

The difference between the `localtime_r` and `localtime` functions is that the former stores the result into a user-specified `tm` structure. The latter stores the result into thread-specific static memory allocated by the C RTL, and which is overwritten by subsequent calls to `localtime`; you must make a copy if you want to save it.

On success, `localtime` returns a pointer to the `tm` structure; `localtime_r` returns its second argument. On failure, these functions return the NULL pointer.

The `tm` structure is defined in the `<time.h>` header file and described in Table 36.

Table 36. tm Structure

int `tm_sec`;	Seconds after the minute (0-60)
int `tm_min`;	Minutes after the hour (0-59)
int `tm_hour`;	Hours since midnight (0-23)
int `tm_mday`;	Day of the month (1-31)
int `tm_mon`;	Months since January (1-11)
int `tm_year`;	Years since 1900
int `tm_wday`;	Days since Sunday (0-6)
int `tm_yday`;	Days since January 1 (0-365)
int `tm_isdst`;	Daylight Savings Time flag • `tm_isdst` = 0 for Standard Time • `tm_isdst` = 1 for Daylight Time
long `tm_gmtoff`;[1]	Seconds east of Greenwich (negative values indicate seconds west of Greenwich)
char * `tm_zone`;[1]	Time zone string, for example "GMT"

[1]This field is an extension to the ANSI C structure. It is present unless you compile your program with /STANDARD=ANSI89 or with _DECC_V4_SOURCE defined.

The type `time_t` is defined in the `<time.h>` header file as follows:

```
typedef long int time_t
```

Note

Generally speaking, UTC-based time functions can affect in-memory time-zone information, which is processwide data. However, if the system time zone remains the same during the execution of the application (which is the common case) and the cache of timezone files is enabled (which is the default), then the `_r` variant of the time functions `asctime_r`, `ctime_r`, `gmtime_r` and `localtime_r`, is both thread-safe and AST-reentrant.

If, however, the system time zone can change during the execution of the application or the cache of timezone files is not enabled, then both variants of the UTC-based time functions belong to the third class of functions, which are neither thread-safe nor AST-reentrant.

Return Values

x

Pointer to a `tm` structure.

NULL

Indicates failure.

log, log2, log10

log, log2, log10 — Return the logarithm of their arguments.

Format

```
#include <math.h>
double log (double x);
float logf (float x);
long double logl (long double x);
double log2 (double x);
float log2f (float x);
long double log2l (long double x);
double log10 (double x);
float log10f (float x);
long double log10l (long double x);
```

Argument

x

A real number.

Description

The `log` functions compute the natural (base e) logarithm of *x*.

The `log2` functions compute the base 2 logarithm of *x*.

The `log10` functions compute the common (base 10) logarithm of *x*.

Return Values

x

The logarithm of the argument (in the appropriate base).

-HUGE_VAL

x is 0 (`errno` is set to ERANGE), or *x* is negative (`errno` is set to EDOM).

NaN

x is NaN; `errno` is set to EDOM.

log1p

log1p — Computes ln(1+ *y*) accurately.

Format

```
#include <math.h>
double log1p (double y);
float log1pf (float y);
long double log1pl (long double y);
```

Argument

y

A real number greater than -1.

Description

The `log1p` functions compute ln(1+ *y*) accurately, even for tiny *y*.

Return Values

x

The natural logarithm of (1+ *y*).

-HUGE_VAL

y is less than -1 (`errno` is set to EDOM), or *y* = -1 (`errno` is set to ERANGE).

NaN

y is NaN; `errno` is set to EDOM.

logb

logb — Returns the radix-independent exponent of the argument.

Format

```
#include <math.h>
double logb (double x);
float logbf (float x);
long double logbl (long double x);
```

Argument

x

A nonzero, real number.

Description

The `logb` functions return the exponent of *x*, which is the integral part of $\log_2 |x|$, as a signed floating-point value, for nonzero *x*.

Return Values

x

The exponent of *x*.

-HUGE_VAL

x = 0.0; `errno` is set to EDOM.

+Infinity

x is +Infinity or -Infinity.

NaN

y is NaN; `errno` is set to EDOM.

longjmp

longjmp — Provides a way to transfer control from a nested series of function invocations back to a predefined point without returning normally; that is, by not using a series of `return` statements. The `longjmp` function restores the context of the environment buffer.

Format

```
#include <setjmp.h>
void longjmp (jmp_buf env, int value);
```

Arguments

env

The environment buffer, which must be an array of integers long enough to hold the register context of the calling function. The type `jmp_buf` is defined in the `<setjmp.h>` header file. The contents of the general-purpose registers, including the program counter (PC), are stored in the buffer.

value

Passed from `longjmp` to `setjmp`, and then becomes the subsequent return value of the `setjmp` call. If *value* is passed as 0, it is converted to 1.

Description

When `setjmp` is first called, it returns the value 0. If `longjmp` is then called, naming the same environment as the call to `setjmp`, control is returned to the `setjmp` call as if it had returned

normally a second time. The return value of `setjmp` in this second return is the *value* you supply in the `longjmp` call. To preserve the true value of `setjmp`, the function calling `setjmp` must not be called again until the associated `longjmp` is called.

The `setjmp` function preserves the hardware general-purpose registers, and the `longjmp` function restores them. After a `longjmp`, all variables have their values as of the time of the `longjmp` except for local automatic variables not marked `volatile`. These variables have indeterminate values.

The `setjmp` and `longjmp` functions rely on the OpenVMS condition-handling facility to effect a nonlocal go to with a signal handler. The `longjmp` function is implemented by generating a C RTL specified signal and allowing the OpenVMS condition-handling facility to unwind back to the desired destination. The C RTL must be in control of signal handling for any VSI C image.

For VSI C to be in control of signal handling, you must establish all exception handlers through a call to the `VAXC$ESTABLISH` function (rather than `LIB$ESTABLISH`). See Section 4.2.5 and the `VAXC $ESTABLISH` function for more information.

Note

The C RTL provides nonstandard `decc$setjmp` and `decc$fast_longjmp` functions for Alpha and Integrity server systems. To use these nonstandard functions instead of the standard ones, a program must be compiled with the _ _FAST_SETJMP or _ _UNIX_SETJMP macros defined.

Unlike the standard `longjmp` function, the `decc$fast_longjmp` function does not convert its second argument from 0 to 1. After a call to `decc$fast_longjmp`, a corresponding `setjmp` function returns with the exact value of the second argument specified in the `decc$fast_longjmp` call.

Restrictions

You cannot invoke the `longjmp` function from an OpenVMS condition handler. However, you may invoke `longjmp` from a signal handler that has been established for any signal supported by the C RTL, subject to the following nesting restrictions:

- The `longjmp` function will not work if invoked from nested signal handlers. The result of the `longjmp` function, when invoked from a signal handler that has been entered as a result of an exception generated in another signal handler, is undefined.
- Do not invoke the `setjmp` function from a signal handler unless the associated `longjmp` is to be issued before the handling of that signal is completed.
- Do not invoke the `longjmp` function from within an exit handler (established with `atexit` or SYS$DCLEXH). Exit handlers are invoked after image tear-down, so the destination address of the `longjmp` no longer exists.
- Invoking `longjmp` from within a signal handler to return to the main thread of execution might leave your program in an inconsistent state. Possible side effects include the inability to perform I/O or to receive any more UNIX signals.

longname

longname — Returns the full name of the terminal.

Format

```
#include <curses.h>
void longname (char *termbuf, char *name);
```

Function Variants

The `longname` function has variants named `_longname32` and `_longname64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

termbuf

A string containing the name of the terminal.

name

A character-string buffer with a minimum length of 64 characters.

Description

The terminal name is in a readable format so that you can double-check to be sure that Curses has correctly identified your terminal. The dummy argument *termbuf* is required for UNIX software compatibility and serves no function in the OpenVMS environment. If portability is a concern, you must write a set of dummy routines to perform the functionality provided by the database `termcap` in the UNIX system environment.

lrand48

lrand48 — Generates uniformly distributed pseudorandom-number sequences. Returns 48-bit signed long integers.

Format

```
#include <stdlib.h>
long int lrand48 (void);
```

Description

The `lrand48` function generates pseudorandom numbers using the linear congruential algorithm and 48-bit integer arithmetic.

It returns nonnegative, long integers uniformly distributed over the range of y values such that $0 \leq y < 2^{31}$.

Before you call the `lrand48` function use either `srand48`, `seed48`, or `lcong48` to initialize the random-number generator. You must initialize prior to invoking the `lrand48` function, because it stores the last 48-bit X i generated into an internal buffer. (Although it is not recommended, constant

default initializer values are supplied automatically if the drand48, lrand48, or mrand48 functions are called without first calling an initialization function.)

The function works by generating a sequence of 48-bit integer values, X *i*, according to the linear congruential formula:

```
Xn+1 = (aXn+c)mod m        n >= 0
```

The argument *m* equals 2^{48}, so 48-bit integer arithmetic is performed. Unless you invoke the lcong48 function, the multiplier value *a* and the addend value *c* are:

$a = 5DEECE66D_{16} = 273673163155_8$
$c = B_{16} = 13_8$

The value returned by the lrand48 function is computed by first generating the next 48-bit X *i* in the sequence. Then the appropriate bits, according to the type of data item to be returned, are copied from the high-order (most significant) bits of X *i* and transformed into the returned value.

See also drand48, lcong48, mrand48, seed48, and srand48.

Return Value

n

Signed nonnegative long integers uniformly distributed over the range $0 \le y < 2^{31}$.

lrint

lrint — Rounds to the nearest integer value, rounding according to the current rounding direction.

Format

```
#include <math.h>
long lrint (double x);
long lrintf (float x);
long lrintl (long double x);
```

Argument

x

A real value.

Description

The lrint functions return the rounded integer value of *x*, rounded according to the current rounding direction.

Return Values

n

Upon success, the rounded integer value.

lround

lround — Rounds to the nearest integer value, rounding halfway cases away from zero regardless of the current rounding direction.

Format

```
#include <math.h>
long lround (double x);
long lroundf (float x);
long lroundl (long double x);
```

Argument

x

A real value.

Description

The `lround` functions return the rounded integer value of *x*, with halfway cases rounded away from zero regardless of the current rounding direction.

Return Values

n

Upon success, the rounded integer value.

lseek

— Positions a file to an arbitrary byte position and returns the new position.

Format

```
#include <unistd.h>
off_t lseek (int file_desc, off_t offset, int direction);
```

Arguments

file_desc

An integer returned by `open`, `creat`, `dup`, or `dup2`.

offset

The offset, specified in bytes. The `off_t` data type is either a 32-bit or a 64-bit integer. The 64-bit interface allows for file sizes greater than 2 GB, and can be selected at compile time by defining the _LARGEFILE feature-test macro as follows:

```
CC/DEFINE=_LARGEFILE
```

direction

An integer indicating whether the offset is to be measured forward from the beginning of the file (direction=SEEK_SET), forward from the current position (direction=SEEK_CUR), or backward from the end of the file (direction=SEEK_END).

Description

The `lseek` function can position a fixed-length record-access file with no carriage control or a stream-access file on any byte offset, but can position all other files only on record boundaries.

The available Standard I/O functions position a record file at its first byte, at the end-of-file, or on a record boundary. Therefore, the arguments given to `lseek` must specify either the beginning or end of the file, a 0 offset from the current position (an arbitrary record boundary), or the position returned by a previous, valid `lseek` call.

This function returns the new file position as an integer of type `off_t` which, like the *offset* argument, is either a 64-bit integer if _LARGEFILE is defined, or a 32-bit integer if not.

For a portable way to position an arbitrary byte location with any type of file, see the `fgetpos` and `fsetpos` functions.

Caution

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

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

See also `open`, `creat`, `dup`, `dup2`, and `fseek`.

Return Values

x

The new file position.

-1

Indicates that the file descriptor is undefined, or a seek was attempted before the beginning of the file.

lstat

lstat — Retrieves information about the specified file.

Format

```
#include <sys/stat.h>
int lstat (const char *restrict file_path,
struct stat *restrict user_buffer);
```

Arguments

file_path

The name of the file for which you want to retrieve information.

user_buffer

The `stat` structure in which information is returned.

Description

The `lstat` function retrieves information about the specified file (*file_path*). If the file is a symbolic link, information about the link itself is returned (in contrast to `stat`, which returns information about the file that the symbolic link points to).

See also `symlink`, `unlink`, `readlink`, `realpath`, and `lchown`.

Return Values

0

Successful completion.

-1

Indicates an error. `errno` is set to any `errno` value returned by `stat`.

lwait

lwait — Waits for I/O on a specific file to complete.

Format

```
#include <stdio.h>
int lwait (int fd);
```

Argument

fd

A file descriptor corresponding to an open file.

Description

The `lwait` function is used primarily to wait for completion of pending asynchronous I/O.

Return Values

0

Indicates successful completion.

-1

Indicates an error.

malloc

malloc — Allocates an area of memory. These functions are AST-reentrant.

Format

```
#include <stdlib.h>
void *malloc (size_t size);
```

Function Variants

The `malloc` function has variants named `_malloc32` and `_malloc64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

size

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.

Note

The `malloc` routines call the system routine LIB$VM_MALLOC. Because LIB$VM_MALLOC is designed as a general-purpose routine to allocate memory, it is called upon in a wide array of scenarios to allocate and reallocate blocks efficiently. The most common usage is the management of smaller blocks of memory, and the most important aspect of memory allocation under these circumstances is efficiency.

LIB$VM_MALLOC makes use of its own free space to satisfy requests, once the heap storage is consumed by splitting large blocks and merging adjacent blocks. Memory can still become fragmented, leaving unused blocks. Once heap storage is consumed, LIB$VM_MALLOC manages its own free space and merged blocks to satisfy requests, but varying sizes of memory allocations can cause blocks to be left unused.

Because LIB$VM_MALLOC cannot be made to satisfy all situations in the best possible manner, perform your own memory management if you have special memory usage needs. This assures the best use of memory for your particular application.

The *VSI OpenVMS Programming Concepts Manual* explains the several memory allocation routines that are available. They are grouped into three levels of hierarchy:

1. At the highest level are the RTL Heap Management Routines LIB$GET_VM and LIB$FREE_VM, which provide a mechanism for allocating and freeing blocks of memory of arbitrary size. Also at this level are the routines based on the concept of zones, such as LIB$CREATE_VM_ZONE, and so on.

2. At the next level are the RTL Page Management routines LIB$GET_VM_PAGE and LIB$FREE_VM_PAGE, which allocate a specified number of contiguous pages.

3. At the lowest level are the Memory Management System Services, such as $CRETVA and $EXPREG, that provide extensive control over address space allocation. At this level, you must manage the allocation precisely.

The maximum amount of memory allocated at once is limited to 0xFFFFD000.

Return Values

x

The address of the first byte, which is aligned on a quadword boundary (Alpha only) or an octaword boundary (Integrity servers only).

NULL

Indicates that the function is unable to allocate enough memory. `errno` is set to ENOMEM.

mblen

mblen — Determines the number of bytes comprising a multibyte character.

Format

```
#include <stdlib.h>
int mblen (const char *s, size_t n);
```

Arguments

s

A pointer to the multibyte character.

n

The maximum number of bytes that comprise the multibyte character.

Description

If the character is *n* bytes or less, the `mblen` function returns the number of bytes comprising the multibyte character pointed to by *s*. If the character is greater than *n* bytes, the function returns -1 to indicate an error.

This function is affected by the LC_CTYPE category of the program's current locale.

Return Values

x

The number of bytes that comprise the multibyte character, if the next *n* or fewer bytes form a valid character.

0

If *s* is NULL or a pointer to the NULL character.

-1

Indicates an error. The function sets `errno` to EILSEQ – Invalid character detected.

mbrlen

mbrlen — Determines the number of bytes comprising a multibyte character.

Format

```
#include <wchar.h>
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);
```

Arguments

s

A pointer to a multibyte character.

n

The maximum number of bytes that comprise the multibyte character.

ps

A pointer to the `mbstate_t` object. If a NULL pointer is specified, the function uses its internal `mbstate_t` object. `mbstate_t` is an opaque datatype intended to keep the conversion state for the state-dependent codesets.

Description

The `mbrlen` function is equivalent to the call:

```
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal)
```

Where *internal* is the `mbstate_t` object for the `mbrlen` function.

If the multibyte character pointed to by *s* is of *n* bytes or less, the function returns the number of bytes comprising the character (including any shift sequences).

If either an encoding error occurs or the next *n* bytes contribute to an incomplete but potentially valid multibyte character, the function returns -1 or -2, respectively.

See also `mbrtowc`.

Return Values

x

The number of bytes comprising the multibyte character.

0

Indicates that *s* is a NULL pointer or a pointer to a null byte.

-1

Indicates an encoding error, in which case the next *n* or fewer bytes do not contribute to a complete and valid multibyte character. `errno` is set to EILSEQ; the conversion state is undefined.

-2

Indicates an incomplete but potentially valid multibyte character (all *n* bytes have been processed).

mbrtowc

mbrtowc — Converts a multibyte character to its wide-character representation.

Format

```
#include <wchar.h>
size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

Arguments

pwc

A pointer to the resulting wide-character code.

s

A pointer to a multibyte character.

n

The maximum number of bytes that comprise the multibyte character.

ps

A pointer to the `mbstate_t` object. If a NULL pointer is specified, the function uses its internal `mbstate_t` object. `mbstate_t` is an opaque datatype intended to keep the conversion state for the state-dependent codesets.

Description

If *s* is a NULL pointer, `mbrtowc` is equivalent to the call:

```
mbrtowc(NULL, "", 1, ps)
```

In this case, the values of *pwc* and *n* are ignored.

If *s* is not a NULL pointer, `mbrtowc` inspects at most *n* bytes beginning with the byte pointed to by *s* to determine the number of bytes needed to complete the next multibyte character (including any shift sequences).

If the function determines that the next multibyte character is completed, it determines the value of the corresponding wide character and then, if *pwc* is not a NULL pointer, stores that value in the object pointed to by *pwc*. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

If `mbrtowc` is called as a counting function, which means that *pwc* is a NULL pointer and *s* is neither a NULL pointer nor a pointer to a null byte, the value of the internal `mbstate_t` object will remain unchanged.

Return Values

x

The number of bytes comprising the multibyte character.

0

The next *n* or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored if *pwc* is not a NULL pointer). The wide-character code corresponding to a null byte is zero.

-1

Indicates an encoding error. The next *n* or fewer bytes do not contribute to a complete and valid multibyte character. `errno` is set to `EILSEQ`. The conversion state is undefined.

-2

Indicates an incomplete but potentially valid multibyte character (all n bytes have been processed).

mbstowcs

mbstowcs — Converts a sequence of multibyte characters into a sequence of corresponding wide-character codes.

Format

```
#include <stdlib.h>
size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n);
```

Arguments

pwcs

A pointer to the array containing the resulting sequence of wide-character codes.

s

A pointer to the array of multibyte characters.

n

The maximum number of wide-character codes that can be stored in the array pointed to by *pwcs*.

Description

The `mbstowcs` function converts a sequence of multibyte characters from the array pointed to by *s* to a sequence of wide-character codes that are stored into the array pointed to by *pwcs*, up to a maximum of *n* codes.

This function is affected by the LC_CTYPE category of the program's current locale. If copying takes place between objects that overlap, the behavior is undefined.

Return Values

x

The number of array elements modified or required, not included any terminating zero code. The array will not be zero-terminated if the value returned is *n*. If *pwcs* is the NULL pointer, `mbstowcs` returns the number of elements required for the wide-character array.

size_t-1

Indicates that an error occurred. The function sets `errno` to EILSEQ - Invalid character detected.

mbtowc

mbtowc — Converts a multibyte character to its wide-character equivalent.

Format

```
#include <stdlib.h>
int mbtowc (wchar_t *pwc, const char *s, size_t n);
```

Arguments

pwc

A pointer to the resulting wide-character code.

s

A pointer to the multibyte character.

n

The maximum number of bytes that comprise the next multibyte character.

Description

If the character is *n* or fewer bytes, the `mbtowc` function converts the multibyte character pointed to by *s* to its wide-character equivalent. If the character is invalid or greater than *n* bytes, the function returns -1 to indicate an error.

If *pwc* is a NULL pointer and *s* is not a null pointer, the function determines the number of bytes that constitute the multibyte character pointed to by *s* (regardless of the value of *n*).

This function is affected by the LC_CTYPE category of the program's current locale.

Return Values

x

The number of bytes that comprise the valid character pointed to by *s*.

0

If *s* is either a NULL pointer or a pointer to the null byte.

-1

Indicates an error. The function sets `errno` to EILSEQ – Invalid character detected.

mbsinit

mbsinit — Determines whether an `mbstate_t` object describes an initial conversion state.

Format

```
#include <wchar.h>
int mbsinit (const mbstate_t *ps);
```

Argument

ps

A pointer to the `mbstate_t` object. `mbstate_t` is an opaque datatype intended to keep the conversion state for the state-dependent codesets.

Description

If *ps* is not a NULL pointer, the `mbsinit` function determines whether the `mbstate_t` object pointed to by *ps* describes an initial conversion state. A zero `mbstate_t` object always describes an initial conversion state.

Return Values

nonzero

The *ps* argument is a NULL pointer, or the `mbstate_t` object pointed to by *ps* describes an initial conversion state.

0

The `mbstate_t` object pointed to by *ps* does not describe an initial conversion state.

mbsrtowcs

mbsrtowcs — Converts a sequence of multibyte characters to a sequence of corresponding wide-character codes.

Format

```
#include <wchar.h>
size_t mbsrtowcs (wchar_t *dst, const char **src, size_t len,
mbstate_t *ps);
```

Function Variants

The `mbsrtowcs` function has variants named `_mbsrtowcs32` and `_mbsrtowcs64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dst

A pointer to the destination array containing the resulting sequence of wide-character codes.

src

An address of the pointer to an array containing a sequence of multibyte characters to be converted.

len

The maximum number of wide character codes that can be stored in the array pointed to by *dst*.

ps

A pointer to the `mbstate_t` object. If a NULL pointer is specified, the function uses its internal `mbstate_t` object. `mbstate_t` is an opaque datatype intended to keep the conversion state for the state-dependent codesets.

Description

The `mbsrtowcs` function converts a sequence of multibyte characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by *src*, into a sequence of corresponding wide characters.

If *dst* is not a NULL pointer, the converted characters are stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored.

Conversion stops earlier for one of the following reasons:

- A sequence of bytes is encountered that does not form a valid multibyte character.
- If *dst* is not a NULL pointer, when *len* codes have been stored into the array pointed to by *dst*.

If *dst* is not a NULL pointer, the pointer object pointed to by *src* is assigned either a NULL pointer (if the conversion stopped because of reaching a terminating null wide character), or the address just beyond the last multibyte character converted (if any). If conversion stopped because of reaching a terminating null wide character, the resulting state described is the initial conversion state.

Return Values

n

The number of multibyte characters successfully converted, sequence, not including the terminating null (if any).

-1

Indicates an error. A sequence of bytes that do not form valid multibyte character was encountered. `errno` is set to EILSEQ; the conversion state is undefined.

memccpy

memccpy — Copies characters sequentially between strings in memory areas.

Format

```
#include <string.h>
void *memccpy (void *dest, void *source, int c, size_t n);
```

Function Variants

The `memccpy` function has variants named `_memccpy32` and `_memccpy64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

A pointer to the location of a destination string.

source

A pointer to the location of a source string.

c

A character that you want to search for.

n

The number of character you want to copy.

Description

The `memccpy` function operates on strings in memory areas. A memory area is a group of contiguous characters bound by a count and not terminated by a null character. The function does not check for overflow of the receiving memory area. The `memccpy` function is defined in the `<string.h>` header file.

The `memccpy` function sequentially copies characters from the location pointed to by *source* into the location pointed to by *dest* until one of the following occurs:

- The character specified by *c* (converted to an `unsigned char`) is copied.
- The number of characters specified by *n* is copied.

Return Values

x

A pointer to the character following the character specified by *c* in the string pointed to by *dest*.

NULL

Indicates an error. The character *c* is not found after scanning *n* characters in the string.

memchr

memchr — Locates the first occurrence of the specified byte within the initial *size* bytes of a given object.

Format

```
#include <string.h>
void *memchr (const void *s1, int c, size_t size);
```

Function Variants

The `memchr` function has variants named `_memchr32` and `_memchr64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s1

A pointer to the object to be searched.

c

The byte value to be located.

size

The length of the object to be searched.

If *size* is zero, `memchr` returns NULL.

Description

Unlike `strchr`, the `memchr` function does not stop when it encounters a null character.

Return Values

pointer

A pointer to the first occurrence of the byte.

NULL

Indicates that the specified byte does not occur in the object.

memcmp

memcmp — Compares two objects, byte by byte. The compare operation starts with the first byte in each object.

Format

```
#include <string.h>
int memcmp (const void *s1, const void *s2, size_t size);
```

Arguments

s1

A pointer to the first object.

s2

A pointer to the second object.

size

The length of the objects to be compared.

If *size* is zero, the two objects are considered equal.

Description

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

Return Value

x

An integer less than, equal to, or greater than 0, depending on whether the lexical value of the first object is less than, equal to, or greater than that of the second object.

memcpy

memcpy — Copies a specified number of bytes from one object to another.

Format

```
#include <string.h>
void *memcpy (void *dest, const void *source, size_t size);
```

Function Variants

The `memcpy` function has variants named `_memcpy32` and `_memcpy64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

A pointer to the destination object.

source

A pointer to the source object.

size

The length of the object to be copied.

Description

The `memcpy` function copies *size* bytes from the object pointed to by *source* to the object pointed to by *dest*; it does not check for the overflow of the receiving memory area (*dest*). Unlike the `strcpy` function, the `memcpy` function does not stop when a null character is encountered.

Return Value

x

The value of *dest*.

memmove

memmove — Copies a specified number of bytes from one object to another.

Format

```
#include <string.h>
void *memmove (void *dest, const void *source, size_t size);
```

Function Variants

The memmove function has variants named _memmove32 and _memmove64 for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

A pointer to the destination object.

source

A pointer to the source object.

size

The length of the object to be copied.

Description

In VSI C for OpenVMS systems, memmove and memcpy perform the same function. Programs that require portability should use memmove if the area pointed at by *dest* could overlap the area pointed at by *source*.

Return Value

x

The value of *dest*.

Example

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

main()
{
   char pdest[14] = "hello   there";
   char *psource = "you are there";

   memmove(pdest, psource, 7);
   printf("%s\n", pdest);
}
```

This example produces the following output:

```
you are there
```

mempcpy

mempcpy — Copies a specified number of bytes from one object to another and returns a pointer to the byte following the last written byte.

Format

```
#include <string.h>
void *mempcpy (void *dest, const void *source, size_t size);
```

Function Variants

The `mempcpy` function has variants named `_mempcpy32` and `_mempcpy64` for use with 32-bit and 64-bit pointer sizes, respectively.

Arguments

dest

A pointer to the destination object.

source

A pointer to the source object.

size

The length of the object to be copied, in bytes.

Description

The `mempcpy` function, similar to the `memcpy` function, copies *size* bytes from the object pointed to by *source* to the object pointed to by *dest*; it does not check for the overflow of the receiving memory area (*dest*). Instead of returning the value of *dest*, `mempcpy` returns a pointer to the byte following the last written byte.

Return Value

x

A pointer to the byte following the last written byte.

memset

memset — Sets a specified number of bytes in a given object to a given value.

Format

```
#include <string.h>
void *memset (void *s, int value, size_t size);
```

Function Variants

The `memset` function has variants named `_memset32` and `_memset64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s

An array pointer.

value

The value to be placed in *s*.

size

The number of bytes to be placed in *s*.

Description

The `memset` function copies *value* (converted to an `unsigned char`) into each of the first `size` characters of the object pointed to by *s*.

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

Return Value

x

The value of *s*.

mkdir

mkdir — Creates a directory.

Format

```
#include <stat.h>
int mkdir (const char *dir_spec, mode_t mode); (ISO POSIX-1)
int mkdir (const char *dir_spec, mode_t mode, ...); (VSI C Extension)
```

Arguments

dir_spec

A valid OpenVMS or UNIX style directory specification that may contain a device name. For example:

```
DBA0:[BAY.WINDOWS]      /*    OpenVMS      */
/dba0/bay/windows       /*   UNIX style    */
```

This specification cannot contain a node name, filename, file extension, file version, or a wildcard character. The same restriction applies to the UNIX style directory specifications. For more information about the restrictions on UNIX style specifications, see Chapter 1.

mode

A file protection. See the `chmod` function in this section for information about the specific file protections.

The file protection of the new directory is derived from the *mode* argument, the process's file protection mask (see the `umask` function), and the parent-directory default protections.

In a manner consistent with the OpenVMS behavior for creating directories, `mkdir` never applies delete access to the directory. An application that needs to set delete access should use an explicit call to `chmod` to set write permission.

See the Description section of this function for more information about how the file protection is set for the newly created directory.

...

Represents the following optional arguments. These arguments have fixed position in the argument list, and cannot be arbitrarily placed.

`unsigned int` **uic**
The user identification code (UIC) that identifies the owner of the created directory. If this argument is 0, the C RTL gives the created directory the UIC of the parent directory. If this argument is not specified, the C RTL gives the created directory your UIC. This optional argument is specific to the C RTL and is not portable.
`unsigned short` **max_versions**
The maximum number of file versions to be retained in the created directory. The system automatically purges the directory keeping, at most, *max_versions* number of every file. If this argument is 0, the C RTL does not place a limit on the maximum number of file versions. If this argument is not specified, the C RTL gives the created directory the default version limit of the parent directory. This optional argument is specific to the C RTL and is not portable.
`unsigned short` **r_v_number**
The volume (device) on which to place the created directory if the device is part of a volume set. If this argument is not specified, the C RTL arbitrarily places the created directory within the volume set. This optional argument is specific to the C RTL and is not portable.

Description

If *dir_spec* specifies a path that includes directories, which do not exist, intermediate directories are also created. This differs from the behavior of the UNIX system where these intermediate directories must exist and will not be created.

If you do not specify any optional arguments, the C RTL gives the directory your UIC and the default version limit of the parent directory, and arbitrarily places the directory within the volume set. You cannot get the default behavior for the *uic* or *max_versions* arguments if you specify any arguments after them.

Note

The way to create files with OpenVMS RMS default protections using the UNIX system-call functions `umask`, `mkdir`, `creat`, and `open` is to call `mkdir`, `creat`, and `open` with a file-protection mode

argument of 0777 in a program that never specifically calls `umask`. These default protections include correctly establishing protections based on ACLs, previous versions of files, and so on.

In programs that do `vfork/exec` calls, the new process image inherits whether `umask` has ever been called or not from the calling process image. The `umask` setting and whether the `umask` function has ever been called are both inherited attributes.

The file protection supplied by the *mode* argument is modified by the process's file protection mask in such a way that the file protection for the new directory is set to the bitwise AND of the *mode* argument and the complement of the file protection mask.

Default file protections are supplied to the new directory from the parent-directory such that if a protection value bit in the new directory is zero, then the value of this bit is inherited from the parent directory. However, bits in the parent directory's file protection that indicate delete access do not cause corresponding bits to be set in the new directory's file protection.

Return Values

0

Indicates success.

-1

Indicates failure.

Examples

1. ```
   umask (0002); /* turn world write access off */
   mkdir ("sys$disk:[.parentdir.childdir]", 0222); /* turn write
                                                      access on */

   Parent directory file protection: System:RWD, Owner:RWD, Group:R,
                                                              World:R
   ```

   The file protection derived from the combination of the mode argument and the file protection mask set by `umask` is (0222) & ~(0002), which is 0220. When the parent directory defaults are applied to this protection, the protection for the new directory becomes:

   ```
   File protection:    System:RWD, Owner:RWD, Group:RWD, World:R
   ```

2. ```
   umask (0000);
   mkdir ("sys$disk:[.parentdir.childdir]", 0444);  /* turn read
                                                       access on */

   Parent directory file protection: System:RWD, Owner:RWD,
                                                    Group:RWD, World:RWD
   ```

   The file protection derived from the combination of the mode argument and the file protection mask set by `umask` is (0444) & ~(0000), which is 0444. When the parent directory defaults are applied to this protection, the protection for the new directory is:

   ```
   File protection:    System:RW, Owner:RW, Group:RW, World:RW
   ```

   Note that delete access is not inherited.

mkostemp

mkostemp — Constructs a unique filename, creates and opens the file specifying some flags.

Format

```
#include <stdlib.h>
int mkostemp (char *template, int flags)
```

Arguments

template

A pointer to a string that is replaced with a unique filename. The string in the *template* argument must be a filename with six trailing Xs.

flags

Zero or more of the flags as for the `open` function.

Description

The `mkostemp` function is equivalent to `mkstemp`, with the difference that flags as for `open` may be specified in *flags*.

The `mkostemp` function replaces the six trailing Xs of the string pointed to by *template* with a unique set of characters, and returns a file descriptor for the file opened using the flags specified in *flags*.

The string pointed to by *template* should look like a filename with six trailing Xs. The `mkostemp` function replaces each X with a character from the portable filename character set, making sure not to duplicate an existing filename.

If the string pointed to by *template* does not contain six trailing Xs, -1 is returned.

Return Values

x

An open file descriptor.

-1

Indicates an error. The string pointed to by *template* does not contain six trailing Xs.

mkstemp

mkstemp — Constructs a unique filename, creates and opens the file.

Format

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

```
int mkstemp (char *template);
```

Argument

template

A pointer to a string that is replaced with a unique filename. The string in the *template* argument must be a filename with six trailing Xs.

Description

The `mkstemp` function replaces the six trailing Xs of the string pointed to by *template* with a unique set of characters, and returns a file descriptor for the file open for reading and writing.

The string pointed to by *template* should look like a filename with six trailing Xs. The `mkstemp` function replaces each X with a character from the portable file-name character set, making sure not to duplicate an existing filename.

If the string pointed to by *template* does not contain six trailing Xs, -1 is returned.

Return Values

x

An open file descriptor.

-1

Indicates an error. The string pointed to by *template* does not contain six trailing Xs.

mktemp

mktemp — Creates a unique filename from a template.

Format

```
#include <stdlib.h>
char *mktemp (char *template);
```

Function Variants

The `mktemp` function has variants named `_mktemp32` and `_mktemp64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

template

A pointer to a buffer containing a user-defined template. You supply the template in the form, namXXXXXX. The six trailing Xs are replaced by a unique series of characters. You may supply

the first three characters. Because the template argument is overwritten, do not specify a string literal (`const` object).

Description

The use of `mktemp` is not recommended for new applications. See the `tmpnam` and `mkstemp` functions for the preferable alternatives.

Return Value

x

A pointer to the template, with the template modified to contain the created filename. If this value is a pointer to a null string, it indicates that a unique filename cannot be created.

mktime

mktime — Converts a local-time structure to a time, in seconds, since the Epoch.

Format

```
#include <time.h>
time_t mktime (struct tm *timeptr);
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `mktime` function that is equivalent to the behavior before OpenVMS Version 7.0.

Argument

timeptr

A pointer to the local-time structure.

Description

The `mktime` function converts a local-time structure (`struct tm`) pointed to by *timeptr*, to a time in seconds since the Epoch (a `time_t` variable), in the same manner as the values returned by the `time` function.

The original values of the `tm_wday` and `tm_yday` components of the structure are ignored, and the original values of the other components are not restricted to the ranges defined in `<time.h>`. Upon successful completion, the `tm_wday` and `tm_yday` components of the structure are set appropriately, and the other components are set to represent the specified time, with their values forced to the normal range.

If the local time cannot be encoded, then `mktime` returns the value (`time_t`)(-1).

The `time_t` type is defined in the `<time.h>` header file as follows:

```
typedef unsigned long int time_t;
```

Local time-zone information is set as if `mktime` called `tzset`.

If the `tm_isdst` field in the local-time structure pointed to by *timeptr* is positive, `mktime` initially presumes that Daylight Savings Time (DST) is in effect for the specified time.

If `tm_isdst` is 0, `mktime` initially presumes that DST is not in effect.

If `tm_isdst` is negative, `mktime` attempts to determine whether or not DST is in effect for the specified time.

Return Values

x

The specified calendar time encoded as a value of type `time_t`.

(time_t)(-1)

If the local time cannot be encoded.

Be aware that a return value of (`time_t`)(-1) can also represent the valid date: Sun Feb 7 06:28:15 2106.)

mmap

mmap — Maps file system object into virtual memory. This function is reentrant.

Format

```
#include <types.h>
#include <mman.h>
void mmap (void *addr, size_t len, int prot, int flags, int filedes,
off_t off); (X/Open, POSIX-1)
void mmap (void *addr, size_t len, int prot, int flags, int filedes,
off_t off ...); (VSI C Extension)
```

Function Variants

The `mmap` function has variants named `_mmap32` and `_mmap64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

addr

The starting address of the new region (must be the page boundary).

len

The length, in bytes, of the new region.

prot

Access permission, as defined in the <mman.h> header file. Specify either PROT_NONE, PROT_READ, or PROT_WRITE.

flags

Attributes of the mapped region as the results of a bitwise-inclusive OR operation on any combination of the following:

- MAP_FILE or MAP_ANONYMOUS
- MAP_VARIABLE or MAP_FIXED
- MAP_SHARED or MAP_PRIVATE

filedes

The file that you want to map to the new mapped file region returned by the `open` function.

off

The offset, specified in bytes. The `off_t` data type is either a 64-bit or 32-bit integer. The 64-bit interface allows for file sizes greater than 2 GB, and can be selected at compile time by defining the _LARGEFILE feature-test macro as follows:

```
CC/DEFINE=_LARGEFILE
```

...

An optional integer specifying additional flags for the SYS$CRMPSC system service for MAP_SHARED. This optional argument (VSI C Extension) of the `mmap` function was introduced in OpenVMS Version 7.2.

Description

The `mmap` function creates a new mapped file region, a new private region, or a new shared memory region.

Your application must ensure correct synchronization when using `mmap` in conjunction with any other file access method, such as `read` and `write`, and standard input/output.

Before calling `mmap`, the calling application must also ensure that all bytes in the range [*off*, *off*+ *len*] are written to the file (using the `fsync` function, for example). If this requirement is not met, `mmap` fails with `errno` set to ENXIO (No such device or address).

The *addr* and *len* arguments specify the requested starting address and length, in bytes, for the new region. The address is a multiple of the page size returned by `sysconf(_SC_PAGE_SIZE)`.

If the *len* argument is not a multiple of the page size returned by `sysconf(_SC_PAGE_SIZE)`, then the result of any reference to an address between the end of the region and the end of the page containing the end of the region is undefined.

The *flags* argument specifies attributes of the mapped region. Values for *flags* are constructed by a bitwise-inclusive OR operation on the flags from the following list of symbolic names defined in the <mman.h> header file:

MAP_FILE	Create a mapped file region.

MAP_ANONYMOUS	Create an unnamed memory region.
MAP_VARIABLE	Place region at the computed address.
MAP_FIXED	Place region at fixed address.
MAP_SHARED	Share changes.
MAP_PRIVATE	Changes are private.

The MAP_FILE and MAP_ANONYMOUS flags control whether the region you want to map is a mapped file region or an anonymous shared memory region. One of these flags must be selected.

If MAP_FILE is set in the *flags* argument:

- A new mapped file region is created, mapping the file associated with the *filedes* argument.
- The *off* argument specifies the file byte offset where the mapping starts. This offset must be a multiple of the page size returned by `sysconf(_SC_PAGE_SIZE)`.
- If the end of the mapped file region is beyond the end of the file, the result of any reference to an address in the mapped file region corresponding to an offset beyond the end of the file is unspecified.

If MAP_ANONYMOUS is set in the *flags* argument:

- A new memory region is created and initialized to all zeros.
- The *filedes* argument is ignored.

The new region is placed at the requested address if the requested address is not null and it is possible to place the region at this address. When the requested address is null or the region cannot be placed at the requested address, the MAP_VARIABLE and MAP_FIXED flags control the placement of the region. One of these flags must be selected.

If MAP_VARIABLE is set in the *flags* argument:

- If the requested address is null or if it is not possible for the system to place the region at the requested address, the region is placed at an address selected by the system.

If MAP_FIXED is set in the *flags* argument:

- If the requested address is not null, the `mmap` function succeeds even if the requested address is already part of another region. (If the address is within an existing region, the effect on the pages within that region and within the area of the overlap produced by the two regions is the same as if they were unmapped. In other words, whatever is mapped between *addr* and *addr* + *len* is unmapped.)
- If the requested address is null and MAP_FIXED is specified, the results are undefined.

The MAP_PRIVATE and MAP_SHARED flags control the visibility of modifications to the mapped file or shared memory region. One of these flags must be selected.

If MAP_SHARED is set in the *flags* argument:

- If the region is a mapped region, modifications to the region are visible to other processes that mapped the same region using MAP_SHARED.

- If the region is a mapped file region, modifications to the region are written to the file. (Note that the modifications are not immediately written to the file because of buffer cache delay; that is, the write to the file does not occur until there is a need to reuse the buffer cache. If the modifications must be written to the file immediately, use the `msync` function to ensure that this is done.)

If MAP_PRIVATE is set in the *flags* argument:

- Modifications to the mapped region by the calling process are not visible to other processes that mapped the same region using either MAP_PRIVATE or MAP_SHARED.
- Modifications to the mapped region by the calling process are not written to the file.

It is unspecified whether modifications by processes that mapped the region using MAP_SHARED are visible to other processes that mapped the same region using MAP_PRIVATE.

The *prot* argument specifies access permissions for the mapped region. Specify one of the following:

PROT_NONE	No access
PROT_READ	Read-only
PROT_WRITE	Read/Write access

After the successful completion of the `mmap` function, you can close the *filedes* argument without effect on the mapped region or on the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, that prevents the file data from being deallocated.

Note

The following rules apply to OpenVMS specific file references:

- Because of the additional file reference, if *filedes* is not opened for file sharing, `mmap` reopens it with file sharing enabled.
- The additional file reference that remains for mapped regions implies that a later `open`, `fopen`, or `create` call to the file that is mapped must specify file sharing.

Modifications made to the file using the `write` function are visible to mapped regions, and modifications to a mapped region are visible with the `read` function.

Note

Beginning with OpenVMS Version 7.2, while processing a MAP_SHARED request, the `mmap` function constructs the *flags* argument of the SYS$CRMPSC service as a bitwise inclusive OR of those bits it sets by itself to fulfill the MAP_SHARED request and those bits specified by the caller in the optional argument.

By default, for MAP_SHARED the `mmap` function creates a temporary group global section. The optional `mmap` argument provides the caller with direct access to the features of the SYS$CRMPSC system service.

Using the optional argument, the caller can create, for example, a system global section (SEC$M_SYSGBL bit) or permanent global section (SEC$M_PERM bit). For example, to create a

system permanent global section, the caller can specify (SEC$M_SYSGBL | SEC$M_PERM) in the optional argument.

The `mmap` function does not check or set any privileges. It is the responsibility of the caller to set appropriate privileges, such as SYSGBL privilege for SEC$M_SYSGBL, and PRMGBL for SEC$M_PERM, before calling `mmap` with the optional argument.

See also `read`, `write`, `open`, `fopen`, `creat`, and `sysconf`.

Return Values

x

The address where the mapping is placed.

MAP_FAILED

Indicates an error; `errno` is set to one of the following values:

- EACCES– The file referred to by *filedes* is not open for read access, or the file is not open for write access and PROT_WRITE was set for a MAP_SHARED mapping operation.
- EBADF– The *filedes* argument is not a valid file descriptor.
- EINVAL–The *flags* or *prot* argument is invalid, or the *addr* argument or *off* argument is not a multiple of the page size returned by `sysconf(_SC_PAGE_SIZE)`.
- ENODEV– The file descriptor *filedes* refers to an object that cannot be mapped, such as a terminal.
- ENOMEM– There is not enough address space to map *len* bytes.
- ENXIO– The addresses specified by the range [*off, off* + *len*] are invalid for *filedes*.
- EFAULT– The *addr* argument is an invalid address.

modf

modf — Decomposes a floating-point number.

Format

```
#include <math.h>
double modf (double x, double *iptr);
float modff (float x, float *iptr);
long double modfl (long double x, long double *iptr);
```

Arguments

x

An object of type `double`, `float`, or `long double`.

iptr

A pointer to an object of type `double`, `float`, or `long double` to match the type of *x*.

Description

The `modf` functions decompose their first argument *x* into a positive fractional part *f* and an integer part *i*, each of which has the same sign as *x*.

The functions return *f* and assign *i* to the object pointed to by the second argument (*iptr*).

Return Values

x

The fractional part of the argument *x*.

NaN

x is NaN; `errno` is set to EDOM and * *iptr* is set to NaN.

0

Underflow occurred; `errno` is set to ERANGE.

[w]move

[w]move — Change the current cursor position on the specified window to the coordinates (*y*, *x*). The `move` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int move (int y, int x);
int wmove (WINDOW *win, int y, int x);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

Description

For more information, see the `scrollok` function in this section.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the screen scroll illegally.

mprotect

mprotect — Modifies access protections of memory mapping. This function is reentrant.

Format

```
#include <mman.h>
int mprotect (void *addr, size_t len, int prot);
```

Arguments

addr

The address of the region that you want to modify.

len

The length, in bytes, of the region that you want to modify.

prot

Access permission, as defined in the `<mman.h>` header file. Specify either PROT_NONE, PROT_READ, or PROT_WRITE.

Description

The `mprotect` function modifies the access protection of a mapped file or shared memory region.

The *addr* and *len* arguments specify the address and length, in bytes, of the region that you want to modify. The *len* argument must be a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`. If *len* is not a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`, the length of the region is rounded up to the next multiple of the page size.

The *prot* argument specifies access permissions for the mapped region. Specify one of the following:

PROT_NONE	No access
PROT_READ	Read-only
PROT_WRITE	Read/Write access

The `mprotect` function does not modify the access permission of any region that lies outside of the specified region, except that the effect on addresses between the end of the region, and the end of the page containing the end of the region, is unspecified.

If the `mprotect` function fails under a condition other than that specified by EINVAL, the access protection of some of the pages in the range [*addr, addr* + *len*] can change. Suppose the error occurs on some page at an *addr2*; `mprotect` can modify protections of all whole pages in the range [*addr, addr2*].

See also `sysconf`.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to one of the following values:

- EACCESS– The *prot* argument specifies a protection that conflicts with the access permission set for the underlying file.
- EINVAL– The *prot* argument is invalid, or the *addr* argument is not a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`.
- EFAULT– The range [*addr, addr* + *len*] includes an invalid address.

mrand48

mrand48 — Generates uniformly distributed pseudorandom-number sequences. Returns 48-bit signed long integers.

Format

```
#include <stdlib.h>
long int mrand48 (void);
```

Description

The `mrand48` function generates pseudorandom numbers using the linear congruential algorithm and 48-bit integer arithmetic.

It returns signed long integers uniformly distributed over the range of *y* values such that $-2^{31} \leq y < 2^{31}$.

Before you call the `mrand48` function, use either `srand48`, `seed48`, or `lcong48` to initialize the random-number generator. You must initialize the `mrand48` function prior to invoking it, because it stores the last 48-bit X*i* generated into an internal buffer. (Although it is not recommended, constant default initializer values are supplied automatically if the `drand48`, `lrand48`, or `mrand48` functions are called without first calling an initialization function.)

The function works by generating a sequence of 48-bit integer values, X*i*, according to the linear congruential formula:

```
Xn+1 = (aXn+c)mod m        n >= 0
```

The argument *m* equals 2^{48}, so 48-bit integer arithmetic is performed. Unless you invoke the `lcong48` function, the multiplier value *a* and the addend value *c*are:

```
a = 5DEECE66D16 = 2736731631558
c = B16 = 138
```

The values returned by the `mrand48` function is computed by first generating the next 48-bit X*i* in the sequence. Then the appropriate bits, according to the type of returned data item, are copied from the high-order (most significant) bits of X*i* and transformed into the returned value.

See also `drand48`, `lrand48`, `lcong48`, `seed48`, and `srand48`.

Return Value

n

Returns signed long integers uniformly distributed over the range $-2^{31} \leq y < 2^{31}$.

msync

msync — Synchronizes a mapped file.

Format

```
#include <mman.h>
int msync (void *addr, size_t len, int flags);
```

Arguments

addr

The address of the region that you want to synchronize.

len

The length, in bytes, of the region that you want to synchronize.

flags

One of the following symbolic constants defined in the `<mman.h>` header file:

MS_SYNC	Synchronous cache flush
MS_ASYNC	Asynchronous cache flush
MS_INVALIDATE	Invalidate cashed pages

Description

The `msync` function controls the caching operations of a mapped file region. Use `msync` to:

- Ensure that modified pages in the region transfer to the underlying storage device of the file.
- Control the visibility of modifications with respect to file system operations.

The *addr* and *len* arguments specify the region to be synchronized. The *len* argument must be a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`; otherwise, the length of the region is rounded up to the next multiple of the page size.

If the *flags* argument is set to:

flags Argument	Then the msync Function...
MS_SYNC	Does not return until the system completes all I/O operations.
MS_ASYNC	Returns after the system schedules all I/O operations.
MS_INVALIDATE	Invalidates all cached copies of the pages. The operating system must obtain new copies of the pages from the file system the next time the application references them.

After a successful call to the `msync` function with the flags argument set to:

- `MS_SYNC`– All previous modifications to the mapped region are visible to processes using the `read` argument. Previous modifications to the file using the `write` function are lost.
- `MS_INVALIDATE`– All previous modifications to the file using the `write` function are visible to the mapped region. Previous direct modifications to the mapped region are lost.

See also `read`, `write`, and `sysconf`.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to one of the following values:

- EIO– An I/O error occurred while reading from or writing to the file system.
- ENOMEM– The range specified by [*addr, addr* + *len*] is invalid for a process's address space, or the range specifies one or more unmapped pages.
- EINVAL– The *addr* argument is not a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`.
- EFAULT– The range [*addr, addr* + *len*] includes an invalid address.

munmap

munmap — Unmaps a mapped region. This function is reentrant.

Format

```
#include <mman.h>
int munmap (void *addr, size_t len);
```

Arguments

addr

The address of the region that you want to unmap.

len

The length, in bytes, of that region the you want to unmap.

Description

The `munmap` function unmaps a mapped file or shared memory region.

The *addr* and *len* arguments specify the address and length, in bytes, respectively, of the region to be unmapped.

The *len* argument must be a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`; otherwise, the length of the region is rounded up to the next multiple of the page size.

The result of using an address that lies in an unmapped region and not in any subsequently mapped region is undefined.

See also `sysconf`.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to one of the following values:

- ENIVAL– The *addr* argument is not a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`.
- EFAULT– The range [*addr*, *addr* + *len*] includes an invalid address.

mv[w]addch

mv[w]addch — Move the cursor to coordinates (*y*, *x*) and add a character to the specified window.

Format

```
#include <curses.h>
int mvaddch (int y, int x, char ch);
int mvwaddch (WINDOW *win, int y, int x, char ch);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

ch

If this argument is a new-line character (\n), the `mvaddch` and `mvwaddch` functions clear the line to the end, and move the cursor to the next line at the same *x* coordinate. A carriage return (\r) moves the cursor to the beginning of the specified line. A tab (\t) moves the cursor to the next tabstop within the window.

Description

This routine performs the same function as `mvwaddch`, but on the `stdscr` window.

When `mvwaddch` is used on a subwindow, it writes the character onto the underlying window as well.

Return Values

OK

Indicates success.

ERR

Indicates that writing the character would cause the screen to scroll illegally. For more information, see the `scrollok` function.

mv[w]addstr

mv[w]addstr — Move the cursor to coordinates (*y*, *x*) and add the specified string, to which *str* points, to the specified window.

Format

```
#include <curses.h>
int mvaddstr (int y, int x, char *str);
int mvwaddstr (WINDOW *win, int y, int x, char *str);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

str

A pointer to the character string.

Description

This routine performs the same function as `mvwaddstr`, but on the `stdscr` window.

When `mvwaddstr` is used on a subwindow, the string is written onto the underlying window as well.

Return Values

OK

Indicates success.

ERR

Indicates that the function causes the screen to scroll illegally, but it places as much of the string onto the window as possible. For more information, see the `scrollok` function.

mvcur

mvcur — Moves the terminal's cursor from (*lasty*, *lastx*) to (*newy*, *newx*).

Format

```
#include <curses.h>
int mvcur (int lasty, int lastx, int newy, int newx);
```

Arguments

lasty

The cursor position.

lastx

The cursor position.

newy

The resulting cursor position.

newx

The resulting cursor position.

Description

In VSI C for OpenVMS systems, `mvcur` and `move` perform the same function.

See also `move`.

Return Values

OK

Indicates success.

ERR

Indicates that moving the window placed part or all of the window off the edge of the terminal screen. The terminal screen remains unaltered.

mv[w]delch

mv[w]delch — Move the cursor to coordinates (*y*, *x*) and delete the character on the specified window. The `mvdelch` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int mvdelch (int y, int x);
int mvwdelch (WINDOW *win, int y, int x);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

Description

Each of the following characters on the same line shifts to the left, and the last character becomes blank.

Return Values

OK

Indicates success.

ERR

Indicates that deleting the character would cause the screen to scroll illegally. For more information, see the `scrollok` function.

mv[w]getch

mv[w]getch — Move the cursor to coordinates (*y*, *x*), get a character from the terminal screen, and echo it on the specified window. The `mvgetch` function acts on the `stdscr` window.

Format

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

```
int mvgetch (int y, int x);
int mvwgetch (WINDOW *win, int y, int x);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

Description

The `mvgetch` and `mvwgetch` functions refresh the specified window before fetching the character.

Return Values

x

The returned character.

ERR

Indicates that the function causes the screen to scroll illegally. For more information, see the `scrollok` function in this section.

mv[w]getstr

mv[w]getstr — Move the cursor to coordinates (*y*, *x*), get a string from the terminal screen, store it in the variable *str* (which must be large enough to contain the string), and echo it on the specified window. The `mvgetstr` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int mvgetstr (int y, int x, char *str);
int mvwgetstr (WINDOW *win, int y, int x, char *str);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

str

The string that is displayed.

Description

The `mvgetstr` and `mvwgetstr` functions strip the new-line terminator (\n) from the string.

Return Values

OK

Indicates success.

ERR

Indicates that the function causes the screen to scroll illegally.

mv[w]inch

mv[w]inch — Move the cursor to coordinates (*y*, *x*) and return the character on the specified window without making changes to the window. The `mvinch` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int mvinch (int y, int x);
int mvwinch (WINDOW *win, int y, int x);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

Return Values

x

The returned character.

ERR

Indicates an input error.

mv[w]insch

mv[w]insch — Move the cursor to coordinates (*y*, *x*) and insert the character *ch* into the specified window. The `mvinsch` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int mvinsch (int y, int x, char ch);
int mvwinsch (WINDOW *win, int y, int x, char ch);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

ch

The character to be inserted at the window's coordinates.

Description

After the character is inserted, each character on the line shifts to the right, and the last character on the line is deleted.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the screen scroll illegally. For more information, see the `scrollok` function in this section.

mv[w]insstr

mv[w]insstr — Move the cursor to coordinates (*y*, *x*) and insert the specified string into the specified window. The `mvinsstr` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int mvinsstr (int y, int x, char *str);
int mvwinsstr (WINDOW *win, int y, int x, char *str);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

str

The string that is displayed.

Description

Each character after the string shifts to the right, and the last character disappears. The `mvinsstr` and `mvwinsstr` functions are specific to VSI C for OpenVMS systems and are not portable.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the screen scroll illegally. For more information, see the `scrollok` function.

mvwin

mvwin — Moves the starting position of the window to the specified (*y*, *x*) coordinates.

Format

```
#include <curses.h>
mvwin (WINDOW *win, int y, int x);
```

Arguments

win

A pointer to the window.

y

A window coordinate.

x

A window coordinate.

Description

When moving subwindows, the `mvwin` function does not rewrite the contents of the subwindow on the underlying window at the new position. If you write anything to the subwindow after the move, the function also writes to the underlying window.

Return Values

OK

Indicates success.

ERR

Indicates that moving the window put part or all of the window off the edge of the terminal screen. The terminal screen remains unaltered.

nanosleep

nanosleep — High-resolution sleep (REALTIME). Suspends a process (or thread in a threaded program) from execution for the specified timer interval.

Format

```
#include <time.h>
int nanosleep (const struct timespec *rqtp, struct timespec *rmtp);
```

Arguments

rqtp

A pointer to the `timespec` data structure that defines the time interval during which the calling process or thread is suspended.

rmtp

A pointer to the `timespec` data structure that receives the amount of time remaining in the previously requested interval, or zero if the full interval has elapsed.

Description

The `nanosleep` function suspends a process or thread until one of the following conditions is met:

- The time interval specified by the *rqtp* argument has elapsed.

- A signal is delivered to the calling process and the action is to invoke a signal-catching function or to terminate the process.

The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. Except when interrupted by a signal, the suspension time is not less than the time specified by the *rqtp* argument (as measured by the system clock, CLOCK_REALTIME).

The use of the `nanosleep` function has no effect on the action or blockage of any signal.

If the requested time has elapsed, the call was successful and the `nanosleep` function returns zero.

On failure, the `nanosleep` function returns -1 and sets `errno` to indicate the failure. The function fails if it has been interrupted by a signal, or if the *rqtp* argument specified a nanosecond value less than 0 or greater than or equal to 1 billion.

If the *rmtp* argument is non-NULL, the `timespec` structure it references is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept).

If the *rmtp* argument is NULL, the remaining time is not returned.

See also `clock_getres`, `clock_gettime`, `clock_settime`, and `sleep`.

Return Values

0

Indicates success. The requested time has elapsed.

-1

Indicates failure. The function call was unsuccessful or was interrupted by a signal; `errno` is set to one of the following values:

- EINTR– The `nanosleep` function was interrupted by a signal.
- EINVAL– The *rqtp* argument specified a nanosecond value less than 0 or greater than or equal to 1 billion.

nearbyint

nearbyint — Rounds to the nearest integer value in floating-point format, using the current rounding direction.

Format

```
#include <math.h>
double nearbyint (double x);
float nearbyintf (float x);
long double nearbyintl (long double x);
```

Argument

x

Value to round.

Description

The `nearbyint` functions round *x* to an integer value in floating-point format, using the current rounding direction and without raising the "inexact" floating-point exception.

Return Value

n

On success, the function returns the rounded integer value.

newwin

newwin — Creates a new window with *numlines* lines and *numcols* columns starting at the coordinates (*begin_y*, *begin_x*) on the terminal screen.

Format

```
#include <curses.h>
WINDOW *newwin (int numlines, int numcols, int begin_y, int begin_x);
```

Arguments

numlines

If it is 0, the `newwin` function sets that dimension to LINES (*begin_y*). To get a new window of dimensions LINES by COLS, use the following line:

```
newwin (0, 0, 0, 0)
```

numcols

If it is 0, the `newwin` function sets that dimension to COLS (*begin_x*). Therefore, to get a new window of dimensions LINES by COLS, use the following line:

```
newwin (0, 0, 0, 0)
```

begin_y

A window coordinate.

begin_x

A window coordinate.

Return Values

x

The address of the allocated window.

ERR

Indicates an error.

nextafter

nextafter — Returns the next machine-representable number following *x* in the direction of *y*.

Format

```
#include <math.h>
double nextafter (double x, double y);
float nextafterf (float x, float y);
long double nextafterl (long double x, long double y);
```

Arguments

x

A real number.

y

A real number.

Description

The `nextafter` functions return the next machine-representable floating-point number following *x* in the direction of *y*. If *y* is less than *x*, `nextafter` returns the largest representable floating-point number less than *x*.

Return Values

n

The next representable floating-point value following *x* in the direction of *y*.

HUGE_VAL

Overflow; `errno` is set to ERANGE.

NaN

x or *y* is NaN; `errno` is set to EDOM.

nexttoward

nexttoward — Equivalent to the `nextcafter` function, with exceptions noted in the Description.

Format

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

```
double nexttoward (double x, long double y);
float nexttowardf (float x, long double y);
long double nexttowardl (long double x, long double y);
```

Arguments

x

A real number.

y

A real number.

Description

The `nexttoward` functions are equivalent to the corresponding `nextafter` functions, except that the second parameter has type `long double` and the functions return *y* converted to the type of the function if *x* equals *y*.

Return Values

n

The next representable floating-point value following *x* in the direction of *y*.

***y* (of the type *x*)**

If *x* equals *y*.

HUGE_VAL

Overflow; `errno` is set to ERANGE.

NaN

x or *y* is NaN; `errno` is set to EDOM.

nice

nice — Increases or decreases process priority relative to the process current priority by the amount of the argument. This function is nonreentrant.

Format

```
#include <unistd.h>
int nice (int increment);
```

Argument

increment

As a positive argument, decreases priority; as a negative argument, increases priority. Issuing `nice(0)` restores the base priority. The resulting priority cannot be less than 1, or greater than the process's base priority. If it is, the `nice` function quietly does nothing.

Description

When a process calls the `vfork` function, the resulting child inherits the parent's priority.

With the DECC$ALLOW_UNPRIVILEGED_NICE feature logical enabled, the `nice` function exhibits its legacy behavior of not checking the privilege of the calling process (that is, any user may lower the `nice` value to increase process priorities). Also, when the caller sets a priority above MAX_PRIORITY, the `nice` value is set to the base priority.

With DECC$ALLOW_UNPRIVILEGED_NICE disabled, the `nice` function conforms to the X/Open standard of checking the privilege of the calling process (only users with ALTPRI privilege can lower the `nice` value to increase process priorities), and when the caller sets a priority above MAX_PRIORITY, the `nice` value is set to MAX_PRIORITY.

See also `vfork`.

Return Values

0

Indicates success.

-1

Indicates failure.

nint

nint — Returns the nearest integral value to the argument.

Format

```
#include <math.h>
double nint (double x);
float nintf (float x,);
long double nintl (long double x);
```

Argument

x

A real number.

Description

The `nint` functions return the nearest integral value to *x*, except halfway cases are rounded to the integral value larger in magnitude. This corresponds to the Fortran generic intrinsic function `nint`.

Return Values

n

The nearest integral value to *x*.

NaN

x is NaN; `errno` is set to EDOM.

[no]nl

[no]nl — The `nl` and `nonl` functions are provided only for UNIX software compatibility and have no function in the OpenVMS environment.

Format

```
#include <curses.h>
void nl (void);
void nonl (void);
```

nl_langinfo

nl_langinfo — Returns a pointer to a string that contains information obtained from the program's current locale.

Format

```
#include <langinfo.h>
char *nl_langinfo (nl_item item);
```

Argument

item

The name of a constant that specifies the information required. These constants are defined in `<langinfo.h>`.

The following constants are valid:

Constant	Category	Description
D_T_FMT	LC_TIME	String for formatting date and time
D_FMT	LC_TIME	String for formatting date
T_FMT	LC_TIME	String for formatting time
T_FMT_AMPM	LC_TIME	Time format with AM/PM string
AM_STR	LC_TIME	String that represents AM in 12-hour clock notation
PM_STR	LC_TIME	String that represents PM in 12-hour clock notation
DAY_1	LC_TIME	The name of the first day of the week

Constant	Category	Description
. . .		
DAY_7	LC_TIME	The name of the seventh day of the week
ABDAY_1	LC_TIME	The abbreviated name of the first day of the week
. . .		
ABDAY_7	LC_TIME	The abbreviated name of the seventh day of the week
MON_1	LC_TIME	The name of the first month in the year
. . .		
MON_12	LC_TIME	The name of the twelfth month in the year
ABMON_1	LC_TIME	The abbreviated name of the first month in the year
. . .		
ABMON_12	LC_TIME	The abbreviated name of the twelfth month in the year
ERA	LC_TIME	Era description strings
ERA_D_FMT	LC_TIME	Era date format string
ERA_T_FMT	LC_TIME	Era time format
ERA_D_T_FMT	LC_TIME	Era date and time format
ALT_DIGITS	LC_TIME	Alternative symbols for digits
RADIXCHAR	LC_NUMERIC	The radix character
THOUSEP	LC_NUMERIC	The character used to separate groups of digits in nonmonetary values
YESEXP	LC_MESSAGES	The expression for affirmative responses to yes/no questions
NOEXP	LC_MESSAGES	The expression for negative responses to yes/no questions
CRNCYSTR	LC_MONETARY	The currency symbol. It is preceded by one of the following: • A minus (-) if the symbol is to appear before the value • A plus (+) if the symbol is to appear after the value • A period (.) if the symbol replaces the radix character
CODESET	LC_CTYPE	Codeset name

Description

If the current locale does not have language information defined, the function returns information from the C locale. The program should not modify the string returned by the function. This string might be overwritten by subsequent calls to `nl_langinfo`.

If the `setlocale` function is called after a call to `nl_langinfo`, then the pointer returned by the previous call to `nl_langinfo` will be unspecified. In this case, the `nl_langinfo` function should be called again.

Return Value

x

Pointer to the string containing the requested information. If *item* is invalid, the function returns an empty string.

Example

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

/* This test sets up the British English locale, and then       */
/* inquires on the data and time format, first day of the week, */
/* and abbreviated first day of the week.                       */

#include <stdlib.h>
#include <string.h>

int main()
{
    char *return_val;
    char *nl_ptr;

    /* set the locale, with user supplied locale name  */

    return_val = setlocale(LC_ALL, "en_gb.iso8859-1");
    if (return_val == NULL) {
        printf("ERROR : The locale is unknown");
        exit(1);
    }
    printf("+--------------------------------------+\n");

    /* Get the date and time format from the locale.  */

    printf("D_T_FMT = ");

    /*  Compare the returned string from nl_langinfo with */
    /*  an empty string.                                  */

    if (!strcmp((nl_ptr = (char *) nl_langinfo(D_T_FMT)), "")) {

  /* The string returned was empty this could mean that either */
  /* 1) The locale does not contain a value for this item      */
  /* 2) The value for this item is an empty string             */

        printf("nl_langinfo returned an empty string\n");
    }
    else {
        /* Display the date and time format  */
```

```
        printf("%s\n", nl_ptr);
    }

/* Get the full name for the first day of the week from locale */
   printf("DAY_1 = ");

  /*  Compare the returned string from nl_langinfo with */
  /*  an empty string.                                  */

    if (!strcmp((nl_ptr = (char *) nl_langinfo(DAY_1)), "")) {

  /* The string returned was empty this could mean that either */
  /*    1) The locale does not contain a value for the first   */
  /*       day of the week                                     */
  /*    2) The value for the first day of the week is          */
  /*       an empty string                                     */

        printf("nl_langinfo returned an empty string\n");
    }

    else {
     /* Display the full name of the first day of the week     */

        printf("%s\n", nl_ptr);
    }
/* Get the abbreviated name for the first day of the week
                                                 from locale  */

    printf("ABDAY_1 = ");

 /* Compare the returned string from nl_langinfo with an empty */
 /* string.                                                    */

    if (!strcmp((nl_ptr = (char *) nl_langinfo(ABDAY_1)), "")) {

 /* The string returned was empty this could mean that either  */
 /*    1) The locale does not contain a value for the first    */
 /*       day of the week                                      */
 /*    2) The value for the first day of the week is an        */
 /*       empty string                                         */

        printf("nl_langinfo returned an empty string\n");
    }

    else {

 /* Display the abbreviated name of the first day of the week  */

        printf("%s\n", nl_ptr);
   }
}
```

Running the example program produces the following result:

```
+---------------------------------------+
D_T_FMT = %a %e %b %H:%M:%S %Y
DAY_1 = Sunday
ABDAY_1 = Sun
```

nrand48

nrand48 — Generates uniformly distributed pseudorandom-number sequences. Returns 48-bit signed long integers.

Format

```
#include <stdlib.h>
long int nrand48 (unsigned short int xsubi[3]);
```

Argument

xsubi

An array of three `short int`s that, when concatenated together, form a 48-bit integer.

Description

The `nrand48` function generates pseudorandom numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The `nrand48` function returns nonnegative, long integers uniformly distributed over the range of *y* values, such that $0 \le y < 2^{31}$.

The function works by generating a sequence of 48-bit integer values, X*i*, according to the linear congruential formula:

```
Xn+1 = (aXn+c)mod m        n >= 0
```

The argument *m* equals 2^{48}, so 48-bit integer arithmetic is performed. Unless you invoke the `lcong48` function, the multiplier value *a* and the addend value *c* are:

```
a = 5DEECE66D16 = 2736731631558
c = B16 = 138
```

The `nrand48` function requires that the calling program pass an array as the *xsubi* argument, which for the first call must be initialized to the initial value of the pseudorandom-number sequence. Unlike the `drand48` function, it is not necessary to call an initialization function prior to the first call.

By using different arguments, the `nrand48` function allows separate modules of a large program to generate several independent sequences of pseudorandom numbers. For example, the sequence of numbers that one module generates does not depend upon how many times the functions are called by other modules.

Return Value

n

Returns nonnegative, long integers over the range $0 \le y < 2^{31}$.

open

open — Opens a file for reading, writing, or editing. It positions the file at its beginning (byte 0).

Format

```
#include <fcntl.h>
int open (const char *file_spec, int flags, mode_t mode); (ANSI C)
int open (const char *file_spec, int flags, ...); (VSI C Extension)
```

Arguments

file_spec

A null-terminated character string containing a valid file specification. If you specify a directory in the *file_spec* and it is a search list that contains an error, VSI C interprets it as a file open error.

If the *file_spec* parameter refers to a symbolic link, the `open` function opens the file pointed to by the symbolic link.

flags

The following values are defined in the `<fcntl.h>` header file:

`O_RDONLY`	Opens a file for reading only.
`O_WRONLY`	Opens a file for writing only.
`O_RDWR`	Opens a file for both reading and writing.
`O_NDELAY`	Opens a file for asynchronous input.
`O_APPEND`	Moves the file pointer to the end of the file before every write operation.
`O_CREAT`	Creates a file if it does not exist.
`O_TRUNC`	Creates a new version of this file.
`O_EXCL`	Returns an error if a file specified by *filename* exists. Applies only when used with `O_CREAT`.
`O_CLOEXEC`	Sets the `FD_CLOEXEC` flag on the file descriptor.

These flags are set using the bitwise OR operator (|) to separate specified flags.

Opening a file with `O_APPEND` causes each write on the file to be appended to the end. In contrast, with the VAX C RTL the behavior of files opened in append mode was to start at EOF and, thereafter, write at the current file position.

If `O_TRUNC` is specified and the file exists, `open` creates a new file by incrementing the version number by 1, leaving the old version in existence.

If `O_CREAT` is set and the named file does not exist, the C RTL creates it with any attributes specified in the fourth and subsequent arguments (. . .). If `O_EXCL` is set with `O_CREAT` and the named file exists, the attempted open returns an error.

mode

An unsigned value that specifies the file-protection mode. The compiler performs a bitwise AND operation on the mode and the complement of the current protection mode.

You can construct modes by using the bitwise OR operator (|) to separate specified modes. The modes are:

0400	OWNER:READ
0200	OWNER:WRITE

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

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

...

Optional file attribute arguments. The file attribute arguments are the same as those used in the `creat` function. For more information, see the `creat` function.

Description

If a version of the file exists, a new file created with `open` inherits certain attributes from the existing file unless those attributes are specified in the `open` call. The following attributes are inherited: record format, maximum record size, carriage control, and file protection.

Notes

- If you intend to do random writing to a file, the file must be opened for update by specifying a *flags* value of `O_RDWR`.
- To create files with OpenVMS RMS default protections by using the UNIX system-call functions `umask`, `mkdir`, `creat`, and `open`, call `mkdir`, `creat`, and `open` with a file-protection mode argument of 0777 in a program that never specifically calls `umask`. These default protections include correctly establishing protections based on ACLs, previous versions of files, and so on.

  In programs that do `vfork`/ `exec` calls, the new process image inherits whether `umask` has ever been called or not from the calling process image. The `umask` setting and whether the `umask` function has ever been called are both inherited attributes.

See also `creat`, `read`, `write`, `close`, `dup`, `dup2`, and `lseek`.

Return Values

x

A nonnegative file descriptor number.

-1

Indicates that the file does not exist, that it is protected against reading or writing, or that it cannot be opened for another reason.

Example

```
#include <unixio.h>
```

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

main()
{
    int file,
        stat;
    int flags;

    flags = O_RDWR;  /*  Open for read and write,            */
                     /*  with user default file protection,  */
                     /*  with max fixed record size of 2048, */
                     /*  and a block size of 2048 bytes.     */

file=open("file.dat", flags, 0, "rfm=fix", "mrs=2048", "bls=2048");
    if (file == -1)
        perror("OPEN error"), exit(1);

    close(file);
}
```

opendir

opendir — Opens a specified directory.

Format

```
#include <dirent.h>
DIR *opendir (const char *dir_name);
```

Argument

dir_name

The name of the directory to be opened.

Description

The `opendir` function opens the directory specified by *dir_name* and associates a directory stream with it. The *dir_name* argument can be specified in OpenVMS style or UNIX style. The directory stream is positioned at the first entry. The type `DIR`, defined in the `<dirent.h>` header file, represents a directory stream. A directory stream is an ordered sequence of all the directory entries in a particular directory.

The `opendir` function also returns a pointer to identify the directory stream in subsequent operations. The NULL pointer is returned when the directory named by *dir_name* cannot be accessed, or when not enough memory is available to hold the entire stream.

Notes

- An open directory must always be closed with the `closedir` function to ensure that the next attempt to open that directory is successful. The `opendir` function should be used with `readdir`, `closedir`, and `rewinddir` to examine the contents of the directory.

- The `opendir` function supports UNIX style path name specifications. For more information about UNIX style directory specifications, see Section 1.3.3.

Example

See the program example in the description of `closedir`.

Return Values

x

A pointer to an object of type `DIR`.

NULL

Indicates an error; `errno` is set to one of the following values:

- EACCES– Search permission is denied for any component of *dir_name* or read permission is denied for *dir_name*.
- ENAMETOOLONG– The length of the *dir_name* string exceeds PATH_MAX, or a pathname component is longer than

  NAME_MAX.
- ENOENT– The *dir_name* argument points to the name of a file that does not exist, or is an empty string.

overlay

overlay — Nondestructively superimposes *win1* on *win2*. The function writes the contents of *win1* that will fit onto *win2* beginning at the starting coordinates of both windows. Blanks on *win1* leave the contents of the corresponding space on *win2* unaltered. The `overlay` function copies as much of a window's box as possible.

Format

```
#include <curses.h>
int overlay (WINDOW *win1, WINDOW *win2);
```

Arguments

win1

A pointer to the window.

win2

A pointer to the window.

Return Values

OK

Indicates success.

ERR

Indicates an error.

overwrite

overwrite — Destructively writes the contents of *win1* on *win2*.

Format

```
#include <curses.h>
int overwrite (WINDOW *win1, WINDOW *win2);
```

Arguments

win1

A pointer to the window.

win2

A pointer to the window.

Description

The `overwrite` function writes the contents of *win1* that will fit onto *win2* beginning at the starting coordinates of both windows. Blanks on *win1* are written on *win2* as blanks. This function copies as much of a window's box as possible.

Return Values

OK

Indicates success.

ERR

Indicates failure.

pathconf

pathconf — Retrieves file implementation characteristics.

Format

```
#include <unistd.h>
long int pathconf (const char *path, int name);
```

Arguments

path

The pathname of a file or directory.

name

The configuration attribute to query. If this attribute is not applicable to the file specified by the *path* argument, the `pathconf` function returns an error.

Description

The `pathconf` function allows an application to determine the characteristics of operations supported by the file system underlying the file named by *path*. Read, write, or execute permission of the named file is not required, but you must be able to search all directories in the path leading to the file.

Symbolic values for the *name* argument are defined in the `<unistd.h>` header file, as follows:

_PC_LINK_MAX	The maximum number of links to the file. If the *path* argument refers to a directory, the value returned applies to the directory itself.
_PC_MAX_CANON	The maximum number of bytes in a canonical input line. This is applicable only to terminal devices.
_PC_MAX_INPUT	The number of types allowed in an input queue. This is applicable only to terminal devices.
_PC_NAME_MAX	Maximum number of bytes in a filename (not including a terminating null). The byte range value is between 13 and 255. This is applicable only to a directory file. The value returned applies to filenames within the directory.
_PC_PATH_MAX	Maximum number of bytes in a pathname (not including a terminating null). The value is never larger than 65,535. This is applicable only to a directory file. The value returned is the maximum length of a relative pathname when the specified directory is the working directory.
_PC_PIPE_BUF	Maximum number of bytes guaranteed to be written atomically. This is applicable only to a FIFO. The value returned applies to the referenced object. If the *path* argument refers to a directory, the value returned applies to any FIFO that exists or can be created within the directory.
_PC_CHOWN_RESTRICTED	This is applicable only to a directory file. The value returned applies to any files (other than directories) that exist or can be created within the directory.
_PC_NO_TRUNC	Returns 1 if supplying a component name longer than allowed by NAME_MAX causes an error. Returns 0 (zero) if long component names are truncated. This is applicable only to a directory file.
_PC_VDISABLE	This is always 0 (zero); no disabling character is defined. This is applicable only to a terminal device.

Return Values

x

Resultant value of the configuration attribute specified in *name*.

-1

Indicates an error; `errno` is set to one of the following values:

- EACCES– Search permission is denied for a component of the path prefix.
- EINVAL– The *name* argument specifies an unknown or inapplicable characteristic.
- EFAULT– The *path* argument is an invalid address.
- ENAMETOOLONG– The length of the *path* string exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
- ENOENT– The named file does not exist or the *path* argument points to an empty string.
- ENOTDI– A component of the *path* prefix is not a directory.

pause

pause — Suspends the calling process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

Format

```
#include <unistd.h>
int pause (void);
```

Description

The `pause` function suspends the calling process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, `pause` does not return.

If the action is to execute a signal-catching function, `pause` returns after the signal-catching function returns.

Return Value

Since the `pause` function suspends process execution indefinitely unless interrupted by a signal, there is no successful completion return value.

In cases where `pause` returns, the return value is -1, and `errno` is set to EINTR.

pclose

pclose — Closes a pipe to a process.

Format

```
#include <stdio.h>
int pclose (FILE *stream);
```

Arguments

stream

A pointer to a `FILE` structure for an open pipe returned by a previous call to the `popen` function.

Description

The `pclose` function closes a pipe between the calling program and a shell command to be executed. Use `pclose` to close any stream you have opened with `popen`. The `pclose` function waits for the associated process to end, and then returns the exit status of the command. See the description of `waitpid` for information on interpreting the exit status.

Beginning with OpenVMS Version 7.3-1, when compiled with the _VMS_WAIT macro defined, the `pclose` function returns the OpenVMS completion code of the child process.

See also `popen`.

Return Values

x

Exit status of child.

-1

Indicates an error. The *stream* argument is not associated with a `popen` function. `errno` is set to the following:

- ECHILD – cannot obtain the status of the child process.

perror

perror — Writes a short error message to `stderr` describing the current value of `errno`.

Format

```
#include <stdio.h>
void perror (const char *str);
```

Argument

str

Usually the name of the program that caused the error.

Description

The `perror` function uses the error number in the external variable `errno` to retrieve the appropriate locale-dependent error message. The function writes out the message as follows: *str* (a user-supplied prefix to the error message), followed by a colon and a space, followed by the message itself, followed by a new-line character.

See the description of `errno` in Chapter 4 for a list of possible errors.

See also `strerror`.

Example

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

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fp;

    fp = fopen(argv[1], "r");   /* Open an input file. */
    if (fp == NULL) {

        /* If the fopen call failed, perror prints out a        */
        /* diagnostic:                                          */
        /*                                                      */
        /*  "open: <error message>"                             */
        /*  This error message provides a diagnostic explaining */
        /*  the cause of the failure.                           */

        perror("open");
        exit(EXIT_FAILURE);
    }
    else
        fclose(fd) ;
}
```

pipe

pipe — Creates a temporary mailbox that can be used to read and write data between a parent and child process. The channels through which the processes communicate are called a *pipe*.

Format

```
#include <unistd.h>
int pipe (int array_fdscptr[2]); (ISO POSIX-1)
int pipe (int array_fdscptr[2], ...); (VSI C Extension)
```

Arguments

array_fdscptr

An array of file descriptors. A pipe is implemented as an array of file descriptors associated with a mailbox. These mailbox descriptors are special in that these are the only file descriptors which, when passed to the `isapipe` function, will return 1.

The file descriptors are allocated in the following way:

- The first available file descriptor is assigned to writing, and the next available file descriptor is assigned to reading.

- The file descriptors are then placed in the array in reverse order; element 0 contains the file descriptor for reading, and element 1 contains the file descriptor for writing.

. . .

Represents three optional, positional arguments, *flag*, *bufsize*, and *bufquota*:

flag

An optional argument used as a bitmask.

If either the O_NDELAY or O_NONBLOCK bit is set, the I/O operations to the mailbox through *array_fdscptr* file descriptors terminate immediately, rather than waiting for another process.

If, for example, the O_NDELAY bit is set and the child issues a `read` request to the mailbox before the parent has put any data into it, the `read` terminates immediately with 0 status. If neither the O_NDELAY nor O_NONBLOCK bit is set, the child will be waiting on the read until the parent writes any data into the mailbox. This is the default behavior if no *flag* argument is specified.

The values of O_NDELAY and O_NONBLOCK are defined in the `<fcntl.h>` header file. Any other bits in the *flag* argument are ignored. You must specify this argument if the second optional, positional argument *bufsize* is specified. If the *flag* argument is needed only to allow specification of the *bufsize* argument, specify *flag* as 0.

bufsize

Optional argument of type `int` that specifies the size of the mailbox, in bytes. Specify a value from 512 to 65535.

If you specify 0 or omit this argument, the operating system creates a mailbox with a default size of 512 bytes.

If you specify a value less than 0 or larger than 65535, the results are unpredictable.

If you do specify this argument, be sure to precede it with a *flag* argument.

The DECC$PIPE_BUFFER_SIZE feature logical can also be used to specify the size of the mailbox. If *bufsize* is supplied, it takes precedence over the value of DECC$PIPE_BUFFER_SIZE. Otherwise, the value of DECC$PIPE_BUFFER_SIZE is used.

If neither *bufsize* nor DECC$PIPE_BUFFER_SIZE is specified, the default buffer size of 512 is used.

bufquota

Optional argument of type `int` that specifies the buffer quota of the pipe's mailbox. Specify a value from 512 to 2147483647.

OpenVMS Version 7.3-2 added this argument. In previous OpenVMS versions, the buffer quota was equal to the buffer size.

The DECC$PIPE_BUFFER_QUOTA feature logical can also be used to specify the buffer quota. If the optional *bufquota* argument of the `pipe` function is supplied, it takes precedence over the value of DECC$PIPE_BUFFER_QUOTA. Otherwise, the value of DECC$PIPE_BUFFER_QUOTA is used.

If neither *bufquota* nor DECC$PIPE_BUFFER_QUOTA is specified, then the buffer quota defaults to the buffer size.

Description

The mailbox used for the pipe is a temporary mailbox. The mailbox is not deleted until all processes that have open channels to that mailbox close those channels. The last process that closes a pipe writes a message to the mailbox, indicating the end-of-file.

The mailbox is created by using the $CREMBX system service, specifying the following characteristics:

- A maximum message length of 512 characters
- A buffer quota of 512 characters
- A protection mask granting all privileges to USER and GROUP and no privileges to SYSTEM or WORLD

The buffer quota of 512 characters implies that you cannot write more than 512 characters to the mailbox before all or part of the mailbox is read. Since a mailbox record is slightly larger than the data part of the message that it contains, not all of the 512 characters can be used for message data. You can increase the size of the buffer by specifying an alternative size using the optional, third argument to the `pipe` function. A pipe under the OpenVMS system is a stream-oriented file with no carriage-control attributes. It is fully buffered by default in the C RTL. A mailbox used as a pipe is different than a mailbox created by the application. A mailbox created by the application defaults to a record-oriented file with carriage return, carriage control. Additionally, writing a zero-length record to a mailbox writes an EOF, as does each close of the mailbox. For a pipe, only the last close of a pipe writes an EOF.

The pipe is created by the parent process before `vfork` and an `exec` function are called. By calling `pipe` first, the child inherits the open file descriptors for the pipe. You can then use the `getname` function to return the name of the mailbox associated with the pipe, if this information is desired. The mailbox name returned by `getname` has the format _MBA `nnnn`: (Alpha only) or _MBA `nnnnn`: (Integrity servers only), where `nnnn` or `nnnnn` is a unique number.

Both the parent and the child need to know in advance which file descriptors will be allocated for the pipe. This information cannot be retrieved at run time. Therefore, it is important to understand how file descriptors are used in any VSI C for OpenVMS program. For more information about file descriptors, see Chapter 2.

File descriptors 0, 1, and 2 are open in a VSI C for OpenVMS program for `stdin` (SYS$INPUT), `stdout` (SYS$OUTPUT), and `stderr` (SYS$ERROR), respectively. Therefore, if no other files are open when `pipe` is called, `pipe` assigns file descriptor 3 for writing and file descriptor 4 for reading. In the array returned by `pipe`, 4 is placed in element 0 and 3 is placed in element 1.

If other files have been opened, `pipe` assigns the first available file descriptor for writing and the next available file descriptor for reading. In this case, the pipe does not necessarily use adjacent file descriptors. For example, assume that two files have been opened and assigned to file descriptors 3 and 4 and the first file is then closed. If `pipe` is called at this point, file descriptor 3 is assigned for writing and file descriptor 5 is assigned for reading. Element 0 of the array will contain 5 and element 1 will contain 3.

In large applications that do large amounts of I/O, it gets more difficult to predict which file descriptors are going to be assigned to a pipe; and, unless the child knows which file descriptors are being used, it will not be able to read and write successfully from and to the pipe.

One way to be sure that the correct file descriptors are being used is to use the following procedure:

1. Choose two descriptor numbers that will be known to both the parent and the child. The numbers should be high enough to account for any I/O that might be done before the pipe is created.

2. Call `pipe` in the parent at some point before calling an `exec` function.
3. In the parent, use `dup2` to assign the file descriptors returned by `pipe` to the file descriptors you chose. This now reserves those file descriptors for the pipe; any subsequent I/O will not interfere with the pipe.

You can read and write through the pipe using the UNIX I/O functions `read` and `write`, specifying the appropriate file descriptors. As an alternative, you can issue `fdopen` calls to associate file pointers with these file descriptors so that you can use the Standard I/O functions (`fread` and `fwrite`).

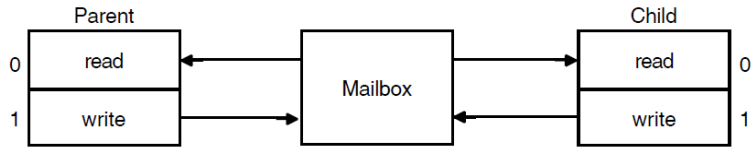
Two separate file descriptors are used for reading from and writing to the pipe, but only one mailbox is used so some I/O synchronization is required. For example, assume that the parent writes a message to the pipe. If the parent is the first process to read from the pipe, then it will read its own message back as shown in Figure 11.

Note

For added UNIX portability, you can use the following feature logicals to control the behavior of the C RTL pipe implementation:

- Define the DECC$STREAM_PIPE feature logical name to ENABLE to direct the `pipe` function to use stream I/O instead of record I/O.
- Define the DECC$POPEN_NO_CRLF_REC_ATTR feature logical to ENABLE to prevent CR/LF carriage control from being added to pipe records for pipes opened with the `popen` function. Be aware that enabling this feature might result in undesired behavior from other functions such as `gets` that rely on the carriage-return character.

Figure 11. Reading and Writing to a Pipe



Return Values

0

Indicates success.

-1

Indicates an error.

poll

poll — Provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open streams.

Format

```
#include <poll.h>
```

```
int poll (struct pollfd filedes [], nfds_t nfds, int timeout);
```

Argument

filedes

Points to an array of `pollfd` structures, one for each file descriptor of interest. Each `pollfd` structure includes the following members:

`int` *fd* – The file descriptor
`int` *events* – The requested conditions
`int` *revents* – The reported conditions

nfds

The number of `pollfd` structures in the *filedes* array.

timeout

The maximum length of time (in milliseconds) to wait for at least one of the specified events to occur.

Description

The `poll` function provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open streams. For each member of the array pointed to by *filedes*, `poll` examines the given file descriptor for the event(s) specified in events. The `poll` function identifies those streams on which an application can send or receive messages, or on which certain events have occurred.

The *filedes* parameter specifies the file descriptor to be examined and the events of interest for each file descriptor. It is a pointer to an array of `pollfd` structures. The *fd* member of each `pollfd` structure specifies an open file descriptor. The `poll` function uses the *events* member to determine what conditions to report for this file descriptor. If one or more of these conditions is true, the `poll` function sets the associated *revents* member. The `poll` function sets *revents* to 0 at the beginning of the call so the *revents* value is erased, even if no events occur.

The *events* and *revents* members of each `pollfd` structure are bitmasks. The calling process sets the events bitmask, and `poll` sets the revents bitmasks. These bitmasks contain inclusive ORed combinations of condition options. The following condition options are defined:

- POLLERR– An error has occurred on the file descriptor. This option is only valid in the revents bitmask; it is not used in the events member. For STREAMS devices, if an error occurs on the file descriptor and the device is also disconnected, `poll` returns POLLERR. For pipes, POLLERR is set for a file descriptor referring to the write end of a pipe when the read end has been closed. (The POSIX standard requires that `poll` should set POLLHUP for an attempt to read from a pipe when all file descriptors referring to the write end of the pipe have been closed. However, the current VSI C RTL implementation does *not* support this behavior.)

- POLLIN– Data other than high-priority data may be read without blocking. This option is set in revents even if the message is of zero length.

- POLLNVAL– The value specified for *fd* is invalid. This option is only valid in the *revents* member; it is ignored in the *events* member.

- POLLOUT– Normal (priority band equals 0) data may be written without blocking.

- POLLPRI– High-priority data may be received without blocking. This option is set in *revents* even if the message is of zero length.
- POLLRDBAND– Data from a nonzero priority band may be read without blocking. This option is set in *revents* even if the message is of zero length.
- POLLRDNORM– Normal data (priority band equals 0) may be read without blocking. This option is set in *revents* even if the message is of zero length.
- POLLWRBAND– Priority data (priority band greater than 0) may be written. This event only examines bands that have been written to at least once.
- POLLWRNORM– Same as POLLOUT.

The conditions indicated by POLLNORM and POLLOUT are true if and only if at least one byte of data can be read or written without blocking. There are two exceptions: regular files, which always poll true for POLLNORM and POLLOUT, and pipes, when the rules for the operation specify to return zero in order to indicate end-of-file.

The condition options POLLERR and POLLNVAL are always set in *revents* if the conditions they indicate are true for the specified file descriptor, whether or not these options are set in *events*.

For each call to the `poll` function, the set of reportable conditions for each file descriptor consists of those conditions that are always reported, together with any further conditions for which options are set in *events*. If any reportable condition is true for any file descriptor, the `poll` function will return with options set in *revents* for each true condition for that file descriptor.

If no reportable condition is true for any of the file descriptors, the `poll` function waits up to *timeout* milliseconds for a reportable condition to become true. If, in that time interval, a reportable condition becomes true for any of the file descriptors, `poll` reports the condition in the file descriptor's associated *revents* member and returns. If no reportable condition becomes true, `poll` returns without setting any *revents* bitmasks.

If the *timeout* parameter is a negative value, the `poll` function does not return until at least one specified event has occurred. If the value of the *timeout* parameter is 0 (zero), the `poll` function does not wait for an event to occur but returns immediately, even if no specified event has occurred.

The behavior of the `poll` function is not affected by whether the O_NONBLOCK option is set on any of the specified file descriptors.

The `poll` function supports regular files, terminal and pseudo-terminal devices, STREAMS-based files, mailboxes, sockets, and pipes. The behavior of `poll` on elements of file descriptors that refer to other types of files is unspecified.

For sockets, a file descriptor for a socket that is listening for connections indicates it is ready for reading after connections are available. A file descriptor for a socket that is connecting asynchronously indicates it is ready for writing after a connection is established.

Return Values

n

Upon successful completion, a nonnegative value is returned, indicating the number of file descriptors for which `poll` has set the revents bitmask.

0

`poll` has timed out and has not set any of the revents bitmasks.

-1

An error occurred. `errno` is set to indicate the error:

- EINTR – A signal was intercepted during the `poll` function, and the signal handler was installed with an indication that functions are not to be restarted.
- EIO – OpenVMS system service (SYS$QIOW etc.) failed.
- ENOMEM – Allocation of internal data structures failed. A later call to the `poll` function might complete successfully.
- ENOSYS – Socket descriptors were passed to the `poll` function, and `poll` failed to load the TCP/IP library.

popen

popen — Initiates a pipe to a process.

Format

```
#include <stdio.h>
FILE *popen (const char *command, const char *type);
```

Arguments

command

A pointer to a null-terminated string containing a shell command line.

type

A pointer to a null-terminated string containing an I/O mode. Because open files are shared, you can use a type `r` command as an input filter and a type `w` command as an output filter. Specify one of the following values for the *type* argument:

- `r`—the calling program can read from the standard output of the command by reading from the returned file stream.
- `w`—the calling program can write to the standard input of the command by writing to the returned file stream.

In addition, the *type* argument can include a type `e` command, which sets the FD_CLOEXEC flag on the file descriptor.

Description

The `popen` function creates a pipe between the calling program and a shell command awaiting execution. It returns a pointer to a `FILE` structure for the stream.

The popen function uses the value of the DECC$PIPE_BUFFER_SIZE feature logical to set the buffer size of the mailbox it creates for the pipe. You can specify a DECC$PIPE_BUFFER_SIZE value of 512 to 65024 bytes. If DECC$PIPE_BUFFER_SIZE is not specified, the default buffer size of 512 is used.

Notes

- When you use the popen function to invoke an output filter, beware of possible deadlock caused by output data remaining in the program buffer. You can avoid this by either using the setvbuf function to ensure that the output stream is unbuffered, or the fflush function to ensure that all buffered data is flushed before calling the pclose function.
- For added UNIX portability, you can use the following feature logicals to control the behavior of the C RTL pipe implementation:
  - Define the DECC$STREAM_PIPE feature logical name to ENABLE to direct the pipe function to use stream I/O instead of record I/O.
  - Define the DECC$POPEN_NO_CRLF_REC_ATTR feature logical to ENABLE to prevent CR/LF carriage control from being added to pipe records for pipes opened with the popen function. Be aware that enabling this feature might result in undesired behavior from other functions such as gets that rely on the carriage-return character.

See also fflush, pclose, and setvbuf.

Return Values

x

A pointer to the FILE structure for the opened stream.

NULL

Indicates an error. Unable to create files or processes.

pow

pow — Returns the first argument raised to the power of the second argument.

Format

```
#include <math.h>
double pow (double x, double y);
float powf (float x, float y);
long double powl (long double x, long double y);
```

Arguments

x

A floating-point base to be raised to an exponent *y*.

y

The exponent to which the base *x* is to be raised.

Description

The `pow` functions raise a floating-point base *x* to a floating-point exponent *y*. The value of `pow`(*x*, *y*) is computed as e**(*y* ln(*x*)) for positive *x*.

If *x* is 0 and *y* is negative, ±HUGE_VAL is returned and `errno` is set to ERANGE or EDOM.

Return Values

x

The result of the first argument raised to the power of the second.

1.0

The base is 0 and the exponent is 0.

HUGE_VAL

The result overflowed; `errno` is set to ERANGE.

±HUGE_VAL

The base is 0 and the exponent is negative; `errno` is set to ERANGE or EDOM.

Example

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

main()
{
    double x;

    errno = 0;

    x = pow(-3.0, 2.0);
    printf("%d, %f\n", errno, x);
}
```

This example program outputs the following:

```
0, 9.000000
```

pread

pread — Reads bytes from a given position within a file without changing the file pointer.

Format

```
#include <unistd.h>
ssize_t pread (int file_desc, void *buffer, size_t nbytes, off_t offset);
```

Arguments

file_desc

A file descriptor that refers to a file currently opened for reading.

buffer

The address of contiguous storage in which the input data is placed.

nbytes

The maximum number of bytes involved in the read operation.

offset

The offset for the desired position inside the file.

Description

The `pread` function performs the same action as `read`, except that it reads from a given position in the file without changing the file pointer. The first three arguments to `pread` are the same as for `read`, with the addition of a fourth argument *offset* for the desired position inside the file. An attempt to perform a `pread` on a file that is incapable of seeking results in an error.

Return Values

n

The number of bytes read.

-1

Upon failure, the file pointer remains unchanged and `pread` sets *errno* to one of the following values:

- EINVAL– The offset argument is invalid. The value is negative.
- EOVERFLOW– The file is a regular file, and an attempt was made to read or write at or beyond the offset maximum associated with the file.
- ENXIO– A request was outside the capabilities of the device.
- ESPIPE – fildes is associated with a pipe or FIFO.

printf

printf — Performs formatted output from the standard output (`stdout`). See Chapter 2 for information on format specifiers.

Format

```
#include <stdio.h>
int printf (const char *format_spec, ...);
```

Arguments

format_spec

Characters to be written literally to the output or converted as specified in the . . . arguments.

. . .

Optional expressions whose resultant types correspond to conversion specifications given in the format specification.

If no conversion specifications are given, you may omit the output sources. Otherwise, the function call must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources.

Conversion specifications are matched to output sources in left-to-right order. Excess output pointers, if any, are ignored.

Return Values

x

The number of bytes written.

Negative value

Indicates that an output error occurred. The function sets `errno`. For a list of `errno` values set by this function, see `fprintf`.

[w]printw

[w]printw — Perform a `printf` in the specified window, starting at the current position of the cursor. The `printw` function acts on the `stdscr` window.

Format

```
#include <curses.h>
printw (char *format_spec, ...);
int wprintw (WINDOW *win, char *format_spec, ...);
```

Arguments

win

A pointer to the window.

format_spec

A pointer to the format specification string.

. . .

Optional expressions whose resultant types correspond to conversion specifications given in the format specification.

If no conversion specifications are given, you may omit the output sources. Otherwise, the function call must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources.

Conversion specifications are matched to output sources in left-to-right order. Excess output pointers, if any, are ignored.

Description

The formatting specification (*format_spec*) and the other arguments are identical to those used with the `printf` function.

The `printw` and `wprintw` functions format and then print the resultant string to the window using the `addstr` function. For more information, see the `printf` and `scrollok` functions in this section. See Chapter 2 for information on format specifiers.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the window scroll illegally.

putc

putc — The `putc` macro writes a single character to a specified file.

Format

```
#include <stdio.h>
int putc (int character, FILE *file_ptr);
```

Arguments

character

The character to be written.

file_ptr

A file pointer to the output stream.

Description

The `putc` macro writes the byte *character* (converted to an unsigned char) to the output specified by the *file_ptr* parameter. The byte is written at the position at which the file pointer is currently pointing (if defined) and advances the indicator appropriately. If the file cannot support positioning requests, or if the output stream was opened with append mode, the byte is appended to the output stream.

Since `putc` is a macro, a file pointer argument with side effects (for example, `putc (ch, *f++)`) might be evaluated incorrectly. In such a case, use the `fputc` function instead.

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that uses a faster, inlined version of this function.

See also `putc_unlocked`.

Return Values

x

The character written to the file. Indicates success.

EOF

Indicates output errors.

putc_unlocked

putc_unlocked — Same as `putc`, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

```
#include <stdio.h>
int putc_unlocked (int character, FILE *file_ptr);
```

Argument

character

The character to be written.

file_ptr

A file pointer to the output stream.

Description

The reentrant version of the `putc` macro is locked against multiple threads calling it simultaneously. This incurs overhead to ensure integrity of the stream. The unlocked version of this call, `putc_unlocked` can be used to avoid the overhead. The `putc_unlocked` macro is functionally identical to the `putc` macro, except that it is not required to be implemented in a thread-safe manner. The `putc_unlocked` macro can be safely used only within a scope that is protected by the `flockfile` and `funlockfile` functions used as a pair. The caller must ensure that the stream is locked before `putc_unlocked` is used.

Since `putc_unlocked` is a macro, a file pointer argument with side effects might be evaluated incorrectly. In such a case, use the `fputc_unlocked` function instead.

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that uses a faster, inlined version of this function.

See also `flockfile`, `ftrylockfile`, and `funlockfile`.

Return Values

x

The character written to the file. Indicates success.

EOF

Indicates the end-of-file or an error.

putchar

putchar — Writes a single character to the standard output (`stdout`) and returns the character.

Format

```
#include <stdio.h>
int putchar (int character);
```

Argument

character

An object of type `int`.

Description

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

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that uses a faster, inlined version of this function.

Return Values

character

Indicates success.

EOF

Indicates output errors.

putchar_unlocked

putchar_unlocked — Same as `putchar`, except used only within a scope protected by `flockfile` and `funlockfile`.

Format

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

```
int putchar_unlocked (int character);
```

Argument

character

An object of type `int`.

Description

The reentrant version of the `putchar` function is locked against multiple threads calling it simultaneously. This incurs overhead to ensure integrity of the output stream. The unlocked version of this call, `putchar_unlocked` can be used to avoid the overhead. The `putchar_unlocked` function is functionally identical to the `putchar` function, except that it is not required to be implemented in a thread-safe manner. The `putchar_unlocked` function can be safely used only within a scope that is protected by the `flockfile` and `funlockfile` functions used as a pair. The caller must ensure that the stream is locked before `putchar_unlocked` is used.

Compiling with the _ _UNIX_PUTC macro defined enables an optimization that uses a faster, inlined version of this function.

See also `flockfile`, `ftrylockfile`, and `funlockfile`.

Return Values

x

The next character from `stdin`, converted to `int`.

EOF

Indicates the end-of-file or an error.

putenv

putenv — Sets an environmental variable.

Format

```
#include <stdlib.h>
int putenv (const char *string);
```

Argument

string

A pointer to a *name=value* string.

Description

The `putenv` function sets the value of an environment variable by altering an existing variable or by creating a new one. The *string* argument points to a string of the form *name=value*, where *name* is the environment variable and *value* is the new value for it.

The string pointed to by *string* becomes part of the environment, so altering the string changes the environment. When a new string-defining name is passed to `putenv`, the space used by *string* is no longer used.

Notes

- The `putenv` function manipulates the environment pointed to by the `environ` external variable, and can be used with `getenv`. However, the third argument to the main function (the environment pointer), is not changed.

  The `putenv` function uses the `malloc` function to enlarge the environment.

  A potential error is to call `putenv` with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

- Do not use the `setenv`, `getenv`, and `putenv` functions to manipulate symbols and logicals. Instead, use the OpenVMS library calls `lib$set_logical`, `lib$get_logical`, `lib$set_symbol`, and `lib$get_symbol`. The * `env` functions deliberately provide UNIX behavior, and are not a substitute for these OpenVMS runtime library calls.

  OpenVMS DCL symbols, not logical names, are the closest analog to environment variables on UNIX systems. While `getenv` is a mechanism to retrieve either a logical name or a symbol, it maintains an internal cache of values for use with `setenv` and subsequent `getenv` calls. The `setenv` function does not write or create DCL symbols or OpenVMS logical names.

  This is consistent with UNIX behavior. On UNIX systems, `setenv` does not change or create any symbols that will be visible in the shell after the program exits.

Return Values

0

Indicates success.

-1

Indicates an error. `errno` is set to ENOMEM – Not enough memory available to expand the environment list.

Restriction

The `putenv` function cannot take a 64-bit address. See Section 1.9.

puts

puts — Writes a character string to the standard output (`stdout`) followed by a new-line character.

Format

```
#include <stdio.h>
int puts (const char *str);
```

Argument

str

A pointer to a character string.

Description

The `puts` function does not copy the terminating null character to the output stream.

Return Values

Nonnegative value

Indicates success.

EOF

Indicates output errors.

putw

putw — Writes characters to a specified file.

Format

```
#include <stdio.h>
int putw (int integer, FILE *file_ptr);
```

Arguments

integer

An object of type `int` or `long`.

file_ptr

A file pointer.

Description

The `putw` function writes four characters to the output file as an `int`. No conversion is performed.

Return Values

integer

Indicates success.

EOF

Indicates output errors.

putwc

putwc — Converts a wide character to its corresponding multibyte value, and writes the result to a specified file.

Format

```
#include <wchar.h>
wint_t putwc (wint_t wc, FILE *file_ptr);
```

Arguments

wc

An object of type `wint_t`.

file_ptr

A file pointer.

Description

Since `putwc` might be implemented as a macro, a file pointer argument with side effects (for example `putwc (wc, *f++)`) might be evaluated incorrectly. In such a case, use the `fputwc` function instead.

See also `fputwc`.

Return Values

x

The character written to the file. Indicates success.

WEOF

Indicates an output error. The function sets `errno`. For a list of the `errno` values set by this function, see `fputwc`.

putwchar

putwchar — Writes a wide character to the standard output (`stdout`) and returns the character.

Format

```
#include <wchar.h>
wint_t putwchar (wint_t wc);
```

Arguments

wc

An object of type `wint_t`.

Description

The `putwchar` function is identical to `fputwc` (*wc*, `stdout`).

Return Values

x

The character written to the file. Indicates success.

WEOF

Indicates an output error. The function sets `errno`. For a list of the `errno` values set by this function, see `fputwc`.

pwrite

pwrite — Writes into a given position within a file without changing the file pointer.

Format

```
#include <unistd.h>
ssize_t pwrite (int file_desc, const void *buffer, size_t nbytes,
off_t offset);
```

Arguments

file_desc

A file descriptor that refers to a file currently opened for writing or updating.

buffer

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

nbytes

The maximum number of bytes involved in the write operation.

offset

The offset for the desired position inside the file.

Description

The `pwrite` function performs the same action as `write`, except that it writes into a given position in the file without changing the file pointer. The first three arguments to `pwrite` are the same as for `write`, with the addition of a fourth argument *offset* for the desired position inside the file.

Return Values

n

The number of bytes written.

-1

Upon failure, the file pointer remains unchanged and `pwrite` sets `errno` to one of the following values:

- EINVAL– The offset argument is invalid. The value is negative.
- ESPIPE– fildes is associated with a pipe or FIFO.

qabs, llabs

qabs, llabs — Returns the absolute value of an integer as an `__int64`. `llabs` is a synonym for `qabs`.

Format

```
#include <stdlib.h>
__int64 qabs (__int64 j);
__int64 llabs (__int64 j);
```

Argument

j

A value of type `__int64`.

qdiv, lldiv

qdiv, lldiv — Returns the quotient and the remainder after the division of its arguments. `lldiv` is a synonym for `qdiv`.

Format

```
#include <stdlib.h>
qdiv_t qdiv (_ _int64 numer, __int64 denom);
lldiv_t lldiv (_ _int64 numer, __int64 denom);
```

Arguments

numer

A numerator of type `__int64`.

denom

A denominator of type `__int64`.

Description

The types `qdiv_t` and `lldiv_t` are defined in the `<stdlib.h>` header file as follows:

```
typedef struct
         {
```

```
          __int64 quot, rem;
          }
qdiv_t, lldiv_t;
```

qsort

qsort — Sorts an array of objects in place. It implements the quick-sort algorithm.

Format

```
#include <stdlib.h>
void qsort (void *base, size_t nmemb, size_t size, int (*compar)
(const void *, const void *));
```

Function Variants

The `qsort` function has variants named `_qsort32` and `_qsort64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

base

A pointer to the first member of the array. The pointer should be of type pointer-to-element and cast to type pointer-to-character.

nmemb

The number of objects in the array.

size

The size of an object, in bytes.

compar

A pointer to the comparison function.

Description

Two arguments are passed to the comparison function pointed to by *compar*. The two arguments point to the objects being compared. Depending on whether the first argument is less than, equal to, or greater than the second argument, the comparison function returns an integer less then, equal to, or greater than 0.

The comparison function *compar* need not compare every byte, so arbitrary data might be contained in the objects in addition to the values being compared.

The order in the output of two objects that compare as equal is unpredictable.

Return Value

The `qsort` function returns no value.

qsort_r

qsort_r — Sorts an array of objects in place by implementing the quick-sort algorithm (reentrant).

Format

```
#include <stdlib.h>
void qsort_r (void *base, size_t nmemb, size_t size, int (*compar)
(const void *, const void *, void *), void *arg)
```

Function Variants

The `qsort_r` function has variants named `_qsort_r32` and `_qsort_r64` for use with 32-bit and 64-bit pointer sizes, respectively.

Arguments

base

A pointer to the first member of the array. The pointer should be of type pointer-to-element and cast to type pointer-to-character.

nmemb

The number of objects in the array.

size

The size of each object in the array, in bytes.

compar

A pointer to the comparison function.

arg

A pointer to arbitrary arguments for the comparison function.

Description

The `qsort_r` function is the reentrant version of `qsort`. `qsort_r` is identical to `qsort` except that the comparison function *compar* takes a third argument. A pointer is passed to the comparison function via *arg*.

Return Value

The `qsort_r` function returns no value.

raise

raise — Generates a specified software signal. Generating a signal causes the action routine established by the `signal`, `ssignal`, or `sigvec` function to be invoked.

Format

```
#include <signal.h>
int raise (int sig); (ANSI C)
int raise (int sig[, int sigcode]); (VSI C Extension)
```

Arguments

sig

The signal to be generated.

sigcode

An optional signal code, available only when not compiling in strict ANSI C mode. For example, signal SIGFPE—the arithmetic trap signal—has 10 different codes, each representing a different type of arithmetic trap.

The signal codes can be represented by mnemonics or numbers. The arithmetic trap codes are represented by the numbers 1 to 10; the SIGILL codes are represented by the numbers 0 to 2. The code values are defined in the `<signal.h>` header file. See Table 4.4 for a list of signal mnemonics, codes, and corresponding OpenVMS exceptions.

Description

Calling the `raise` function has one of the following results:

- If `raise` specifies a *sig* argument that is outside the range defined in the `<signal.h>` header file, then the `raise` function returns 0, and the `errno` variable is set to EINVAL.
- If `signal`, `ssignal`, or `sigvec` establishes SIG_DFL (default action) for the signal, then the functions do not return. The image is exited with the OpenVMS error code corresponding to the signal.
- If `signal`, `ssignal`, or `sigvec` establishes SIG_IGN (ignore signal) as the action for the signal, then `raise` returns its argument, *sig*.
- `signal`, `ssignal`, or `sigvec` must establish an action function for the signal. That function is called and its return value is returned by `raise`.

See Chapter 4 for more information on signal processing.

See also `gsignal`, `signal`, `ssignal`, and `sigvec`.

Return Values

0

If successful.

nonzero

If unsuccessful.

rand, rand_r

rand, rand_r — Returns pseudorandom numbers in the range 0 to $2^{31} - 1$.

Format

```
#include <stdlib.h>
int rand (void);
int rand_r (unsigned int seed);
```

Argument

seed

An initial seed value.

Description

The `rand` function computes a sequence of pseudorandom integers in the range 0 to {RAND_MAX} with a period of at least 2^{32}.

The `rand_r` function computes a sequence of pseudorandom integers in the range 0 to {RAND_MAX}. The value of the {RAND_MAX} macro will be at least 32767.

If `rand_r` is called with the same initial value for the object pointed to by *seed* and that object is not modified between successive returns and calls to `rand_r`, the same sequence is generated.

See also `srand`.

For other random-number algorithms, see `random` and all the * `48` functions.

Return Value

n

A pseudorandom number.

random

random — Generates pseudorandom numbers in a more random sequence.

Format

```
#include <stdlib.h>
long int random (void);
```

Description

The `random` function is a random-number generator that has virtually the same calling sequence and initialization properties as the `rand` function, but produces sequences that are more random. The low 12 bits generated by `rand` go through a cyclic pattern. All bits generated by `random` are usable. For example, `random`() &1 produces a random binary value.

The `random` function uses a nonlinear, additive-feedback, random-number generator employing a default state-array size of 31 integers to return successive pseudorandom numbers in the range from 0 to 2^{31}-1. The period of this random-number generator is approximately 16*(2^{31}-1). The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

With a full 256 bytes of state information, the period of the random-number generator is greater than 2^{69}, and is sufficient for most purposes.

Like the `rand` function, the `random` function produces by default a sequence of numbers that you can duplicate by calling the `srandom` function with a value of 1 as the seed. The `srandom` function, unlike the `srand` function, does not return the old seed because the amount of state information used is more than a single word.

See also `rand`, `srand`, `srandom`, `setstate`, and `initstate`.

Return Value

n

A random number.

[no]raw

[no]raw — Raw mode only works with the Curses input routines `[w]getch` and `[w]getstr`. Raw mode is not supported with the C RTL emulation of UNIX I/O, Terminal I/O, or Standard I/O.

Format

```
#include <curses.h>
raw()
noraw()
```

Description

Raw mode reads are satisfied on one of two conditions: after a minimum number (5) of characters are input at the terminal or after waiting a fixed time (10 seconds) from receipt of any characters from the terminal.

Example

```
/* Example of standard and raw input in Curses package. */

#include <curses.h>

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

    /* Initialize standard screen, turn echo off.  */
```

```
    initscr();
    noecho();

    /* Define a user window.  */

    win1 = newwin(22, 78, 1, 1);
    leaveok(win1, TRUE);
    leaveok(stdscr, TRUE);

    box(stdscr, vert, hor);

    /* Reset the video, refresh(redraw) both windows. */

    mvwaddstr(win1, 2, 2, "Test line terminated input");
    wrefresh(win1);

    /* Do some input and output it. */
    nocrmode();
    wgetstr(win1, str);

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

    /* Get another character then delete the window. */

    wgetch(win1);
    wclear(win1);

    mvwaddstr(win1, 2, 2, "Test raw input");
    wrefresh(win1);

    /* Do some raw input 5 chars or timeout - and output it. */
    raw();
    getstr(str);
    noraw();
    mvwaddstr(win1, 5, 5, str);
    mvwaddstr(win1, 7, 7, "Raw input completed");
    wrefresh(win1);

    endwin();
}
```

read

read — Reads bytes from a file and places them in a buffer.

Format

```
#include <unistd.h>
ssize_t read (int file_desc, void *buffer, size_t nbytes); (ISO POSIX-1)
int read (int file_desc, void *buffer, int nbytes); (Compatibility)
```

Arguments

file_desc

A file descriptor. The specified file descriptor must refer to a file currently opened for reading.

buffer

The address of contiguous storage in which the input data is placed.

nbytes

The maximum number of bytes involved in the read operation.

Description

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

Note

The `read` function does not span record boundaries in a record file and, therefore, reads at most one record. A separate read must be done for each record.

Return Values

n

The number of bytes read.

-1

Indicates a read error, including physical input errors, illegal buffer addresses, protection violations, undefined file descriptors, and so forth.

Example

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

main()
{
    int fd,
        i;
    char buf[10];
    FILE *fp ;          /* Temporary STDIO file */

    /* Create a dummy data file  */

    if ((fp = fopen("test.txt", "w+")) == NULL) {
        perror("open");
        exit(1);
    }
    fputs("XYZ\n",fp) ;
    fclose(fp) ;
```

```
    /* And now practice "read" */

    if ((fd = open("test.txt", O_RDWR, 0, "shr=upd")) <= 0) {
        perror("open");
        exit(0);
    }

    /* Read 2 characters into buf.  */

    if ((i = read(fd, buf, 2)) < 0) {
        perror("read");
        exit(0);
    }

    /* Print out what was read.  */

    if (i > 0)
        printf("buf='%c%c'\n", buf[0], buf[1]);

    close(fd);
}
```

readdir, readdir_r

readdir, readdir_r — Finds entries in a directory.

Format

```
#include <dirent.h>
struct dirent *readdir (DIR *dir_pointer);
int readdir_r (DIR *dir_pointer, struct dirent *entry,
struct dirent **result);
```

Arguments

dir_pointer

A pointer to the `dir` structure of an open directory.

entry

A pointer to a `dirent` structure that will be initialized with the directory entry at the current position of the specified stream.

result

Upon successful completion, the location where a pointer to *entry* is stored.

Description

The `readdir` function returns a pointer to a structure representing the directory entry at the current position in the directory stream specified by *dir_pointer*, and positions the directory stream at the next entry. It returns a NULL pointer upon reaching the end of the directory stream. The `dirent` structure defined in the `<dirent.h>` header file describes a directory entry.

The type `DIR` defined in the `<dirent.h>` header file represents a directory stream. A directory stream is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files. You can remove files from or add files to a directory asynchronously to the operation of the `readdir` function.

The pointer returned by the `readdir` function points to data that you can overwrite by another call to `readdir` on the same directory stream. This data is not overwritten by another call to `readdir` on a different directory stream.

If a file is removed from or added to the directory after the most recent call to the `opendir` or `rewinddir` function, a subsequent call to the `readdir` function might not return an entry for that file.

When it reaches the end of the directory, or when it detects an invalid `seekdir` operation, the `readdir` function returns the null value.

An attempt to seek to an invalid location causes the `readdir` function to return the null value the next time it is called. A previous `telldir` function call returns the position.

The `readdir_r` function is a reentrant version of `readdir`. In addition to *dir_pointer*, you must specify a pointer to a `dirent` structure in which the current directory entry of the specified stream is returned.

If the operation is successful, `readdir_r` returns 0 and stores one of the following two pointers in *result*:

- Pointer to *entry* if the entry was found
- NULL pointer if the end of the directory stream was reached

The storage pointed to by *entry* must be large enough for a `dirent` with an array of `char d_name` member containing at least NAME_MAX + 1 elements.

If an error occurred, an error value is returned that indicates the cause of the error.

Applications wishing to check for error situations should set `errno` to 0 before calling `readdir`. If `errno` is set to nonzero on return, then an error occurred.

Example

See the description of `closedir` for an example.

Return Values

x

On successful completion of `readdir`, a pointer to an object of type `struct dirent`.

0

Successful completion of `readdir_r`.

x

On error, an error value (`readdir_r` only).

NULL

An error occurred or end of the directory stream (`readdir_r` only). If an error occurred, `errno` is set to a value indicating the cause.

readlink

readlink — Reads the contents of the specified symbolic link and places them into a user-supplied buffer.

Format

```
#include <unistd.h>
ssize_t readlink (const char *restrict link_name,
char *restrict user_buffer, size_t buffer_size);
```

Arguments

link_name

Pointer to the text string representing the name of the symbolic link file.

user_buffer

Pointer to the user buffer.

buffer_size

Size of the user buffer.

Description

The `readlink` function reads the contents of the specified symbolic link (*link_name*) and places them into a user-supplied buffer (*user_buffer*) of size (*buffer_size*).

See also `symlink`, `unlink`, `realpath`, `lchown`, and `lstat`.

Return Values

n

Upon successful completion, the count of bytes placed in the *user_buffer*

-1

Indicates an error. The buffer is unchanged, and `errno` is set to indicate the error:

- EACCES– Read permission is denied in the directory where the symbolic link is being read, or search permission is denied for a component of the path prefix of *link_name*.
- ENAMETOOLONG– The length of the *link_name* argument exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
- Any `errno` value from `close`, `open`, or `read`.

readv

readv — Reads from a file.

Format

```
#include <sys/uio.h>
ssize_t readv (int file_desc, const struct iovec *iov, int iovcnt);
ssize_t _readv64 (int file_desc, struct __iovec64 *iov, int iovcnt);
```

Function Variants

The `readv` function has variants named `_readv32` and `_readv64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

file_desc

A file descriptor. A file descriptor that must refer to a file currently opened for reading.

iov

Array of `iovec` structures into which the input data is placed.

iovcnt

The number of buffers specified by the members of the *iov* array.

Description

The `readv` function is equivalent to `read`, but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1]. The *iovcnt* argument is valid if it is greater than 0 and less than or equal to IOV_MAX.

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. The `readv` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv` marks for update the st_atime field of the file.

If the Synchronized Input and Output option is supported:

If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor will complete as defined by synchronized I/O data integrity completion.

If the O_SYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor will complete as defined by synchronized I/O file integrity completion.

If the Shared Memory Objects option is supported:

If *file_desc* refers to a shared memory object, the result of the `read` function is unspecified.

For regular files, no data transfer occurs past the offset maximum established in the open file description associated with *file_desc*.

Return Values

n

The number of bytes read.

- 1

Indicates a read error. The function sets errno to one of the following values:

- EAGAIN– The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed.
- EBADF– The *file_desc* argument is not a valid file descriptor open for reading.
- EBADMSG– The file is a STREAM file that is set to control-normal mode, and the message waiting to be read includes a control part.
- EINTER– The read operation was terminated because of the receipt of a signal, and no data was transferred.
- EINVAL– The STREAM or multiplexer referenced by *file_desc* is linked (directly or indirectly) downstream from a multiplexer.

  OR

  The sum of the iov_len values in the *iov* array overflowed an `ssize_t`.
- EIO– A physical I/O error has occurred.

  OR

  The process is a member of a background process attempting to read from its controlling terminal, the process is ignoring or blocking the SIGTTIN signal, or the process group is orphaned.
- EISDIR– The *file_desc* argument refers to a directory, and the implementation does not allow the directory to be read using `read`, `pread` or `readv`. Use the `readdir` function instead.
- EOVERFLOW– The file is a regular file, *nbyte* is greater than 0, and the starting position is before the end-of-file and is greater than or equal to the offset maximum established in the open file description associated with *file_desc*.

The `readv` function *may* fail if:

- EINVAL– The *iovcnt* argument was less than or equal to 0, or greater than IOV_MAX.

realloc

realloc — Changes the size of the area pointed to by the first argument to the number of bytes given by the second argument. These functions are AST-reentrant.

Format

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

```
void *realloc (void *ptr, size_t size);
```

Function Variants

The `realloc` function has variants named `_realloc32` and `_realloc64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

ptr

Points to an allocated area, or can be NULL.

size

The new size of the allocated area.

Description

If *ptr* is the NULL pointer, the behavior of the `realloc` function is identical to the `malloc` function.

The contents of the area are unchanged up to the lesser of the old and new sizes. The ANSI C Standard states that, "If the new size is larger than the old size, the value of the newly allocated portion of memory is indeterminate." For compatibility with old implementations, VSI C initializes the newly allocated memory to 0.

For efficiency, the previous actual allocation could have been larger than the requested size. If it was allocated with `malloc`, the value of the portion of memory between the previous requested allocation and the actual allocation is indeterminate. If it was allocated with `calloc`, that same memory was initialized to 0. If your application relies on `realloc` initializing memory to 0, then use `calloc` instead of `malloc` to perform the initial allocation. The maximum amount of memory allocated at once is limited to 0xFFFFD000.

See also `free`, `cfree`, `calloc`, and `malloc`.

Return Values

x

The address of the area, quadword-aligned (Alpha only) or octaword-aligned (Integrity servers only). The address is returned because the area may have to be moved to a new address to reallocate enough space. If the area was moved, the space previously occupied is freed.

NULL

Indicates that space cannot be reallocated (for example, if there is not enough room).

realpath

realpath — Returns an absolute pathname from the POSIX root.

Format

```
#include <stdlib.h>
char realpath (const char *restrict file_name,
char *restrict resolved_name);
```

Arguments

file_name

Pointer to the text string representing the name of the file for which you want the absolute path.

resolved name

Pointer to the generated absolute path stored as a null-terminated string.

Description

The `realpath` function returns an absolute pathname from the POSIX root. The generated pathname is stored as a null-terminated string, up to a maximum of PATH_MAX bytes, in the buffer pointed to by *resolved_name*.

The `realpath` function is supported only in POSIX-compliant modes (that is, with DECC$POSIX_COMPLIANT_PATHNAMES defined to one of the allowed values).

See also `symlink`, `unlink`, `readlink`, `lchown`, and `lstat`.

Return Values

x

Upon successful completion, a pointer to the *resolved_name*.

NULL

Indicates an error. A null pointer is returned, the contents of the buffer pointed to by *resolved_name* are undefined, and `errno` is set to indicate the error:

- ENAMETOOLONG– The length of the *file_name* argument exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
- ENOENT– A component of *file_name* does not name an existing file, or *file_name* points to an empty string.
- Any `errno` value from `chdir` or `stat`.

[w]refresh

[w]refresh — Repaint the specified window on the terminal screen. The `refresh` function acts on the `stdscr` window.

Format

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

```
int refresh();
int wrefresh (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

The result of this process is that the portion of the window not occluded by subwindows or other windows appears on the terminal screen. To see the entire occluded window on the terminal screen, call the `touchwin` function instead of the `refresh` or `wrefresh` function.

See also `touchwin`.

Return Values

OK

Indicates success.

ERR

Indicates an error.

remainder

remainder — Returns the floating-point remainder *r* = *x* - *n***y*) when *y* is nonzero.

Format

```
#include <math.h>
double remainder (double x, double y);
float remainderf (float x, float y);
ong double remainderl (long double x, long double y);
```

Argument

x

A real number.

y

A real number.

Description

These functions return the floating-point remainder *r* = *x* - *n***y*) when *y* is nonzero. The value *n* is the integral value nearest the exact value *x*/ *y*. That is, *n* =`rint`(*x*/*y*).

When `|`*n* `- x/y| = 1/2`, the value *n* is chosen to be even.

The behavior of the `remainder` function is independent of the rounding mode.

The `remainder` functions are functionally equivalent to the `remquo` functions.

Return Values

r

Upon successful completion, these functions return the floating-point remainder *r* = *x* - *ny* when *y* is nonzero.

Nan

If *x* or *y* is Nan.

remquo

remquo — Returns the floating-point remainder *r* = *x* - *n***y*) when *y* is nonzero.

Format

```
#include <math.h>
double remquo (double x, double y, int * quo);
float remquof (float x, float y, int * quo);
long double remquol (long double x, long double y, int * quo);
```

Argument

x

A real number.

y

A real number.

quo

Description

The remquo(), remquof(), and remquol() functions compute the same remainder as the remainder(), remainderf(), and remainderl() functions, respectively. In the object pointed to by *quo*, they store a value whose sign is the sign of *x*/ *y* and whose magnitude is congruent modulo 2*n* to the magnitude of the integral quotient of *x*/ *y*, where *n* is an implementation-defined integer greater than or equal to 3.

The `remquo` functions are functionally equivalent to the `remainder` functions.

Return Values

r

Upon successful completion, these functions return the floating-point remainder *r* = *x* - *ny* when *y* is nonzero.

Nan

If *x* or *y* is Nan.

remove

remove — Deletes a file.

Format

```
#include <stdio.h>
int remove (const char *file_spec);
```

Argument

file_spec

A pointer to the string that is an OpenVMS or a UNIX style file specification. The file specification can include a wildcard in its version number. So, for example, files of the form *filename*.txt;* can be deleted.

Description

If you specify a directory in the filename and it is a search list that contains an error, VSI C for OpenVMS systems interprets it as a file error.

Note

The DECC$ALLOW_REMOVE_OPEN_FILES feature logical controls the behavior of the `remove` function on open files. Ordinarily, the operation fails. However, POSIX conformance dictates that the operation succeed.

With DECC$ALLOW_REMOVE_OPEN_FILES enabled, this POSIX conformant behavior is achieved.

When `remove` is used to delete a symbolic link, the link itself is deleted, not the file to which it refers.

The `remove` and `delete` functions are functionally equivalent in the C RTL.

See also `delete`.

Return Values

0

Indicates success.

nonzero value

Indicates failure.

rename

rename — 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

A pointer to a string that is the existing name of the file to be renamed.

new_file_spec

A pointer to a string that is to be the new name of the file.

Description

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

If the *new_file_spec* does not contain a file extension, the file extension of *old_file_spec* is used. To rename a file to have no file extension, *new_file_spec* must contain a period (.) For example, the following renames SYS$DISK:[]FILE.DAT to SYS$DISK:[]FILE1.DAT:

```
rename("file.dat", "file1");
```

However, the following renames SYS$DISK:[]FILE.DAT to SYS$DISK:[]FILE1:

```
rename("file.dat", "file1.");
```

Note

Because the `rename` function does special processing of the file extension, the caller must be careful when specifying the name of the renamed file in a call to a C Run-Time Library function that accepts a file-name argument. For example, after the following call to the `rename` function, the new file should be opened as `fopen("bar.dat",...)`:

```
rename("foo.dat", "bar");
```

The `rename` function is affected by the setting of the DECC$RENAME_NO_INHERIT and DECC$RENAME_ALLOW_DIR feature logicals as follows:

- DECC$RENAME_NO_INHERIT provides more UNIX compliant behavior in `rename`, and affects whether or not the new name for the file inherits anything (like file type) from the old name or must be specified completely.
- DECC$RENAME_ALLOW_DIR lets you choose between the previous OpenVMS behavior of allowing the renaming of a file from one directory to another, or the more UNIX compliant behavior of not allowing the renaming of a file to a directory.

See the DECC$RENAME_NO_INHERIT and DECC$RENAME_ALLOW_DIR descriptions in Section 1.5 for more information.

Return Values

0

Indicates success.

-1

Indicates failure. The function sets `errno` to one of the following values:

- EISDIR– The new argument points to a directory, and the old argument points to a file that is not a directory.
- EEXIST– The new argument points to a directory that already exists.
- ENOTDIR– The old argument names a directory, and new argument names a non-directory file.
- ENOENT – The old argument points to a file, directory, or device that does not exist.

  Or the new argument points to a nonexisting directory path or device.

rewind

rewind — Sets the file to its beginning.

Format

```
#include <stdio.h>
void rewind (FILE *file_ptr); (ISO POSIX-1)
int rewind (FILE *file_ptr); (VSI C Extension)
```

Argument

file_ptr

A file pointer.

Description

The `rewind` function is equivalent to `fseek (file_ptr, 0, SEEK_SET)`. You can use the `rewind` function with either record or stream files.

A successful call to `rewind` clears the error indicator for the file.

The ANSI C standard defines `rewind` as not returning a value; therefore, the function prototype for `rewind` is declared with a return type of `void`. However, since a `rewind` can fail, and since previous versions of the C RTL have declared `rewind` to return an `int`, the code for `rewind` does return 0 on success and -1 on failure.

See also `fseek`.

rewinddir

rewinddir — Resets the position of the specified directory stream to the beginning of a directory.

Format

```
#include <dirent.h>
void rewinddir (DIR *dir_pointer);
```

Argument

dir_pointer

A pointer to the `dir` structure of an open directory.

Description

The `rewinddir` function resets the position of the specified directory stream to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, the same as using the `opendir` function. If the *dir_pointer* argument does not refer to a directory stream, the effect is undefined.

The type `DIR`, defined in the `<dirent.h>` header file, represents a directory stream. A directory stream is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files.

See also `opendir`.

rindex

rindex — Searches for a character in a string.

Format

```
#include <strings.h>
char *rindex (const char *s, int c);
```

Function Variants

The `rindex` function has variants named `_rindex32` and `_rindex64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s

The string to search.

c

The character to search for.

Description

The `rindex` function is identical to the `strchr` function, and is provided for compatibility with some UNIX implementations.

rint

rint — Rounds its argument to an integral value according to the current IEEE rounding direction specified by the user.

Format

```
#include <math.h>
double rint (double x);
float rintf (float x,);
long double rintl (long double x);
```

Argument

x

A real number.

Description

The `rint` functions return the nearest integral value to *x* in the direction of the current IEEE rounding mode specified on the /ROUNDING_MODE command-line qualifier.

If the current rounding mode rounds toward negative Infinity, then `rint` is identical to `floor`. If the current rounding mode rounds toward positive Infinity, then `rint` is identical to `ceil`.

If |*x*| = Infinity, `rint` returns *x*.

Return Values

n

The nearest integral value to *x* in the direction of the current IEEE rounding mode.

NaN

x is NaN; `errno` is set to EDOM.

rmdir

rmdir — Removes a directory file.

Format

```
#include <unistd.h>
int rmdir (const char *path);
```

Argument

path

A directory pathname.

Description

The `rmdir` function removes a directory file whose name is specified in the *path* argument. The directory is removed only if it is empty.

If *path* names a symbolic link, then `rmdir` fails and sets `errno` to ENOTDIR.

Restriction

When using OpenVMS format names, the *path* argument must be in the form *directory.dir*.

Return Values

0

Indicates success.

-1

An error occurred; `errno` is set to indicate the error.

round

round — Rounds to the nearest integer value in floating-point format, rounding halfway cases away from zero.

Format

```
#include <math.h>
double round (double x);
float roundf (float x);
long double roundl (long double x);
```

Argument

x

Value to round.

Description

The `round` functions round *x* to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

Return Value

n

On success, the function returns the rounded integer value.

sbrk

sbrk — Determines the lowest virtual address that is not used with the program.

Format

```
#include <unistd.h>
void *sbrk (long int incr);
```

Argument

incr

The number of bytes to add to the current break address.

Description

The `sbrk` function adds the number of bytes specified by its argument to the current break address and returns the old break address.

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

`sbrk(0)` returns the current break address.

Return Values

x

The old break address.

(`void` *)(-1)

Indicates that the program is requesting too much memory.

Restriction

Unlike other C library implementations, the C RTL memory allocation functions (such as `malloc`) do not rely on `brk` or `sbrk` to manage the program heap space. Consequently, on OpenVMS systems, calling `brk` or `sbrk` can interfere with memory allocation routines. The `brk` and `sbrk` functions are provided only for compatibility purposes.

scalb

scalb — Returns the exponent of a floating-point number.

Format

```
#include <math.h>
double scalb (double x, double n);
float scalbf (float x, float n);
```

```
long double scalbl (long double x, long double n);
```

Arguments

x

A nonzero floating-point number.

n

An integer.

Description

The `scalb` functions return *x*(2** *n*) for integer *n*.

Return Values

x

On successful completion, *x*(2** *n*) is returned.

±HUGE_VAL

On overflow, `scalb` returns ±HUGE_VAL (according to the sign of *x*) and sets `errno` to ERANGE.

0

Underflow occurred; `errno` is set to ERANGE.

x

x is ±Infinity.

NaN

x or *n* is NaN; `errno` is set to EDOM.

scalbln

scalbln — Returns the exponent of a floating-point number using the floating-point base exponent (`long int`).

Format

```
#include <math.h>
double scalbln (double x, long int n);
float scalblnf (float x, long int n);
long double scalblnl (long double x, long int n);
```

Arguments

x

A nonzero floating-point number.

n

Value of the exponent (`long int`).

Description

These functions multiply their first argument *x* by FLT_RADIX (on binary systems, it has a value of 2) to the power of *n*, which is:

```
x * FLT_RADIX ** n
```

The definition of FLT_RADIX can be obtained by including `<float.h>`.

Return Values

x

On success, `x * FLT_RADIX ** n` is returned.

NaN

If *x* is NaN.

±0

If *x* is ±0.

If the result underflows, a *range error* occurs, and the functions return ±0 with a sign the same as *x*.

±Infinity

If *x* is ±Infinity.

±HUGE_VAL, HUGE_VALF, or HUGE_VALL

If the result overflows, a *range error* occurs, and the functions return ±HUGE_VAL, HUGE_VALF, or HUGE_VALL, respectively, with a sign the same as *x*.

scalbn

scalbn — Returns the exponent of a floating-point number using the floating-point base exponent.

Format

```
#include <math.h>
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
```

Arguments

x

A nonzero floating-point number.

n

Value of the exponent (`int`).

Description

These functions multiply their first argument *x* by FLT_RADIX (on binary systems, it has a value of 2) to the power of *n*, which is:

```
x * FLT_RADIX ** n
```

The definition of FLT_RADIX can be obtained by including `<float.h>`.

Return Values

x

On success, `x * FLT_RADIX ** n` is returned.

NaN

If *x* is NaN.

±0

If *x* is ±0.

If the result underflows, a *range error* occurs, and the functions return ±0 with a sign the same as *x*.

±Infinity

If *x* is ±Infinity.

±HUGE_VAL, HUGE_VALF, or HUGE_VALL

If the result overflows, a *range error* occurs, and the functions return ±HUGE_VAL, HUGE_VALF, or HUGE_VALL, respectively, with a sign the same as *x*.

scanf

scanf — Performs formatted input from the standard input (`stdin`), interpreting it according to the format specification. See Chapter 2 for information on format specifiers.

Format

```
#include <stdio.h>
int scanf (const char *format_spec, ...);
```

Arguments

format_spec

Pointer to a string containing the format specification. The format specification consists of characters to be taken literally from the input or converted and placed in memory at the specified input sources. For a list of conversion characters, see Chapter 2.

. . .

Optional expressions that are pointers to objects whose resultant types correspond to conversion specifications given in the format specification.

If no conversion specifications are given, you can omit these input pointers. Otherwise, the function call must have at least as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input pointers.

Conversion specifications are matched to input sources in left-to-right order. Excess input pointers, if any, are ignored.

Return Values

x

The number of successfully matched and assigned input items.

EOF

Indicates that a read error occurred prior to any successful conversions. The function sets `errno`. For a list of `errno` values set by this function, see `fscanf`.

[w]scanw

[w]scanw — Perform a `scanf` on the window. The `scanw` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int scanw (char *format_spec, ...);
int wscanw (WINDOW *win, char *format_spec, ...);
```

Arguments

win

A pointer to the window.

format_spec

A pointer to the format specification string.

. . .

Optional expressions that are pointers to objects whose resultant types correspond to conversion specifications given in the format specification. If no conversion specifications are given, you may omit these input pointers.

Otherwise, the function call must have at least as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input pointers.

Conversion specifications are matched to input sources in left-to-right order. Excess input pointers, if any, are ignored.

Description

The formatting specification (*format_spec*) and the other arguments are identical to those used with the `scanf` function.

The `scanw` and `wscanw` functions accept, format, and return a line of text from the terminal screen. For more information, see the `scrollok` and `scanf` functions.

Return Values

OK

Indicates success.

ERR

Indicates that the function makes the screen scroll illegally or that the scan was unsuccessful.

scroll

scroll — Moves all the lines on the window up one line. The top line scrolls off the window and the bottom line becomes blank.

Format

```
#include <curses.h>
int scroll (WINDOW *win);
```

Argument

win

A pointer to the window.

Return Values

OK

Indicates success.

ERR

Indicates an error.

scrollok

scrollok — Sets the scroll flag for the specified window.

Format

```
#include <curses.h>
scrollok (WINDOW *win, bool boolf);
```

Arguments

win

A pointer to the window.

boolf

A Boolean TRUE or FALSE value. If *boolf* is FALSE, scrolling is not allowed. This is the default setting. The *bool* type is defined in the `<curses.h>` header file as follows:

```
#define bool int
```

seed48

seed48 — Initializes a 48-bit random-number generator.

Format

```
#include <stdlib.h>
unsigned short *seed48 (unsigned short seed_16v[3]);
```

Argument

seed_16v

An array of three `unsigned short int`s that form a 48-bit seed value.

Description

The `seed48` function initializes the random-number generator. You can use this function in your program before calling the `drand48`, `lrand48`, or `mrand48` functions. (Although it is not recommended practice, constant default initializer values are supplied automatically if you call `drand48`, `lrand48`, or `mrand48` without calling an initialization function).

The `seed48` function works by generating a sequence of 48-bit integer values, X*i*, according to the linear congruential formula:

$$X_{n+1} = (aX_n + c) \bmod m \qquad n > 0$$

The argument *m* equals 2^{48}, so 48-bit integer arithmetic is performed. Unless you invoke the `lcong48` function, the multiplier value *a* and the addend value *c* are:

$$a = 5DEECE66D_{16} = 273673163155_8$$
$$c = B_{16} = 13_8$$

The initializer function `seed48`:

- Sets the value of X*i* to the 48-bit value specified in the array pointed to by *seed_16v*.

- Returns a pointer to a 48-bit internal buffer that contains the previous value of X*i*, used only by `seed48`.

The returned pointer allows you to restart the pseudorandom sequence at a given point. Use the pointer to copy the previous X*i* value into a temporary array. To resume where the original sequence left off, you can call `seed48` with a pointer to this array.

See also `drand48`, `lrand48`, and `mrand48`.

Return Value

x

A pointer to a 48-bit internal buffer.

seekdir

seekdir — Sets the position of a directory stream.

Format

```
#include <dirent.h>
void seekdir (DIR *dir_pointer, long int location);
```

Arguments

dir_pointer

A pointer to the `dir` structure of an open directory.

location

The number of an entry relative to the start of the directory.

Description

The `seekdir` function sets the position of the next `readdir` operation on the directory stream specified by *dir_pointer* to the position specified by *location*. The value of *location* should be returned from an earlier call to *telldir*.

If the value of *location* was not returned by a call to the `telldir` function, or if there was an intervening call to the `rewinddir` function on this directory stream, the effect is unspecified.

The type `DIR`, defined in the `<dirent.h>` header file, represents a directory stream. A directory stream is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files. You can remove files from or add files to a directory asynchronously to the operation of the `readdir` function.

See `readdir`, `rewinddir`, and `telldir`.

sem_close

sem_close — Deallocates the specified named semaphore.

Format

```
#include <semaphore.h>
int sem_close (sem_t *sem);
```

Argument

sem

The semaphore to be closed. Use the *sem* argument returned by the previous call to `sem_open`.

Description

The `sem_close` function makes a semaphore available for reuse by deallocating any system resources allocated for use by the current process for the named semaphore indicated by *sem*.

If the semaphore has not been removed with a call to `sem_unlink`, `sem_close` does not change the current state of the semaphore.

If the semaphore has been removed with a call to `sem_unlink` after the most recent call to `sem_open` with O_CREAT, the semaphore is no longer available after all processes that opened the semaphore close it.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL– The *sem* argument is not a valid semaphore descriptor.
- ENOSYS– The function is not implemented.
- EVMSERR– OpenVMS specific nontranslatable error code.

semctl

semctl — Semaphore control operations.

Format

```
#include <sem.h>
int semctl (int semid, int semnum, int cmd, ...);
```

Argument

semid

A semaphore set identifier, a positive integer. It is created by the `semget` function and used to identify the semaphore set on which to perform the control operation.

semnum

Semaphore number, a non-negative integer. It identifies a semaphore within the semaphore set on which to perform the control operation.

cmd

The control operation to perform on the semaphore.

`...`

Optional fourth argument of type `union semun`, which depends on the control operation requested in *cmd*.

Description

The `semctl` function provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional and depends upon the operation requested. If required, it is of type `union semun`, which is explicitly declared as:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
} arg;
```

The following semaphore control operations as specified by *cmd* are executed with respect to the semaphore specified by *semid* and *semnum*. The level of permission required for each operation is shown with each command. The symbolic names for the values of *cmd* are defined in the `<sem.h>` header:

- GETVAL

  Returns the value of *semval*. Requires read permission.

- SETVAL

  Sets the value of *semval* to *arg.val*, where *arg* is the value of the fourth argument to `semctl`. When this command is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared. Requires alter permission.

- GETPID

  Returns the value of *sempid*; requires read permission.

- GETNCNT

  Returns the value of *semncnt*; requires read permission.

- GETZCNT

  Returns the value of *semzcnt*; requires read permission.

The following values of *cmd* operate on each *semval* in the set of semaphores:

- GETALL

  Returns the value of *semval* for each semaphore in the semaphore set and places it into the array pointed to by *arg.array*, where *arg* is the fourth argument to `semctl`; requires read permission.

- SETALL

  Sets the value of *semval* for each semaphore in the semaphore set according to the array pointed to by *arg.array*, where *arg* is the fourth argument to `semctl`. When this command is successfully executed, the *semadj* values corresponding to each specified semaphore in all processes are cleared. Requires alter permission.

The following values of *cmd* are also available:

- IPC_STAT

  Places the current value of each member of the *semid_ds* data structure associated with *semid* into the structure pointed to by *arg.buf*, where *arg* is the fourth argument to `semctl`. The contents of this structure are defined in `<sem.h>`. Requires read permission.

- IPC_SET

  Sets the value of the following members of the *semid_ds* data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*, where *arg* is the fourth argument to `semctl`:

  ```
  sem_perm.uid sem_perm.gid sem_perm.mode
  ```

  The mode bits specified in *The Open Group Base Specifications* IPC General Description section are copied into the corresponding bits of the *sem_perm.mode* associated with *semid*. The stored values of any other bits are unspecified.

  This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of *sem_perm.cuid* or *sem_perm.uid* in the *semid_ds* data structure associated with *semid*.

- IPC_RMID

  Removes the semaphore identifier specified by *semid* from the system and destroys the set of semaphores and *semid_ds* data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of *sem_perm.cuid* or *sem_perm.uid* in the *semid_ds* data structure associated with *semid*.

Return Values

n or 0

Upon successful completion, the value returned by the function depends on *cmd* as follows:

- GETVAL - The value of *semval*
- GETPID - The value of *sempid*
- GETNCNT - The value of *semncnt*
- GETZCNT - The value of *semzcnt*

- All others - 0

-1

Indicates an error. The function sets `errno` to one of the following values:

- EACCES– Operation permission is denied to the calling process.
- EFAULT– The arguments passed to the function are not accessible.
- EINVAL– The value of *semid* is not a valid semaphore identifier, or the value of *semnum* is less than zero or greater than or equal to *sem_nsems*, or the value of *cmd* is not a valid command.
- EPERM– The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of *sem_perm.cuid* or *sem_perm.uid* in the data structure associated with *semid*.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_destroy

sem_destroy — Destroys an unnamed semaphore.

Format

```
#include <semaphore.h>
int sem_destroy (sem_t *sem);
```

Argument

sem

The unnamed semaphore to be destroyed. Use the *sem* argument that was supplied to, and filled in by, the previous call to `sem_init`.

Description

The `sem_destroy` function destroys an unnamed semaphore indicated by *sem*. Only a semaphore created using `sem_init` may be destroyed using `sem_destroy`.

The potential for deadlock exists if a process calls `sem_destroy` for a semaphore while there is a pending `sem_wait`, because a process may be waiting for a poster that has not yet opened the semaphore.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values, without destroying the semaphore indicated by the *sem* argument:

- EINVAL– The *sem* argument is not a valid semaphore.
- ENOSYS– The function is not implemented.
- EVMSERR– OpenVMS specific nontranslatable error code.
- EBUSY– The processes are blocked on the semaphore.

semget

semget — Gets a set of semaphores.

Format

```
#include <sem.h>
int semget (key_t key, int nsems, int semflg);
```

Argument

key

The key for which the associated semaphore identifier is returned.

nsems

Value used to initialize the *sem_nsems* member of the `semid_ds` data structure. See the description.

semflg

Flag used to initialize the low-order 9 bits of the *sem_perm.mode* member of the `semid_ds` data structure associated with the new semaphore. See the description.

value

The initial value to be given to the semaphore. This argument is used only when the semaphore is being created.

Description

The `semget` function returns the semaphore identifier associated with *key*.

A semaphore identifier with its associated `semid_ds` data structure and its associated set of *nsems* semaphores (see the `<sys/sem.h>` header file) is created for *key* if:

- The *key* argument does not already have a semaphore identifier associated with it and (*semflg* &IPC_CREAT) is nonzero.

When it is created, the `semid_ds` data structure associated with the new semaphore identifier is initialized as follows:

- In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and *sem_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of *sem_perm.mode* are set equal to the low-order 9 bits of the *semflg* argument.

- The variable *sem_nsems* is set equal to the value of the *nsems* argument.
- The variable *sem_otime* is set equal to 0 and the variable *sem_ctime* is set equal to the current time.
- The data structure associated with each semaphore in the set does not need to be initialized. You can use the `semctl` function with the command SETVAL or SETALL to initialize each semaphore.

Note

The *key* argument value IPC_PRIVATE is not supported.

Return Values

n

Successful completion. The function returns a non-negative integer semaphore identifier.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EACCES– A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* was not granted.
- EEXIST– A semaphore identifier exists for *key* but ((semflg&IPC_CREAT) && (semflg &IPC_EXCL)) is nonzero.
- EFAULT– The arguments passed to the function are not accessible.
- EINVAL– The value of *nsems* is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to 0.
- ENOENT– A semaphore identifier does not exist for *key* and (*semflg* &IPC_CREAT) is equal to 0.
- ENOSPC– A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system-wide will be exceeded.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_getvalue

sem_getvalue — Gets the value of a specified semaphore.

Format

```
#include <semaphore.h>
int sem_getvalue (sem_t *sem, int *sval);
```

Argument

sem

The semaphore for which a value is to be returned.

sval

The location to be updated with the value of the semaphore indicated by the *sem* argument.

Description

The `sem_getvalue` function updates a location referenced by the *sval* argument with the value of semaphore *sem*. The updated value represents an actual semaphore value that occurred during the call, but may not be the actual value of the semaphore at the time that the value is returned to the calling process.

If the semaphore is locked, the value returned will either be zero or a negative number indicating the number of processes waiting for the semaphore at some time during the call.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL– The *sem* argument is not a valid semaphore.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_init

sem_init — Initializes an unnamed semaphore.

Format

```
#include <semaphore.h>
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

Argument

sem

The location to receive the descriptor of the initialized semaphore.

pshared

A value indicating whether the semaphore should be sharable between the creating process and its descendants (nonzero value) or not (zero).

Note

The value for pshared must be zero between threads because this release does not support unnamed semaphores to be shared across processes.

value

The initial value to be given to the semaphore.

Description

The `sem_init` function creates a new counting semaphore with a specific value. A semaphore is used to limit access to a critical resource. When a process requires access to the resource without interference from other processes, it attempts to establish a connection with the associated semaphore. If the semaphore value is greater than zero, the connection is established and the semaphore value is decremented by one. If the semaphore value is less than or equal to zero, the process attempting to access the resource is blocked and must wait for another process to release the semaphore and increment the semaphore value.

The `sem_init` function establishes a connection between an unnamed semaphore and a process; the `sem_wait` and `sem_trywait` functions lock the semaphore; and the `sem_post` function unlocks the semaphore. Use the `sem_destroy` function to deallocate system resources allocated to the process for use with the semaphore. You can use the `sem_getvalue` function to obtain the value of a semaphore.

A semaphore created by a call to the `sem_init` function remains valid until the semaphore is removed by a call to the `sem_destroy` function.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL– The value argument exceeds {SEM_VALUE_MAX}.
- ENOSYS– The function is not implemented.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_open

sem_open — Opens/creates a named semaphore for use by a process.

Format

```
#include <semaphore.h>
sem_t *sem_open (const char *name, int ooption...);
```

Argument

name

a string naming the semaphore object.

ooption

Specifies whether the semaphore is to be created (O_CREAT option bit set) or only opened (O_CREAT option bit clear). If O_CREAT is set, the O_EXCL option bit may additionally be set to specify that the call should fail if a semaphore of the same name already exists. The O_CREAT and O_EXCL options are defined in the `<fcntl.h>` header file.

mode

The semaphore's permission bits. This argument is used only when the semaphore is being created.

value

The initial value to be given to the semaphore. This argument is used only when the semaphore is being created.

Description

Use the `sem_open` function to establish the connection between a named semaphore and a process. Subsequently, the calling process can reference the semaphore by using the address returned from the call. The semaphore is available in subsequent calls to `sem_wait`, `sem_trywait`, `sem_post`, and `sem_getvalue` functions. The semaphore remains usable by the process until the semaphore is closed by a successful call to the `sem_close` function.

The O_CREAT option bit in the *ooption* parameter controls whether the semaphore is created or only opened by the call to `sem_open`.

A created semaphore's user ID is set to the user ID of the calling process and its group ID is set to a system default group or to the group ID of the process. The semaphore's permission bits are set to the value of the mode argument, except for those set in the file mode creation mask of the process.

After a semaphore is created, other processes can open the semaphore by calling `sem_open` with the same value for the name argument.

Return Values

sem

Successful completion. The function opens the semaphore and returns the semaphore's descriptor.

sem_failed

Indicates an error. The function sets `errno` to one of the following values:

- EACCES – The named semaphore exists and the permissions specified by *ooption* are denied, or the named semaphore does not exist and the permissions specified by *ooption* are denied.
- EEXIST – O_CREAT and O_EXCL are set, and the named semaphore already exists.
- EINVAL – The `sem_open` operation is not supported for the given name. Or, O_CREAT was specified in *ooption* and *value* was greater than {SEM_VALUE_MAX}.
- EMFILE – Too many semaphore descriptors or file descriptors are currently in use by this process.

- ENAMETOOLONG – The length of the *name* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.
- ENFILE – Too many semaphores are currently open in the system. ENOENT O_CREAT is not set, and the named semaphore does not exist.
- ENOSPC – Insufficient space exists for the creation of a new named semaphore.
- EVMSERR – OpenVMS specific nontranslatable error code.

semop

semop — Performs operations on semaphores in a semaphore set.

Format

```
#include <sem.h>
int semop (int semid, struct sembuf *sops, size_t nsops);
```

Argument

semid

Semaphore set identifier.

sops

Pointer to a user-defined array of semaphore operation (`sembuf`) structures.

nsops

Number of `sembuf` structures in the `sops` array.

Description

The `semop` function performs operations on semaphores in the semaphore set specified by *semid*. These operations are supplied in a user-defined array of semaphore operation `sembuf` structures specified by *sops*. Each `sembuf` structure includes the following member variables:

```
struct sembuf {            /* semaphore operation structure */
  unsigned short sem_num;  /* semaphore number */
           short sem_op;   /* semaphore operation */
           short sem_flg;  /* operation flags SEM_UNDO and IPC_NOWAIT */
```

Each semaphore operation specified by the *sem_op* variable is performed on the corresponding semaphore specified by the *semid* function argument and the *sem_num* variable.

The *sem_op* variable specifies one of three semaphore operations:

1. If *sem_op* is a negative integer and the calling process has change permission, one of the following occurs:
   - If *semval* (see `<sem.h>`) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* &SEM_UNDO) is non-zero,

the absolute value of *sem_op* is added to the calling process' *semadj* value for the specified semaphore.

- If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is nonzero, `semop` returns immediately.
- If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is 0, `semop` increments the *semncnt* associated with the specified semaphore and suspends execution of the calling thread until one of the following conditions occurs:
  - The value of *semval* becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from *semval* and, if (*sem_flg* &SEM_UNDO) is nonzero, the absolute value of *sem_op* is added to the calling process' *semadj* value for the specified semaphore.
  - The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, `errno` is set equal to EIDRM and -1 is returned.
  - The calling thread receives a signal that is to be intercepted. When this occurs, the value of *semncnt* associated with the specified semaphore is decremented, and the calling thread is resumes execution in the manner prescribed in `sigaction`.

2. If *sem_op* is a positive integer and the calling process has change permission, the value of *sem_op* is added to *semval* and, if (*sem_flg* &SEM_UNDO) is nonzero, the value of *sem_op* is subtracted from the calling process' *semadj* value for the specified semaphore.
3. If *sem_op* is 0 and the calling process has read permission, one of the following occurs:
   - If *semval* is 0, `semop` returns immediately.
   - If *semval* is nonzero and (*sem_flg* &IPC_NOWAIT) is nonzero, `semop` returns immediately.
   - If *semval* is nonzero and (*sem_flg* &IPC_NOWAIT) is 0, `semop` increments the *semzcnt* associated with the specified semaphore and suspends execution of the calling thread until one of the following occurs:
     - The value of *semval* becomes 0, at which time the value of *semzcnt* associated with the specified semaphore is decremented.
     - The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, `errno` is set equal to EIDRM and -1 is returned.
     - The calling thread receives a signal that is to be intercepted. When this occurs, the value of *semzcnt* associated with the specified semaphore is decremented, and the calling thread resumes execution in the manner prescribed in `sigaction`.

On successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- E2BIG– The value of *nsops* is greater than the system-imposed maximum.
- EACCES– Operation permission is denied to the calling process.
- EAGAIN– The operation would result in suspension of the calling process but (*sem_flg* &IPC_NOWAIT) is nonzero.
- EFAULT– The arguments passed to the function are not accessible.
- EFBIG– The value of *sem_num* is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.
- EIDRM– The semaphore identifier *semid* is removed from the system.
- EINVAL– The value of *semid* is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the system-imposed limit.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_post

sem_post — Unlocks a semaphore.

Format

```
#include <semaphore.h>
int sem_post (sem_t *sem);
```

Argument

sem

The semaphore to be unlocked.

Description

The `sem_post` function unlocks the specified semaphore by performing the semaphore unlock operation on that semaphore. The appropriate function (`sem_open` for named semaphores or `sem_init` for unnamed semaphores) must be called for a semaphore before you can call the locking and unlocking functions, `sem_wait`, `sem_trywait`, and `sem_post`.

If the semaphore value after a `sem_post` function is positive, no processes were blocked waiting for the semaphore to be unlocked; the semaphore value is incremented. If the semaphore value after a `sem_post` function is zero, one of the processes blocked waiting for the semaphore is allowed to return successfully from its call to `sem_wait`.

If more than one process is blocked while waiting for the semaphore, only one process is unblocked and the state of the semaphore remains unchanged when the `sem_post` function returns. The process to be unblocked is selected according to the scheduling policies and priorities of all blocked processes. If the

scheduling policy is SCHED_FIFO or SCHED_RR, the highest-priority waiting process is unblocked. If more than one process of that priority is blocked, then the process that has waited the longest is unblocked.

The `sem_post` function can be called from a signal-catching function.

Return Values

0

Successful completion. The `sem_post` function performs a semaphore unlock operation, unblocking a process.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL– The *sem* argument is not a valid semaphore.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_timedwait

sem_timedwait — Performs a semaphore lock.

Format

```
#include <semaphore.h>
#include <time.h>
 int sem_timedwait (sem_t *sem, const struct timespec *abs_timeout);
```

Argument

sem

The semaphore to be locked.

abs_timeout

The absolute time after which the timeout expires.

Description

The `sem_timedwait` function locks the semaphore referenced by *sem* as in the `sem_wait` function. But if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a `sem_post` function, this wait terminates when the specified timeout expires.

The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*, or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

The function will not fail with a timeout if the semaphore can be locked immediately. The validity of *abs_timeout* does not need to be checked if the semaphore can be locked immediately.

Return Values

0

Successful completion. The function executes the semaphore lock operation.

-1

Indicates an error. The function sets `errno` to one of the following values:

- ETIMEDOUT– The semaphore could not be locked before the specified timeout expired.
- EINVAL– The *sem* argument does not refer to a valid semaphore. Or the process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_trywait

sem_trywait — Conditionally performs a semaphore lock.

Format

```
#include <semaphore.h>
int sem_trywait (sem_t *sem);
```

Argument

sem

The semaphore to be locked.

Description

The `sem_trywait` function locks a semaphore only if the semaphore is currently not locked. If the semaphore value is zero, the `sem_trywait` function returns without locking the semaphore.

The `sem_wait` and `sem_trywait` functions help ensure that the resource associated with the semaphore cannot be accessed by other processes. The semaphore remains locked until the process unlocks it with a call to the `sem_post` function.

Use the `sem_wait` function instead of the `sem_trywait` function if the process should wait for access to the semaphore.

Return Values

0

Successful completion. The function executes the semaphore lock operation.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EAGAIN– The semaphore was already locked and cannot be locked by the `sem_trywait` operation.
- EINVAL– The *sem* argument does not refer to a valid semaphore.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_unlink

sem_unlink — Removes the specified named semaphore.

Format

```
#include <semaphore.h>
int sem_unlink (const char *name);
```

Argument

name

The name of the semaphore to remove.

Description

The `sem_unlink` function removes a semaphore named by the *name* string. If the semaphore is referenced by other processes, `sem_unlink` does not change the state of the semaphore.

If other processes have the semaphore open when `sem_unlink` is called, the semaphore is not destroyed until all references to the semaphore have been destroyed by calls to `sem_close`. The `sem_unlink` function returns immediately; it does not wait until all references have been destroyed.

Calls to `sem_open` to recreate or reconnect to the semaphore refer to a new semaphore after `sem_unlink` is called.

Return Values

0

Successful completion. The function executes the semaphore unlink operation.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EACCESS– Permission is denied to unlink the named semaphore.
- ENAMETOOLONG– The length of the path name exceeds PSEM_MAX_PATHNAME defined in semaphore.h.
- ENOENT– The named semaphore does not exist.
- EVMSERR– OpenVMS specific nontranslatable error code.

sem_wait

sem_wait — Performs a semaphore lock.

Format

```
#include <semaphore.h>
int sem_wait (sem_t *sem);
```

Argument

sem

The semaphore to be locked.

Description

The `sem_wait` function locks the semaphore referenced by sem by performing a semaphore lock operation on it. If the semaphore value is zero, the `sem_wait` function blocks until it either locks the semaphore or is interrupted by a signal.

The `sem_wait` and `sem_trywait` functions help ensure that the resource associated with the semaphore cannot be accessed by other processes. The semaphore remains locked until the process unlocks it with a call to the `sem_post` function.

Use the `sem_wait` function instead of the `sem_trywait` function if the process should wait for access tothe semaphore.

Return Values

0

Successful completion. The function executes the semaphore lock operation.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINTR– A signal interrupted this function.
- EVMSERR– OpenVMS specific nontranslatable error code.

[w]setattr

[w]setattr — Activate the video display attribute *attr* within the window. The `setattr` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int setattr (int attr);
int wsetattr (WINDOW *win, int attr);
```

Arguments

win

A pointer to the window.

attr

One of a set of video display attributes, which are blinking, boldface, reverse video, and underlining, and are represented by the defined constants _BLINK, _BOLD, _REVERSE, and _UNDERLINE, respectively. You can set multiple attributes by separating them with a bitwise OR operator (|) as follows:

```
setattr(_BLINK | _UNDERLINE);
```

Description

The `setattr` and `wsetattr` functions are specific to VSI C for OpenVMS systems and are not portable.

Return Values

OK

Indicates success.

ERR

Indicates an error.

setbuf

setbuf — Associates a new buffer with an input or output file and potentially modifies the buffering behavior.

Format

```
#include <stdio.h>
void setbuf (FILE *file_ptr, char *buffer);
```

Arguments

file_ptr

A file pointer.

buffer

A pointer to a character array or a NULL pointer.

Description

You can use the `setbuf` function after the specified file is opened but before any I/O operations are performed.

If *buffer* is a NULL pointer, then the call is equivalent to a call to `setvbuf` with the same *file_ptr*, a NULL *buffer* pointer, a buffering type of _IONBF (no buffering), and a buffer size of 0.

If *buffer* is not a NULL pointer, then the call is equivalent to a call to `setvbuf` with the same *file_ptr*, the same *buffer* pointer, a buffering type of _IOFBF, and a buffer size given by the value BUFSIZ (defined in `<stdio.h>`). Therefore, use BUFSIZ to allocate the *buffer* argument used in the call to `setbuf`. For example:

```
#include <stdio.h>
    .
    .
    .
char my_buf[BUFSIZ];
    .
    .
    .
setbuf(stdout, my_buf);
    .
    .
    .
```

User programs must not depend on the contents of *buffer* once I/O has been performed on the stream. The C RTL might or might not use *buffer* for any given I/O operation.

The `setbuf` function originally allowed programmers to substitute larger buffers in place of the system default buffers in obsolete versions of UNIX. The large default buffer sizes in modern implementations of C make the use of this function unnecessary most of the time. The `setbuf` function is retained in the ANSI C standard for compatibility with old programs. New programs should use `setvbuf` instead, because it allows the programmer to bind the buffer size at run time instead of compile time, and it returns a result value that can be tested.

The `setbuf` function now takes 64-bit arguments. However, the *buffer* parameter must contain a 32-bit memory buffer, so when compiling the application with /POINTER=64 or /POINTER=LONG, `malloc32` must be used to allocate the *buffer*.

setenv

setenv — Inserts or resets the environment variable specified by *name* in the current environment list.

Format

```
#include <stdlib.h>
int setenv (const char *name, const char *value, int overwrite);
```

Arguments

name

A variable name in the environment variable list.

value

The value for the environment variable.

overwrite

A value of 0 or 1 indicating whether to reset the environment variable, if it exists.

Description

The `setenv` function inserts or resets the environment variable *name* in the current environment list. If the variable *name* does not exist in the list, it is inserted with the *value* argument. If the variable does exist, the *overwrite* argument is tested. When the *overwrite* argument value is:

- 0 then the variable is not reset.
- 1 then the variable is reset to *value*.

Note

Do not use the `setenv`, `getenv`, and `putenv` functions to manipulate symbols and logicals. Instead, use the OpenVMS library calls `lib$set_logical`, `lib$get_logical`, `lib$set_symbol`, and `lib$get_symbol`. The * `env` functions deliberately provide UNIX behavior, and are not a substitute for these OpenVMS runtime library calls.

OpenVMS DCL symbols, not logical names, are the closest analog to environment variables on UNIX systems. While `getenv` is a mechanism to retrieve either a logical name or a symbol, it maintains an internal cache of values for use with `setenv` and subsequent `getenv` calls. The `setenv` function does not write or create DCL symbols or OpenVMS logical names.

This is consistent with UNIX behavior. On UNIX systems, `setenv` does not change or create any symbols that will be visible in the shell after the program exits.

Return Values

0

Indicates success.

-1

Indicates an error. `errno` is set to ENOMEM – Not enough memory available to expand the environment list.

seteuid

seteuid — Sets the process's effective user ID.

Format

```
#include <unistd.h>
int seteuid (uid_t euid);
```

Argument

euid

The value to which you want the effective user ID set.

Description

If the process has the IMPERSONATE privilege, the `seteuid` function sets the process's effective user ID.

An unprivileged process can set the effective user ID only if the *euid* argument is equal to either the real, effective, or saved user ID of the process.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

See also `getuid` to know how UIC is represented.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL– The value of the *euid* argument is invalid and not supported.
- EPERM– The process does not have the IMPERSONATE privilege, and *euid* does not match the real user ID or the saved set-user-ID.

setgid

setgid — With POSIX IDs disabled, `setgid` is implemented for program portability and serves no function. It returns 0 (to indicate success). With POSIX IDs enabled, `setgid` sets the group IDs.

Format

```
#include <types.h>
#include <unistd.h>
int setgid (__gid_t gid); (_DECC_V4_SOURCE)
int setgid (gid_t gid); (not _DECC_V4_SOURCE)
```

Argument

gid

The value to which you want the group IDs set.

Description

The `setgid` function can be used with POSIX style identifiers enabled or disabled.

POSIX style IDs are supported on OpenVMS Version 7.3-2 and higher.

With POSIX IDs disabled, the `setgid` function is implemented for program portability and serves no function. It returns 0 (to indicate success).

With POSIX style IDs enabled:

- If the process has the IMPERSONATE privilege, the `setgid` function sets the real group ID, effective group ID, and the saved set-group-ID to *gid*.
- If the process does not have appropriate privileges but *gid* is equal to the real group ID or to the saved set-group-ID, then the `setgid` function sets the effective group ID to *gid*. The real group ID and saved set-group-ID remain unchanged.
- Any supplementary group IDs of the calling process remain unchanged.

To enable/disable POSIX style IDs, see Section 1.6.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL– The value of the *gid* argument is invalid and not supported by the implementation.
- EPERM– The process does not have appropriate privileges and *gid* does not match the real group ID or the saved set-group-ID.

setgrent

setgrent — Rewinds the group database.

Format

```
#include <grp.h>
void setgrent (void);
```

Description

The `setgrent` function effectively rewinds the group database to allow repeated searches.

This function is always successful. No value is returned, and `errno` is not set.

setitimer

setitimer — Sets the value of interval timers.

Format

```
#include <time.h>
int setitimer (int which, struct itimerval *value,
struct itimerval *ovalue);
```

Arguments

which

The type of interval timer. The C RTL only supports ITIMER_REAL.

value

A pointer to an `itimerval` structure whose members specify a timer interval and the time left to the end of the interval.

ovalue

A pointer to an `itimerval` structure whose members specify a current timer interval and the time left to the end of the interval.

Description

The `setitimer` function sets the timer specified by *which* to the value specified by *value*, returning the previous value of the timer if *ovalue* is nonzero.

A timer value is defined by the `itimerval` structure:

```
struct itimerval {
        struct  timeval it_interval;
        struct  timeval it_value;
};
```

The value of the `itimerval` structure members are as follows:

itimerval Member Value	Meaning
`it_interval` = 0	Disables a timer after its next expiration (assumes `it_value` is nonzero).
`it_interval` = nonzero	Specifies a value used in reloading `it_value` when the timer expires.
`it_value` = 0	Disables a timer.
`it_value` = nonzero	Indicates the time to the next timer expiration.

Time values smaller than the resolution of the system clock are rounded up to this resolution.

The `getitimer` function provides one interval timer, defined in the `<time.h>` header file as ITIMER_REAL. This timer decrements in real time. When the timer expires, it delivers a SIGALARM signal.

Note

The interaction between `setitimer` and any of `alarm`, `sleep`, or `usleep` is unspecified.

Return Values

0

Indicates success.

-1

An error occurred; `errno` is set to indicate the error.

setjmp

setjmp — Provides a way to transfer control from a nested series of function invocations back to a predefined point without returning normally. It does not use a series of `return` statements. The `setjmp` function saves the context of the calling function in an environment buffer.

Format

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

Argument

env

The environment buffer, which must be an array of integers long enough to hold the register context of the calling function. The type `jmp_buf` is defined 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 `setjmp` is first called, it returns the value 0. If `longjmp` is then called, naming the same environment as the call to `setjmp`, control is returned to the `setjmp` call as if it had returned normally a second time. The return value of `setjmp` in this second return is the value supplied by you in the `longjmp` call. To preserve the true value of `setjmp`, the function calling `setjmp` must not be called again until the associated `longjmp` is called.

The `setjmp` function preserves the hardware general-purpose registers, and the `longjmp` function restores them. After a `longjmp`, all variables have their values as of the time of the `longjmp` except for local automatic variables not marked `volatile`. These variables have indeterminate values.

The `setjmp` and `longjmp` functions rely on the OpenVMS condition-handling facility to effect a nonlocal go to with a signal handler. The `longjmp` function is implemented by generating a C RTL specified signal that allows the OpenVMS condition-handling facility to unwind back to the desired destination.

The C RTL must be in control of signal handling for any VSI C image. For VSI C to be in control of signal handling, you must establish all exception handlers through a call to the `VAXC$ESTABLISH` function. See Section 4.2.5 and the `VAXC$ESTABLISH` function for more information.

Note

The C RTL provides nonstandard `decc$setjmp` and `decc$fast_longjmp` functions for Alpha and Integrity server systems. To use these nonstandard functions instead of the standard ones, a program must be compiled with _ _FAST_SETJMP or _ _UNIX_SETJMP macros defined.

Unlike the standard `longjmp` function, the `decc$fast_longjmp` function does not convert its second argument from 0 to 1. After a call to `decc$fast_longjmp`, a corresponding `setjmp`

function returns with the exact value of the second argument specified in the `decc$fast_longjmp` call.

Restrictions

You cannot invoke the `longjmp` function from an OpenVMS condition handler. However, you may invoke `longjmp` from a signal handler that has been established for any signal supported by the C RTL, subject to the following nesting restrictions:

- The `longjmp` function will not work if you invoke it from nested signal handlers. The result of the `longjmp` function, when invoked from a signal handler that has been entered as a result of an exception generated in another signal handler, is undefined.
- Do not invoke the `setjmp` function from a signal handler unless the associated `longjmp` is to be issued before the handling of that signal is completed.
- Do not invoke the `longjmp` function from within an exit handler (established with `atexit` or SYS$DCLEXH). Exit handlers are invoked after image tear-down, so the destination address of the `longjmp` no longer exists.
- Invoking `longjmp` from within a signal handler to return to the main thread of execution might leave your program in an inconsistent state. Possible side effects include the inability to perform I/O or to receive any more UNIX signals. Use `siglongjmp` instead.

Return Values

See the Description section.

setkey

setkey — Sets an encoding key for use by the `encrypt` function.

Format

```
#include <unistd.h>
#include <stdlib.h>
void setkey (const char *key;)
```

Argument

key

A character array of length 64 containing 0s and 1s.

Description

The argument of `setkey` is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

No value is returned.

See also `crypt` and `encrypt`.

setlocale

setlocale — Selects the appropriate portion of the program's locale as specified by the *category* and *locale* arguments. You can use this function to change or query one category or the program's entire current locale.

Format

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

Arguments

category

The name of the category. Specify LC_ALL to change or query the entire locale. Other valid category names are:

- LC_COLLATE
- LC_CTYPE
- LC_MESSAGES
- LC_MONETARY
- LC_NUMERIC
- LC_TIME

locale

Pointer to a string that specifies the locale.

Description

The `setlocale` function sets or queries the appropriate portion of the program's locale as specified by the *category* and *locale* arguments. Specifying LC_ALL for the category argument names the entire locale; specifying the other values name only a portion of the program's locale.

The *locale* argument points to a character string that identifies the locale to be used. This argument can be one of the following:

- Name of the public locale

  Specifies the public locale in the following format:

  ```
  language_country.codeset[@modifier]
  ```

  The function searches for the public locale binary file in the location defined by the logical name SYS$I18N_LOCALE. The file type defaults to .LOCALE. The period (.) and at-sign (@) characters in the name are replaced by an underscore (_).

  For example, if the specified name is "zh_CN.dechanzi@radical", the function searches for the SYS$I18N_LOCALE:ZH_CN_DECHANZI_RADICAL.LOCALE binary locale file.

- A file specification

  Specifies the binary locale file. It can be any valid file specification. If either the device or directory is omitted, the function first applies the current caller's device and directory as defaults for any missing component. If the file is not found, the function applies the device and directory defined by the SYS$I18N_LOCALE logical name as defaults. The file type defaults to .LOCALE.

  No wildcards are allowed. The binary locale file cannot reside on a remote node.

- "C"

  Specifies the C locale. If a program does not call `setlocale`, the C locale is the default.

- "POSIX"

  This is the same as the C locale.

- ""

  Specifies that the locale is initialized from the setting of the international environment logical names. The function checks the following logical names in the order shown until it finds a logical that is defined:

  1. LC_ALL
  2. Logical names corresponding to the category. For example, if LC_NUMERIC is specified as the category, then the first logical name that `setlocale` checks is LC_NUMERIC.
  3. LANG
  4. SYS$LC_ALL
  5. The system default for the category, which is defined by the SYS$LC_* logical names. For example, the default for the LC_NUMERIC category is defined by the SYS$LC_NUMERIC logical name.
  6. SYS$LANG

     If none of the logical names is defined, the C locale is used as the default. The SYS$LC_* logical names are set up at the system startup time.

  Like the *locale* argument, the equivalence name of the international environment logical name can be either the name of the public locale or the file specification. The `setlocale` function treats this equivalence name as if it were specified as the *locale* argument.

- NULL

  Causes `setlocale` to query the current locale. The function returns a pointer to a string describing the portion of the program's locale associated with *category*. Specifying the LC_ALL category returns the string describing the entire locale. The locale is not changed.

- The string returned from the previous call to `setlocale`

  Causes the function to restore the portion of the program's locale associated with *category*. If the string contains the description of the entire locale, the part of the string corresponding to *category* is used. If the string describes the portion of the program's locale for a single category, this locale is

used. For example, this means that you can use the string returned from the call `setlocale` with the LC_COLLATE category to set the same locale for the LC_MESSAGES category.

If the specified locale is available, then `setlocale` returns a pointer to the string that describes the portion of the program's locale associated with *category*. For the LC_ALL category, the returned string describes the entire program's locale. If an error occurs, a NULL pointer is returned and the program's locale is not changed.

Subsequent calls to `setlocale` overwrite the returned string. If that part of the locale needs to be restored, the program should save the string. The calling program should make no assumptions about the format or length of the returned string.

Return Values

x

Pointer to a string describing the locale.

NULL

Indicates an error occurred; `errno` is set.

Example

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

/* This program calls setlocale() three times. The second call  */
/* is for a nonexistent locale. The third call is for an        */
/* existing file that is not a locale file.                     */

main()
{
    char *ret_str;

    errno = 0;
    printf("setlocale (LC_ALL, \"POSIX\")");
    ret_str = (char *) setlocale(LC_ALL, "POSIX");

    if (ret_str == NULL)
        perror("setlocale error");
    else
        printf(" call was successful\n");

    errno = 0;
    printf("\n\nsetlocale (LC_ALL, \"junk.junk_codeset\")");
    ret_str = (char *) setlocale(LC_ALL, "junk.junk_codeset");

    if (ret_str == NULL)
        perror(" returned error");
    else
        printf(" call was successful\n");

    errno = 0;
    printf("\n\nsetlocale (LC_ALL, \"sys$login:login.com\")");
```

```
    ret_str = (char *) setlocale(LC_ALL, "sys$login:login.com");

    if (ret_str == NULL)
        perror(" returned error");
    else
        printf(" call was successful\n");
}
```

Running the example program produces the following result:

```
setlocale (LC_ALL, "POSIX") call was successful

setlocale (LC_ALL, "junk.junk_codeset")
returned error: no such file or directory

setlocale (LC_ALL, "sys$login:login.com")
returned error: nontranslatable vms error code: 0x35C07C
%c-f-localebad, not a locale file
```

setpgid

setpgid — Sets the process group ID for job control.

Format

```
#include <unistd.h>
int setpgid (pid_t pid, pid_t pgid);
```

Arguments

pid

The process ID for which the process group ID is to be set.

pgid

The value to which the process group ID is set.

Description

The `setpgid` function is used either to join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader will not change.

Upon successful completion, the process group ID of the process with a process ID of *pid* is set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process is used. Also, if *pgid* is 0, the process group ID of the indicated process is used.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EACCES– The value of the *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the `exec` functions.
- EINVAL– The value of the *pgid* argument is less than 0, or is not a value supported by the implementation.
- EPERM– The process indicated by the *pid* argument is a session leader. The value of the *pid* argument matches the process ID of a child process of the calling process, and the child process is not in the same session as the calling process. The value of the *pgid* argument is valid but does not match the process ID of the process indicated by the *pid* argument, and there is no process with a process group ID that matches the value of the *pgid* argument in the same session as the calling process.
- ESRCH– The value of the *pid* argument does not match the process ID of the calling process or of a child process of the calling process.

setpgrp

setpgrp — Sets the process group ID.

Format

```
#include <unistd.h>
pid_t setpgrp (void);
```

Description

If the calling process is not already a session leader, `setpgrp` sets the process group ID of the calling process to the process ID of the calling process. If `setpgrp` creates a new session, then the new session has no controlling terminal.

The `setpgrp` function has no effect when the calling process is a session leader.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

Return Value

x

The process group ID of the calling process.

setpwent

setpwent — Rewinds the user database.

Format

```
#include <pwd.h>
```

```
void setpwent (void);
```

Description

The `setpwent` function effectively rewinds the user database to allow repeated searches.

No value is returned, but `errno` is set to EIO if an I/O error occurred.

See also `getpwent`.

setregid

setregid — Sets the real and effective group IDs.

Format

```
#include <unistd.h>
int setregid (gid_t rgid, gid_t egid);
```

Arguments

rgid

The value to which you want the real group ID set.

egid

The value to which you want the effective group ID set.

Description

The `setregid` function is used to set the real and effective group IDs of the calling process. If *rgid* is -1, the real group ID is not changed; if *egid* is -1, the effective group ID is not changed. The real and effective group IDs can be set to different values in the same call.

Only a process with the IMPERSONATE privilege can set the real group ID and the effective group ID to any valid value.

A nonprivileged process can set either the real group ID to the saved set-group-ID from an `exec` function, or the effective group ID to the saved set-group-ID or the real group ID.

Any supplementary group IDs of the calling process remain unchanged.

If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group-ID.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

Return Values

0

Successful completion.

-1

Indicates an error. Neither of the group IDs is changed, and `errno` is set to one of the following values:

- EINVAL– The value of the *rgid* or *egid* argument is invalid or out-of-range.
- EPERM– The process does not have the IMPERSONATE privilege, and a change other than changing the real group ID to the saved set-group-ID, or changing the effective group ID to the real group ID or the saved group ID, was requested.

setreuid

setreuid — Sets the user IDs.

Format

```
#include <unistd.h>
int setreuid (uid_t ruid, uid_t euid);
```

Arguments

ruid

The value to which you want the real user ID set.

euid

The value to which you want the effective user ID set.

Description

The `setreuid` function sets the real and effective user IDs of the current process to the values specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is -1, the corresponding effective or real user ID of the current process is left unchanged.

A process with the IMPERSONATE privilege can set either ID to any value. An unprivileged process can set the effective user ID only if the *euid* argument is equal to either the real, effective, or saved user ID of the process.

It is unspecified whether a process without the IMPERSONATE privilege is permitted to change the real user ID to match the current real, effective, or saved user ID of the process.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

See also `getuid` to know how UIC is represented.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL– The value of the *ruid* or *euid* argument is invalid or out of range.
- EPERM– The current process does not have the IMPERSONATE privilege, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID, or an attempt was made to change the real user ID to a value not permitted by the implementation.

setsid

setsid — Creates a session and sets the process group ID.

Format

```
#include <unistd.h>
pid_t setsid (void);
```

Description

The `setsid` function creates a new session if the calling process is not a process group leader. Upon return, the calling process is the session leader of this new session and the process group leader of a new process group, and it has no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group and the only process in the new session.

This function requires that long (32-bit) UID/GID support be enabled. See Section 1.4.8 for more information.

Return Values

x

The process group ID of the calling process.

(pid_t) -1

Indicates an error. The function sets `errno` to the following value:

- EPERM – The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

setstate

setstate — Restarts and changes random-number generators.

Format

```
char *setstate (char *state;)
```

Argument

state

Points to the array of state information.

Description

The `setstate` function handles restarting and changing random-number generators.

Once you initialize a state, the `setstate` function allows rapid switching between state arrays. The array defined by *state* is used for further random-number generation until the `initstate` function is called or the `setstate` function is called again. The `setstate` function returns a pointer to the previous state array.

After initialization, you can restart a state array at a different point in one of two ways:

- Use the `initstate` function, with the desired *seed*, state array, and size of the array.
- Use the `setstate` function, with the desired state, followed by the `srandom` function with the desired `seed`. The advantage of using both functions is that you do not have to save the state array size once you initialize it.

See also `initstate`, `srandom`, and `random`.

Return Values

x

A pointer to the previous state array information.

0

Indicates an error. The state information is damaged, and `errno` is set to the following value:

- EINVAL – The *state* argument is invalid.

setuid

setuid — With POSIX IDs disabled, implemented for program portability and serves no function. It returns 0 (to indicate success). With POSIX IDs enabled, sets the user IDs.

Format

```
#include <types.h>
#include <unistd.h>
int setuid (__uid_t uid); (_DECC_V4_SOURCE)
uid_t setuid (uid_t uid); (not _DECC_V4_SOURCE)
```

Argument

uid

The value to which you want the user IDs set.

Description

The `setuid` function can be used with POSIX style identifiers enabled or disabled.

POSIX style IDs are supported on OpenVMS Version 7.3-2 and higher.

With POSIX IDs disabled (the default), the `setuid` function is implemented for program portability and serves no function. It returns 0 (to indicate success).

With POSIX style IDs enabled:

- If the process has the IMPERSONATE privilege, the `setuid` function sets the real user ID, effective user ID, and the saved set-user-ID to *uid*.
- If the process does not have appropriate privileges but *uid* is equal to the real user ID or to the saved set-user-ID, then the `setuid` function sets the effective user ID to *uid*. The real user ID and saved set-user-ID remain unchanged.

See also `getuid` to know how UIC is represented.

To enable/disable POSIX style IDs, see Section 1.6.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL– The value of the *uid* argument is invalid and not supported by the implementation.
- EPERM– The process does not have appropriate privileges and *uid* does not match the real user ID or the saved set-user-ID.

setvbuf

setvbuf — Associates a buffer with an input or output file and potentially modifies the buffering behavior.

Format

```
#include <stdio.h>
int setvbuf (FILE *file_ptr, char *buffer, int type, size_t size);
```

Arguments

file_ptr

A pointer to a file.

buffer

A pointer to a character array, or a NULL pointer.

type

The buffering type. Use one of the following values defined in `<stdio.h>`: _IOFBF, _IOLBF, or _IONBF.

Use the values _IOLBF and _IOFBF (defined in `<stdio.h>`) for the *type* argument to specify line-buffered and fully buffered I/O, respectively. If *type* is _IONBF, *file_ptr* is unbuffered and *size* and *buffer* are ignored.

size

The number of bytes to be used in *buffer* by the C RTL for buffering this file. The buffer size must be a minimum of 8192 bytes and a maximum of 32767 bytes.

Description

You can use the `setvbuf` function after the file is opened but before any I/O operations are performed.

The C RTL provides the following types of ANSI-conforming file buffering:

In line-buffered I/O, characters are buffered in an area of memory until a new-line character is seen, at which point the appropriate RMS routine is called to transmit the entire buffer. Line buffering is more efficient than unbuffered I/O since it reduces the system overhead, but it delays the availability of the data to the user or disk on output.

In fully buffered I/O, characters are buffered in an area of memory until the buffer is full, regardless of the presence of break characters. Full buffering is more efficient than line buffering or unbuffered I/O, but it delays the availability of output data even longer than line buffering.

Use the values _IOLBF and _IOFBF defined in `<stdio.h>` for the *type* argument to specify line-buffered and fully buffered I/O, respectively.

If *file_ptr* specifies a terminal device, the C RTL uses line-buffered I/O; otherwise, it uses fully buffered I/O.

The C RTL automatically allocates a buffer to use for each I/O stream, so there are several buffer allocation possibilities:

- If *buffer* is not a NULL pointer and *size* is not smaller than the automatically allocated buffer, then `setvbuf` uses *buffer* as the file buffer.
- If *buffer* is a NULL pointer or *size* is smaller than the automatically allocated buffer, the automatically allocated buffer is used as the buffer area.
- If *buffer* is a NULL pointer and *size* is larger than the automatically allocated buffer, then `setvbuf` allocates a new buffer equal to the specified size and uses that as the file buffer.

User programs must not depend on the contents of *buffer* once I/O has been performed on the stream. The C RTL might or might not use *buffer* for any given I/O operation.

Generally, it is unnecessary to use `setvbuf` or `setbuf` to control the buffer size used by the C RTL. The automatically allocated buffer sizes are chosen for efficiency based on the kind of I/O operations performed and the device characteristics (such as terminal, disk, or socket).

The `setvbuf` and `setbuf` functions are useful to introduce buffering for improved performance when writing a large amount of text to the `stdout` stream. This stream is unbuffered by default when bound to a terminal device (the normal case), and therefore incurs a large number of OpenVMS buffered I/O operations unless C RTL buffering is introduced by a call to `setvbuf` or `setbuf`.

The `setvbuf` function is used only to control the buffering used by the C RTL, not the buffering used by the underlying RMS I/O operations. You can modify RMS default buffering behavior by specifying various values for the ctx, fop, rat, gbc, mbc, mbf, rfm, and rop RMS keywords when the file is opened by the `creat`, `freopen` or `open` functions.

The `setvbuf` function now takes 64-bit arguments. However, the *buffer* parameter must contain a 32-bit memory buffer, so when compiling the application with /POINTER=64 or /POINTER=LONG, `malloc32` must be used to allocate the *buffer*.

Return Values

0

Indicates success.

nonzero value

Indicates that an invalid input value was specified for *type* or *file_ptr*, or because *file_ptr* is being used by another thread (see Section 1.8.1).

shm_open

shm_open — Opens a shared memory object.

Format

```
#include <sys/mman.h>
int shm_open (const char *name, int oflag, mode_t mode);
```

Argument

name

Pointer to a string naming a shared memory object.

oflag

Specifies options that define file status and file access modes. This argument is constructed from the bitwise inclusive OR of zero or more of the options defined in the `<fcntl.h>` header file.

mode

The shared memory object's permission bits. This argument is used only when the shared memory object is being created.

Description

The `shm_open` function establishes a connection between a shared memory object and a file descriptor. It creates an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that shared memory object. The *name* argument points to a string naming a shared memory object. The name can be a pathname, in which case other processes referring to the same pathname refer to the same shared memory object.

When a shared memory object is created, its state and all data associated with it persist until the shared memory is unlinked.

The `shm_open` function returns a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that process.

The file status flags and file access modes of the open file description are set according to the value of *oflag*, and can have zero or more of the following values:

O_RDONLY – Open for read access only.
O_RDWR – Open for read or write access.
O_CREAT – Create the shared memory if the memory object does not exist already. The user ID and group ID of the shared memory object are identical to those of the calling process. The shared memory object's permission bits are set to the value of *mode*, except those set in the file mode creation mask of the process.
O_EXCL – Prevent the opening of a shared memory object if O_CREAT is set and the shared memory object already exists. Use this option only in combination with O_CREAT.
O_TRUNC – Truncate the shared memory object to zero length if it is successfully opened for read or write access (O_RDWR).

The initial contents of the shared memory object are binary zeros.

Return Values

n

Upon success, a nonnegative integer representing the lowest numbered unused file descriptor. The file descriptor points to the shared memory object.

-1

Indicates failure. `errno` is set to indicate the error:

- EACCES– Permission to create the shared memory object is denied, or the shared memory object exists and the permissions specified by *oflag* are denied, or O_TRUNC is specified and write permission is denied.
- EEXIST– O_CREAT and O_EXCL are set, but the named shared memory object already exists.
- EINTR– A signal has interrupted the `shm_open` operation.
- EINVAL– The `shm_open` operation is not supported for the given *name*.
- EMFILE– Too many file descriptors are currently in use by this process.

- ENAMETOOLONG– The length of the *name* argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
- ENFILE– Too many shared memory objects are currently open in the system.
- ENOENT– O_CREAT is not set and the named shared memory object does not exist.
- ENOSPC– Memory space for creation of the new shared memory object is insufficient.

shm_unlink

shm_unlink — Removes a shared memory object.

Format

```
#include <sys/mman.h>
int shm_unlink (const char *name);
```

Argument

name

Pointer to a string naming the shared memory object to remove.

Description

The `shm_unlink` function removes the name of the shared memory object named by the string pointed to by *name*.

If one or more references to the shared memory object exist when the object is unlinked, the name is removed before `shm_unlink` returns, but the removal of the memory object contents is postponed until all open and map references to the shared memory object have been removed.

Even if the object continues to exist after the last `shm_unlink`, reuse of the name subsequently causes `shm_unlink` to behave as if no shared memory object with this name exists (that is, `shm_open` will fail if O_CREAT is not set, or will create a new shared memory object if O_CREAT is set).

Return Values

0

Indicates success.

-1

Indicates failure, the named shared memory object is not changed by the function call, and `errno` is set to indicate the error:

- EACCES– Permission is denied to unlink the named shared memory object.
- ENAMETOOLONG– The length of the *name* argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
- ENOENT– The named shared memory object does not exist.

shmat

shmat — Shared memory attach operation.

Format

```
#include <shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Arguments

shmid

A shared memory identifier, a positive integer. It is created by the `shmget()` function and used to identify the shared memory segment on which to perform the shared memory attach operation.

shmaddr

Address within the calling process to which the shared memory segment attached. For more information, see the Description.

shmflg

Flag to indicate type of attach operation. For more information, see the Description.

Description

The `shmat` function attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the address space of the calling process.

The segment is attached at the address specified by one of the following criteria:

- If *shmaddr* is a null pointer, the segment is attached at the first available address as selected by the system. This is the preferred method of using `shmat`.
- If *shmaddr* is not a null pointer and (*shmflg* & SHM_RND) is non-zero, the segment is attached at the address given by (*shmaddr* -((__int64)*shmaddr*% SHMLBA)). The character '%' is the C language remainder operator.
- If *shmaddr* is not a null pointer and (*shmflg* & SHM_RND) is zero, the segment is attached at the address given by *shmaddr*.
- The segment is attached for reading if (*shmflg* & SHM_RDONLY) is non-zero and the calling process has read permission; otherwise, if it is zero and the calling process has both read and write permission, the segment is attached for reading and writing.

A successful `shmat()` function updates the members of the `shmid_ds` structure associated with the shared memory segment as follows:

- `shm_atime` is set to the current time.
- `shm_lpid` is set to the process ID of the calling process.
- `shm_nattch` is incremented by one.

It is not possible to attach a shared memory segment if it has already been marked as removed.

Note

1. For the current implementation the SHMLBA value is 8 KB.
2. If the application passes its own valid non-zero address, the current implementation requires this address to be 32 bit; a 64 bit virtual address is not supported.

Return Values

x

Successful completion. The function returns the start address of the attached shared memory segment.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EACCES – Operation permission is denied to the calling process.
- EINVAL – The value of *shmid* is not a valid shared memory identifier; or the *shmaddr* is not a null pointer, and the value of (*shmaddr* ((__int64)*shmaddr* % SHMLBA)) is an illegal address for attaching shared memory; or the *shmaddr* is not a null pointer, (*shmflg* & SHM_RND) is zero, and the value of *shmaddr* is an illegal address for attaching shared memory.
- EMFILE – The number of shared memory segments attached to the calling process exceeds the system imposed limit.
- ENOMEM – The available data space is not large enough to accommodate the shared memory segment.
- EVMSERR – OpenVMS specific non-translatable error code.

shmctl

shmctl — Shared memory control operations.

Format

```
#include <shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Arguments

shmid

A shared memory identifier, a positive integer. It is created by the `shmget()` function and used to identify the shared memory segment on which to perform the control operation.

cmd

The control operation (IPC_STAT, IPC_SET, or IPC_RMID) to perform on the shared memory segment. For more information, see the Description.

buf

Pointer to a `shmid_ds` structure, defined in <shm.h> as follows:

```
struct shmid_ds {
struct  ipc_perm shm_perm; /* Permissions structure defined in <ipc.h> */
size_t  shm_segsz;  /* Size of memory segment in bytes */
pid_t  shm_lpid;    /* Process ID of last memory operation */
pid_t  shm_cpid;    /* Process ID of segment creator */
shmatt_t   shm_nattch;    /* Number of current attaches */
time_t shm_atime;    /* Time of last shmat  */
time_t shm_dtime;    /* Time of last shmdt  */
time_t shm_ctime;    /* Time of last change */
};
```

Description

The `shmctl()` function provides a variety of shared memory control operations as specified by the *cmd* argument. It can have the following values:

IPC_STAT

Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in <shm.h>.

IPC_SET

Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */
```

This *cmd* can be executed by a process that has an effective user ID equal to either that of a user having appropriate privileges or to the value of either `shm_perm.uid` or `shm_perm.cuid` in the data structure associated with *shmid*.

IPC_RMID

Remove the shared memory identifier specified by *shmid* from the system and delete the shared memory segment and data structure associated with it. If the segment is attached to one or more processes, the segment key is changed to IPC_PRIVATE and the segment is marked as removed. The segment disappears when the last attached process detaches it. This *cmd* can be executed by a process that has an effective user ID equal to either that of a user with appropriate privileges or to the value of either `shm_perm.uid` or `shm_perm.cuid` in the data structure associated with *shmid*.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EACCES – The argument *cmd* is equal to IPC_STAT and the calling process does not have read permission.
- EINVAL – The value of *shmid* is not a valid shared memory identifier, or the value of *cmd* is not a valid command.
- EFAULT – The argument *cmd* has value IPC_SET or IPC_STAT but the address pointed to by *buf* is not accessible.
- EPERM – The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmid*.
- EVMSERR – OpenVMS specific non-translatable error code.

shmdt

shmdt — Shared memory detach operation.

Format

```
#include <shm.h>
int shmdt(const void *shmaddr);
```

Arguments

shmaddr

The address returned by a previous call to `shmat`.

Description

The `shmdt()` function detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process.

The to-be-detached segment must be currently attached with *shmaddr* equal to the value returned by the attaching `shmat()` function.

On a successful `shmdt()` function the system updates the members of the `shmid_ds` structure associated with the shared memory segment as follows:

- `shm_dtime` is set to the current time.
- `shm_lpid` is set to the process-ID of the calling process.
- `shm_nattch` is decremented by one. If it becomes zero and the segment is marked for deletion, the segment is deleted. For more information see the `shmctl` function.

Upon exit, all the attached shared memory segments are detached from the process.

Return Values

0

Successful completion.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL – The value of *shmaddr* is not the data segment start address of a shared memory segment.
- EVMSERR – OpenVMS specific non-translatable error code.

shmget

shmget — Gets a shared memory segment.

Format

```
#include <shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

Arguments

key

The key for which the associated shared memory identifier is returned.

size

Shared memory segment size in bytes.

shmflg

Flag used to initialize the low order 9 bits of the `shm_perm.mode` member of the `shmid_ds` data structure associated with the new shared memory segment. For more information, see the Description.

Description

The `shmget()` function returns the shared memory identifier associated with the *key*.

A shared memory identifier with its associated `shmid_ds` data structure is created for a *key* if one of the following is TRUE:

- The *key* argument is equal to IPC_PRIVATE.
- The *key* argument does not already have a shared memory identifier associated with it and (*shmflg* and IPC_CREAT) is non-zero.

If *shmflg* specifies both IPC_CREAT and IPC_EXCL and a shared memory segment already exists for a *key*, `shmget()` function fails with `errno` set to EEXIST.

When it is created, the `shmid_ds` data structure associated with the new shared memory identifier is initialized as follows:

- The values of `shm_perm.cuid`, `shm_perm.uid`, `shm_perm.cgid`, and `shm_perm.gid` are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low order 9 bits of `shm_perm.mode` are set equal to the low order 9 bits of the `shmflg` argument.
- The variable `shm_segsz` is set equal to the value of the size argument.
- The variables `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are set equal to zero, the variable `shm_cpid` is set equal to the Process ID of the segment creator, and the variable `shm_ctime` is set equal to the current time.

Return Values

n

Successful completion. The function returns a non-negative integer shared memory identifier.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EACCES – A shared memory identifier exists for the *key*, but the operation permission as specified by the low order 9 bits of *shmflg* are not granted.
- EEXIST – A shared memory identifier exists for a *key* but ((*shmflg* & IPC_CREAT) && (*shmflg* & IPC_EXCL)) is non-zero.
- EINVAL – The value of size is either less than SHMMIN, greater than the SHMMAX, or a shared memory identifier exists for a *key*, but the size is greater than the size of that segment. With current implementation SHMMIN is defined to 1 byte and SHMMAX is defined to 512 MB.
- ENOENT – A shared memory identifier does not exist for a *key* and (*shmflg* & IPC_CREAT) is equal to zero.
- ENOSPC – A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory segments system-wide is exceeded.
- EVMSERR – OpenVMS specific non-translatable error code.

Example

```
/*
Abstract: This test program creates a Shared Memory Segment, writes
data to the segment then reads the data after attaching again. It also
ensures that an EINVAL error is generated when trying to attach a
memory that has already been deleted.
*/
#include <errno.h>
#include <types.h>
#include <ipc.h>
#include <shm.h>
```

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

#define SHMSZ  128
#define SHMKEY 9229

main()
{

    int shmid;
    char *shm_write;
    char *shm_read;

    printf("Creating Shared Memory Segment\n");
    if ((shmid = shmget(SHMKEY, SHMSZ, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");
        printf("\nshmget(): FAILED\n");
    }
    if ((shm_write = shmat(shmid, NULL, 0)) == (char *) -1)
    {
        perror("shmat");
        printf("\nshmat(): FAILED\n");
        return;
    }
    printf("Writing Data to the Created Shared Memory Segment\n\n");
    memcpy(shm_write, "Test Shared Memory Segment", SHMSZ);
    printf("Detaching Shared Memory Segment\n");
    if( shmdt(shm_write)<0)
        perror("shmdt");

    printf("Attach again to read the data from Shared Memory Segment\n\n");
    if ((shm_read = shmat(shmid, NULL, 0)) == (char *) -1)
    {
        perror("shmat");
        printf("\nshmat(): FAILED\n");
    }
    printf("Reding Data from Shared Memory Segment\n");
    printf("Data in Segment is: %s\n\n",shm_read);

    printf("Detaching Shared Memory Segment\n");
    if( shmdt(shm_read)<0)
        perror("shmdt");
    printf("Deleting Shared Memory Segment using IPC_RMID\n\n");
    if( shmctl(shmid, IPC_RMID, NULL)<0)
        perror("shmctl");

    printf("Attaching to the deleted Shared Memory Segment - error EINVAL
    should be generated\n\n");
    if ((shm_write = shmat(shmid, NULL, 0)) == (char *) -1)
    {
        perror("shmat");
    }
}
```

This example produces the following output:

```
Creating Shared Memory Segment
Writing Data to the Created Shared Memory Segment

Detaching Shared Memory Segment
```

```
Attach again to read the data from Shared Memory Segment

Reding Data from Shared Memory Segment
Data in Segment is: Test Shared Memory Segment

Detaching Shared Memory Segment
Deleting Shared Memory Segment using IPC_RMID

Attaching to the deleted Shared Memory Segment - error EINVAL should be
generated

shmat: invalid argument
```

sigaction

sigaction — Specifies the action to take upon delivery of a signal.

Format

```
#include <signal.h>
int sigaction (int sig, const struct sigaction *action,
struct sigaction *o_action);
```

Arguments

sig

The signal for which the action is to be taken.

action

A pointer to a `sigaction` structure that describes the action to take when you receive the signal specified by the *sig* argument.

o_action

A pointer to a `sigaction` structure. When the `sigaction` function returns from a call, the action previously attached to the specified signal is stored in this structure.

Description

When a process requests the `sigaction` function, the process can both examine and specify what action to perform when the specified signal is delivered. The arguments determine the behavior of the `sigaction` function as follows:

- Specifying the *sig* argument identifies the affected signal. Use any one of the signal values defined in the `<signal.h>` header file, except SIGKILL.

  If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in `sa_flags`, then a SIGCHLD signal is generated for the calling process whenever any of its child processes stop. If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in `sa_flags`, then SIGCHLD signal is not generated in this way.

- Specifying the *action* argument, if not null, points to a `sigaction` structure that defines what action to perform when the signal is received. If the *action* argument is null, signal handling remains unchanged, so you can use the call to inquire about the current handling of the signal.

- Specifying the *o_action* argument, if not null, points to a `sigaction` structure that contains the action previously attached to the specified signal.

The `sigaction` structure consists of the following members:

```
void        (*sa_handler)(int);
sigset_t    sa_mask;
int         sa_flags;
```

The `sigaction` structure members are defined as follows:

`sa_handler`	This member can contain the following values: • SIG_DFL – Specifies the default action taken when the signal is delivered. • SIG_IGN – Specifies that the signal has no effect on the receiving process. • Function pointer – Requests to catch the signal. The signal causes the function call.
`sa_mask`	This member can request that individual signals, in addition to those in the process signal mask, are blocked from delivery while the signal handler function specified by the *sa_handler* member is executing.
`sa_flags`	This member can set the flags to enable further control over the actions taken when a signal is delivered.

The `sa_flags` member of the `sigaction` structure has the following values:

SA_ONSTACK	Setting this bit causes the system to run the signal catching function on the signal stack specified by the `sigstack` function. If this bit is not set, the function runs on the stack of the process where the signal is delivered.
SA_RESETHAND	Setting this bit resets the signal to SIG_DFL. Be aware that you cannot automatically reset SIGILL and SIGTRAP.
SA_NODEFER	Setting this bit does not automatically block the signal as it is intercepted.
SA_NOCLDSTOP	If this bit is set and the *sig* argument is equal to SIGCHLD and a child process of the calling process stops, then a SIGCHLD signal is sent to the calling process only if SA_NOCLDSTOP is not set for SIGCHLD.

When a signal is intercepted by a signal-catching function installed by `sigaction`, a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either `sigprocmask` or `sigsuspend` is made. This mask is formed by taking the union of the current signal mask and the value of the `sa_mask` for the signal being delivered unless SA_NODEFER or SA_RESETHAND is set, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to `sigaction`), until the SA_RESETHAND flag causes resetting of the handler, or until one of the `exec` functions is called.

If the previous action for a specified signal had been established by `signal`, the values of the fields returned in the structure pointed to by the *o_action* argument of `sigaction` are unspecified, and in particular *o_action*->`sa_handler` is not necessarily the same value passed to `signal`. However, if a

pointer to the same structure or a copy thereof is passed to a subsequent call to `sigaction` by means of the *action* argument of `sigaction`), the signal is handled as if the original call to `signal` were repeated.

If `sigaction` fails, no new signal handler is installed.

It is unspecified whether an attempt to set the action for a signal that cannot be intercepted or ignored to SIG_DFL is ignored or causes an error to be returned with `errno` set to EINVAL.

See Section 4.2 for more information on signal handling.

Note

The `sigvec` and `signal` functions are provided for compatibility to old UNIX systems; their function is a subset of that available with the `sigaction` function.

See also `sigvec`, `signal`, `wait`, `read`, and `write`.

Return Values

0

Indicates success.

-1

Indicates an error; A new signal handler is not installed. `errno` is set to one of the following values:

- EFAULT– The *action* or *o_action* argument points to a location outside of the allocated address space of the process.
- EINVAL– The *sig* argument is not a valid signal number. Or an attempt was made to ignore or supply a handler for the SIGKILL, SIGSTOP, and SIGCONT signals.

sigaddset

sigaddset — Adds the specified individual signal.

Format

```
#include <signal.h>
int sigaddset (sigset_t *set, int sig_number);
```

Arguments

set

The signal set.

sig_number

The individual signal.

Description

The `sigaddset` function manipulates sets of signals. This function operates on data objects that you can address by the application, not on any set of signals known to the system. For example, this function does not operate on the set blocked from delivery to a process or the set pending for a process.

The `sigaddset` function adds the individual signal specified by *sig_number* from the signal set specified by *set*.

Example

The following example shows how to generate and use a signal mask that blocks only the SIGINT signal from delivery:

```
#include <signal.h>
int return_value;
sigset_t newset;
  . . .
sigemptyset(&newset);
sigaddset(&newset, SIGINT);
return_value = sigprocmask (SIG_SETMASK, &newset, NULL);
```

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to the following value:

- EINVAL– The value of *sig_number* is not a valid signal number.

sigblock

sigblock — Adds the signals in *mask* to the current set of signals being blocked from delivery.

Format

```
#include <signal.h>
int sigblock (int mask);
```

Argument

mask

The signals to be blocked.

Description

Signal *i* is blocked if the *i* - 1 bit in *mask* is a 1. For example, to add the protection-violation signal to the set of blocked signals, use the following line:

```
sigblock(1 << (SIGBUS - 1));
```

You can express signals in mnemonics (such as SIGBUS for a protection violation) or numbers as defined in the `<signal.h>` header file, and you can express combinations of signals by using the bitwise OR operator (|).

Return Value

x

Indicates the previous set of masked signals.

sigdelset

sigdelset — Deletes a specified individual signal.

Format

```
#include <signal.h>
int sigdelset (sigset_t *set, int sig_number;)
```

Arguments

set

The signal set.

sig_number

The individual signal.

Description

The `sigdelset` function deletes the individual signal specified by *sig_number* from the signal set specified by *set*.

This function operates on data objects that you can address by the application, not on any set of signals known to the system. For example, this function does not operate on the set blocked from delivery to a process or the set pending for a process.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to the following value:

- EINVAL– The value of *sig_number* is not a valid signal number.

sigemptyset

sigemptyset — Initializes the signal set to exclude all signals.

Format

```
#include <signal.h>
int sigemptyset (sigset_t *set);
```

Argument

set

The signal set.

Description

The `sigemptyset` function initializes the signal set pointed to by *set* such that you exclude all signals. A call to `sigemptyset` or `sigfillset` must be made at least once for each object of type `sigset_t` prior to any other use of that object.

This function operates on data objects that you can address by the application, not on any set of signals known to the system. For example, this function does not operate on the set blocked from delivery to a process or the set pending for a process.

See also `sigfillset`.

Example

The following example shows how to generate and use a signal mask that blocks only the SIGINT signal from delivery:

```
#include <signal.h>
int return_value;
sigset_t newset;
  . . .
sigemptyset(&newset);
sigaddset(&newset, SIGINT);
return_value = sigprocmask (SIG_SETMASK, &newset, NULL);
```

Return Values

0

Indicates success.

-1

Indicates an error; the global `errno` is set to indicate the error.

sigfillset

sigfillset — Initializes the signal set to include all signals.

Format

```
#include <signal.h>
int sigfillset (sigset_t *set);
```

Argument

set

The signal set.

Description

The `sigfillset` function initializes the signal set pointed to by *set* such that you include all signals. A call to `sigemptyset` or `sigfillset` must be made at least once for each object of type `sigset_t` prior to any other use of that object.

This function operates on data objects that you can address by the application, not on any set of signals known to the system. For example, this function does not operate on the set blocked from delivery to a process or the set pending for a process.

See also `sigemptyset`.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to the following value:

- EINVAL– The value of the *sig_number* argument is not a valid signal number.

sighold

sighold — Adds the specified signal to the calling process's signal mask.

Format

```
#include <signal.h>
int sighold (int signal);
```

Argument

signal

The specified signal. The *signal* argument can be assigned any of the signals defined in the `<signal.h>` header file, except SIGKILL and SIGSTOP.

Description

The `sighold`, `sigrelse`, and `sigignore` functions provide simplified signal management:

- The `sighold` function adds *signal* to the calling process's signal mask.
- The `sigrelse` function removes *signal* from the calling process's signal mask.

- The `sigignore` function sets the disposition of *signal* to SIG_IGN.

The `sighold` function, in conjunction with `sigrelse` and `sigpause`, can be used to establish critical regions of code that require the delivery of a signal to be temporarily deferred.

Upon success, the `sighold` function returns a value of 0. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

Note

These interfaces are provided for compatibility only. New programs should use `sigaction` and `sigprocmask` to control the disposition of signals.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to the following value:

- EINVAL– The value of the *signal* argument is either an invalid signal number or SIGKILL.

sigignore

sigignore — Sets the disposition of the specified signal to SIG_IGN.

Format

```
#include <signal.h>
int sigignore (int signal);
```

Argument

signal

The specified signal. The *signal* argument can be assigned any of the signals defined in the `<signal.h>` header file, except SIGKILL and SIGSTOP.

Description

The `sighold`, `sigrelse`, and `sigignore` functions provide simplified signal management:

- The `sighold` function adds *signal* to the calling process's signal mask.
- The `sigrelse` function removes *signal* from the calling process's signal mask.
- The `sigignore` function sets the disposition of *signal* to SIG_IGN.

The `sighold` function, in conjunction with `sigrelse` and `sigpause`, can be used to establish critical regions of code that require the delivery of a signal to be temporarily deferred.

Upon success, the `sigignore` function returns a value of 0. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

Note

These interfaces are provided for compatibility only. New programs should use `sigaction` and `sigprocmask` to control the disposition of signals.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to the following value:

- EINVAL– The value of the *signal* argument is either an invalid signal number or SIGKILL, or an attempt is made to catch a signal that cannot be intercepted or to ignore a signal that cannot be ignored.

sigismember

sigismember — Tests whether a specified signal is a member of the signal set.

Format

```
#include <signal.h>
int sigismember (const sigset_t *set, int sig_number);
```

Arguments

set

The signal set.

sig_number

The individual signal.

Description

The `sigismember` function tests whether *sig_number* is a member of the signal set pointed to by *set*.

This function operates on data objects that you can address by the application, not on any set of signals known to the system. For example, this function does not operate on the set blocked from delivery to a process or the set pending for a process.

Return Values

1

Indicates success. The specified signal is a member of the specified set.

0

Indicates an error. The specified signal is not a member of the specified set.

siglongjmp

siglongjmp — Nonlocal go to with signal handling.

Format

```
#include <setjmp.h>
void siglongjmp (sigjmp_buf env, int value);
```

Arguments

env

An address for a `sigjmp_buf` structure.

value

A nonzero value.

Description

The `siglongjmp` function restores the environment saved by the most recent call to `sigsetjmp` in the same process with the corresponding `sigjmp_buf` argument.

All accessible objects have values when `siglongjmp` is called, with one exception: values of objects of automatic storage duration that changed between the `sigsetjmp` call and `siglongjmp` call are indeterminate.

Because it bypasses the usual function call and return mechanisms, `siglongjmp` executes correctly during interrupts, signals, and any of their associated functions. However, if you invoke `siglongjmp` from a nested signal handler (for example, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

The `siglongjmp` function restores the saved signal mask only if you initialize the *env* argument by a call to `sigsetjmp` with a nonzero *savemask* argument.

After `siglongjmp` is completed, program execution continues as if the corresponding call of `sigsetjmp` just returned the value specified by *value*. The `siglongjmp` function cannot cause `sigsetjmp` to return 0 (zero); if *value* is 0, `sigsetjmp` returns 1.

See also `sigsetjmp`.

sigmask

sigmask — Constructs the mask for a given signal number.

Format

```
#include <signal.h>
int sigmask (signum);
```

Argument

signum

The signal number for which the mask is to be constructed.

Description

The `sigmask` function is used to construct the mask for a given *signum*. This mask can be used with the `sigblock` function.

Return Value

x

The mask constructed for *signum*

signal

signal — Allows you to specify the way in which the signal *sig* is to be handled: use the default handling for the signal, ignore the signal, or call the signal handler at the address specified.

Format

```
#include <signal.h>
void (*signal (int sig, void (*func) (int))) (int);
```

Arguments

sig

The number or mnemonic associated with a signal. This argument is usually one of the mnemonics defined in the `<signal.h>` header file.

func

Either the action to take when the signal is raised, or the address of a function needed to handle the signal.

Description

If *func* is the constant SIG_DFL, the action for the given signal is reset to the default action, which is to terminate the receiving process. If the argument is SIG_IGN, the signal is ignored. Not all signals can be ignored.

If *func* is neither SIG_DFL nor SIG_IGN, it specifies the address of a signal-handling function. When the signal is raised, the addressed function is called with *sig* as its argument. When the addressed function returns, the interrupted process continues at the point of interruption. (This is called catching a signal. Signals are reset to SIG_DFL after they are intercepted, except as shown in Chapter 4.)

You must call the `signal` function each time you want to catch a signal.

See Section 4.2 for more information on signal handling.

To cause an OpenVMS exception or a signal to generate a UNIX style signal, OpenVMS condition handlers must return SS$_RESIGNAL upon receiving any exception that they do not want to handle. Returning SS$_CONTINUE prevents the correct generation of a UNIX style signal. See Chapter 4 for a list of OpenVMS exceptions that correspond to UNIX signals.

Return Values

x

The address of the function previously established to handle the signal.

SIG_ERR

Indicates that the *sig* argument is out of range.

sigpause

sigpause — Assigns *mask* to the current set of masked signals and then waits for a signal.

Format

```
#include <signal.h>
int sigpause (int mask);
```

Argument

mask

The signals to be blocked.

Description

See the `sigblock` function for information about the *mask* argument.

When control returns to `sigpause`, the function restores the previous set of masked signals, sets `errno` to EINTR, and returns -1 to indicate an interrupt. The value EINTR is defined in the `<errno.h>` header file.

Return Value

-1

Indicates an interrupt. `errno` is set to EINTR.

sigpending

sigpending — Examines pending signals.

Format

```
#include <signal.h>
int sigpending (sigset_t *set);
```

Argument

set

A pointer to a `sigset_t` structure.

Description

The `sigpending` function stores the set of signals that are blocked from delivery and pending to the calling process in the location pointed to by the *set* argument.

Call either the `sigemptyset` or the `sigfillset` function at least once for each object of type `sigset_t` prior to any other use of that object. If you do not initialize an object in this way and supply an argument to the `sigpending` function, the result is undefined.

See also `sigemptyset` and `sigfillset` in this section.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to the following value:

- SIGSEGV– Bad mask argument.

sigprocmask

sigprocmask — Sets the current signal mask.

Format

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t *o_set);
```

Arguments

how

An integer value that indicates how to change the set of masked signals. Use one of the following values:

SIG_BLOCK	The resulting set is the union of the current set and the signal set pointed to by the *set* argument.
SIG_UNBLOCK	The resulting set is the intersection of the current set and the complement of the signal set pointed to by the *set* argument.
SIG_SETMASK	The resulting set is the signal set pointed to by the *set* argument.

set

The signal set. If the value of the *set* argument is:

- Not NULL– It points to a set of signals used to change the currently blocked set.
- NULL– The value of the *how* argument is not significant, and the process signal mask is unchanged, so you can use the call to inquire about currently blocked signals.

o_set

A non-NULL pointer to the location where the signal mask in effect at the time of the call is stored.

Description

The `sigprocmask` function is used to examine or change the signal mask of the calling process.

Typically, use the `sigprocmask` SIG_BLOCK value to block signals during a critical section of code, then use the `sigprocmask` SIG_SETMASK value to restore the mask to the previous value returned by the `sigprocmask` SIG_BLOCK value.

If there are any unblocked signals pending after the call to the `sigprocmask` function, at least one of those signals is delivered before the `sigprocmask` function returns.

You cannot block SIGKILL or SIGSTOP signals with the `sigprocmask` function. If a program attempts to block one of these signals, the `sigprocmask` function gives no indication of the error.

Example

The following example shows how to set the signal mask to block only the SIGINT signal from delivery:

```
#include <signal.h>

int return_value;
sigset_t newset;
 . . .
sigemptyset(&newset);
sigaddset(&newset, SIGINT);
return_value = sigprocmask (SIG_SETMASK, &newset, NULL);
```

Return Values

0

Indicates success.

-1

Indicates an error. The signal mask of the process is unchanged. `errno` is set to one of the following values:

- EINVAL– The value of the *how* argument is not equal to one of the defined values.
- EFAULT– The *set* or *o_set* argument points to a location outside the allocated address space of the process.

sigrelse

sigrelse — Removes the specified signal from the calling process's signal mask.

Format

```
#include <signal.h>
int sigrelse (int signal);
```

Argument

signal

The specified signal. The *signal* argument can be assigned any of the signals defined in the `<signal.h>` header file, except SIGKILL and SIGSTOP.

Description

The `sighold`, `sigrelse`, and `sigignore` functions provide simplified signal management:

- The `sighold` function adds *signal* to the calling process's signal mask.
- The `sigrelse` function removes *signal* from the calling process's signal mask.
- The `sigignore` function sets the disposition of *signal* to SIG_IGN.

The `sighold` function, in conjunction with `sigrelse` and `sigpause`, can be used to establish critical regions of code that require the delivery of a signal to be temporarily deferred.

Upon success, the `sigrelse` function returns a value of 0. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

Note

These interfaces are provided for compatibility only. New programs should use `sigaction` and `sigprocmask` to control the disposition of signals.

Return Values

0

Indicates success.

-1

Indicates an error; `errno` is set to the following value:

- EINVAL– The value of the *signal* argument is either an invalid signal number or SIGKILL.

sigsetjmp

sigsetjmp — Sets a jump point for a nonlocal goto.

Format

```
#include <setjmp.h>
init sigsetjmp (sigjmp_buf env, int savemask);
```

Arguments

env

An address for a `sigjmp_buf` structure.

savemask

An integer value that specifies whether you need to save the current signal mask.

Description

The `sigsetjmp` function saves its calling environment in its *env* argument for later use by the `siglongjmp` function.

If the value of *savemask* is not 0 (zero), `sigsetjmp` also saves the process's current signal mask as part of the calling environment.

See also `siglongjmp`.

Restrictions

You cannot invoke the `longjmp` function from an OpenVMS condition handler. However, you may invoke `longjmp` from a signal handler that has been established for any signal supported by the C RTL, subject to the following nesting restrictions:

- The `longjmp` function will not work if you invoke it from nested signal handlers. The result of the `longjmp` function, when invoked from a signal handler that has been entered as a result of an exception generated in another signal handler, is undefined.
- Do not invoke the `sigsetjmp` function from a signal handler unless the associated `longjmp` is to be issued before the handling of that signal is completed.
- Do not invoke the `longjmp` function from within an exit handler (established with `atexit` or SYS$DCLEXH). Exit handlers are invoked after image tear-down, so the destination address of the `longjmp` no longer exists.
- Invoking `longjmp` from within a signal handler to return to the main thread of execution might leave your program in an inconsistent state. Possible side effects include the inability to perform I/O or to receive any more UNIX signals. Use `siglongjmp` instead.

Return Values

0

Indicates success.

nonzero

The return is a call to the `siglongjmp` function.

sigsetmask

sigsetmask — Establishes those signals that are blocked from delivery.

Format

```
#include <signal.h>
int sigsetmask (int mask);
```

Argument

mask

The signals to be blocked.

Description

See the `sigblock` function for information about the *mask* argument.

Return Value

x

The previous set of masked signals.

sigsuspend

sigsuspend — Atomically changes the set of blocked signals and waits for a signal.

Format

```
#include <signal.h>
int sigsuspend (const sigset_t *signal_mask);
```

Argument

signal_mask

A pointer to a set of signals.

Description

The `sigsuspend` function replaces the signal mask of the process with the set of signals pointed to by the *signal_mask* argument. Then it suspends execution of the process until delivery of a signal whose action is either to execute a signal catching function or to terminate the process. You cannot block the SIGKILL or SIGSTOP signals with the `sigsuspend` function. If a program attempts to block either of these signals, `sigsuspend` gives no indication of the error.

If delivery of a signal causes the process to terminate, `sigsuspend` does not return. If delivery of a signal causes a signal catching function to execute, `sigsuspend` returns after the signal catching function returns, with the signal mask restored to the set that existed prior to the call to `sigsuspend`.

The `sigsuspend` function sets the signal mask and waits for an unblocked signal as one atomic operation. This means that signals cannot occur between the operations of setting the mask and waiting for a signal. If a program invokes `sigprocmask` SIG_SETMASK and `sigsuspend` separately, a signal that occurs between these functions is often not noticed by `sigsuspend`.

In normal usage, a signal is blocked by using the `sigprocmask` function at the beginning of a critical section. The process then determines whether there is work for it to do. If there is no work, the process waits for work by calling `sigsuspend` with the mask previously returned by `sigprocmask`.

If a signal is intercepted by the calling process and control is returned from the signal handler, the calling process resumes execution after `sigsuspend`, which always returns a value of -1 and sets `errno` to EINTR.

See also `sigpause` and `sigprocmask`.

sigtimedwait

sigtimedwait — Suspends a calling thread and waits for queued signals to arrive.

Format

```
#include <signal.h>
int sigtimedwait (const sigset_t set, siginfo_t *info,
const struct timespec *timeout);
```

Arguments

set

The set of signals to wait for.

info

Pointer to a `siginfo` structure that is receiving data describing the signal, including any application-defined data specified when the signal was posted.

timeout

A timeout for the wait. If *timeout* is NULL, the argument is ignored.

Description

The `sigtimedwait` function behaves the same as the `sigwaitinfo` function except that if none of the signals specified by *set* are pending, `sigtimedwait` waits for the time interval specified in the `timespec` structure referenced by *timeout*. If the `timespec` structure pointed to by *timeout* is zero-valued and if none of the signals specified by *set* are pending, then `sigtimedwait` returns immediately with an error.

See also `sigwait` and `sigwaitinfo`.

See Section 4.2 for more information on signal handling.

Return Values

x

Upon successful completion, the signal number selected is returned.

-1

Indicates that an error occurred; `errno` is set to one of the following values:

- EINVAL– The timeout argument specified a `tv_nsec` value less than 0 or greater than or equal to 1 billion.
- EINTR– The wait was interrupted by an unblocked, intercepted signal.
- EAGAIN– No signal specified by *set* was generated within the specified timeout period.

sigvec

sigvec — Permanently assigns a handler for a specific signal.

Format

```
#include <signal.h>
int sigvec (int sigint, struct sigvec *sv, struct sigvec *osv);
```

Arguments

sigint

The signal identifier.

sv

Pointer to a `sigvec` structure (see the Description section).

osv

If *osv* is not NULL, the previous handling information for the signal is returned.

Description

If *sv* is not NULL, it specifies the address of a structure containing a pointer to a handler routine and mask to be used when delivering the specified signal, and a flag indicating whether the signal is to be processed on an alternative stack. If *sv–>onstack* has a value of 1, the system delivers the signal to the process on a signal stack specified with `sigstack`.

The `sigvec` function establishes a handler that remains established until explicitly removed or until the image terminates.

The `sigvec` structure is defined in the `<signal.h>` header file:

```
struct sigvec
   {
      int   (*handler)();
      int   mask;
      int   onstack;
   };
```

See Section 4.2 for more information on signal handling.

Return Values

0

Indicates that the call succeeded.

-1

Indicates that an error occurred.

sigwait

sigwait — Suspends a calling thread and waits for queued signals to arrive.

Format

```
#include <signal.h>
int sigwait (const sigset_t set, int *sig);
```

Arguments

set

The set of signals to wait for.

sig

Returns the signal number of the selected signal.

Description

The `sigwait` function suspends the calling thread until at least one of the signals in the *set* argument is in the caller's set of pending signals. When this happens, one of those signals is automatically selected and removed from the set of pending signals. The signal number identifying that signal is then returned in the location referenced by *sig*.

The effect is unspecified if any signals in the *set* argument are not blocked when the `sigwait` function is called.

The *set* argument is created using the set manipulation functions `sigemptyset`, `sigfillset`, `sigaddset`, and `sigdelset`.

If, while the `sigwait` function is waiting, a signal occurs that is eligible for delivery (that is, not blocked by the signal mask), that signal is handled asynchronously and the wait is interrupted.

See also `sigtimedwait` and `sigwaitinfo`.

See Section 4.2 for more information on signal handling.

Return Values

0

Upon successful completion, `sigwait` stores the signal number of the received signal at the location referenced by *sig* and returns 0.

nonzero

Indicates that an error occurred; `errno` is set to the following value:

- EINVAL– The *set* argument contains an invalid or unsupported signal number.

sigwaitinfo

sigwaitinfo — Suspends a calling thread and waits for queued signals to arrive.

Format

```
#include <signal.h>
int sigwaitinfo (const sigset_t set, siginfo_t *info);
```

Arguments

set

The set of signals to wait for.

info

Pointer to a `siginfo` structure that is receiving data describing the signal, including any application-defined data specified when the signal was posted.

Description

The `sigwaitinfo` function behaves the same as the `sigwait` function if the *info* argument is NULL.

If the *info* argument is non-NULL, the `sigwaitinfo` function behaves the same as `sigwait`, except that the selected signal number is stored in the `si_signo` member of the `siginfo` structure, and the cause of the signal is stored in the `si_code` member. If any value is queued to the selected signal, the first such queued value is dequeued and the value is stored in the `si_value` member of *info*. The system resource used to queue the signal is released and made available to queue other signals. If no value is queued, the content of the `si_value` member is undefined. If no further signals are queued for the selected signal, the pending indication for that signal is reset.

See also `sigtimedwait` and `sigwait`.

See Section 4.2 for more information on signal handling.

Return Values

x

Upon successful completion, the signal number selected is returned.

-1

Indicates that an error occurred; `errno` is set to one of the following values:

- EINVAL– The *set* argument contains an invalid or unsupported signal number.

- EINTR– The wait was interrupted by an unblocked, intercepted signal.

sin

sin — Returns the sine of its radian argument.

Format

```
#include <math.h>
double sin (double x);
float sinf (float x);
long double sinl (long double x);
double sind (double x);
float sindf (float x);
long double sindl (long double x);
```

Argument

x

A radian expressed as a floating-point number.

Description

The `sin` functions compute the sine of *x* measured in radians.

The `sind` functions compute the sine of *x* measured in degrees.

Return Values

x

The sine of the argument.

NaN

x = ±Infinity or NaN; `errno` is set to EDOM.

0

Underflow occurred; `errno` is set to ERANGE.

sinh

sinh — Returns the hyperbolic sine of its argument.

Format

```
#include <math.h>
double sinh (double x);
float sinhf (float x);
long double sinhl (long double x);
```

Argument

x

A real number.

Return Values

n

The hyperbolic sine of the argument.

HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

0

Underflow occurred; `errno` is set to ERANGE.

NaN

x is NaN; `errno` is set to EDOM.

sleep

sleep — Suspends the execution of the current process (or thread in a threaded program) for at least the number of seconds indicated by its argument.

Format

```
#include <unistd.h>
unsigned int sleep (unsigned seconds); (_DECC_V4_SOURCE)
int sleep (unsigned seconds); (not _DECC_V4_SOURCE)
```

Argument

seconds

The number of seconds.

Description

The `sleep` function sleeps for the specified number of seconds, or until a signal is received, or until the process (or thread in a threaded program) executes a call to SYS$WAKE.

If a SIGALRM signal is generated, but blocked or ignored, the `sleep` function returns. For all other signals, a blocked or ignored signal does not cause `sleep` to return.

Return Values

x

The number of seconds that the process awoke early.

0

If the process slept the full number of seconds specified by *seconds*.

snprintf

snprintf — Performs formatted output to a string in memory.

Format

```
#include <stdio.h>
int snprintf (char *str, size_t n, const char *format_spec, ...);
```

Arguments

str

The address of the string that will receive the formatted output.

n

The size of the buffer referred to by *str*.

format_spec

A pointer to a character string that contains the format specification. For more information about format specifications and conversion characters, see Chapter 2.

. . .

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 at least 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. Excess output pointers, if any, are ignored.

Description

The `snprintf` function is identical to the `sprintf` function with the addition of the *n* argument, which specifies the size of the buffer referred to by *str*.

On successful completion, `snprintf` returns the number of bytes (excluding the terminating null byte) that would be written to *str* if *n* is sufficiently large.

If *n* is 0, nothing is written, the number of bytes (excluding the terminating null) that would be written if *n* were sufficiently large are returned, and *str* might be a NULL pointer. Otherwise, output bytes beyond the n - 1st are discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If an output error is encountered, a negative value is returned.

For a complete description of the format specification and the output source, see Chapter 2.

Return Values

x

The number of bytes (excluding the terminating null byte) that would be written to *str* if *n* is sufficiently large.

Negative value

Indicates an output error occurred. The function sets `errno`. For a list of `errno` values set by this function, see `fprintf`.

sprintf

sprintf — Performs formatted output to a string in memory.

Format

```
#include <stdio.h>
int sprintf (char *str, const char *format_spec, ...);
```

Arguments

str

The address of the string that will receive the formatted output. It is assumed that this string is large enough to hold the output.

format_spec

A pointer to a character string that contains the format specification. For more information about format specifications and conversion characters, see Chapter 2.

. . .

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 at least 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. Excess output pointers, if any, are ignored.

Description

The `sprintf` function places output followed by the null character (\ 0) in consecutive bytes starting at **str*. The user must ensure that enough space is available.

Consider the following example of a conversion specification:

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

```
main()
{
   int  temp = 4, temp2 = 17;
   char s[80];

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

In this example, character string `s` has the following contents:

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

For a complete description of the format specification and the output source, see Chapter 2.

Return Values

x

The number of characters placed in the output string, not including the final null character.

Negative value

Indicates an output error occurred. The function sets `errno`. For a list of `errno` values set by this function, see `fprintf`.

sqrt

sqrt — Returns the square root of its argument.

Format

```
#include <math.h>
double sqrt (double x);
float sqrtf (float x);
long double sqrtl (long double x);
```

Argument

x

A real number.

Return Values

val

The square root of *x*, if *x* is nonnegative.

0

x is negative; `errno` is set to EDOM.

NaN

x is NaN; `errno` is set to EDOM.

srand

srand — Initializes the pseudorandom-number generator `rand`.

Format

```
#include <stdlib.h>
void srand (unsigned int seed);
```

Argument

seed

An unsigned integer.

Description

The `srand` function uses the argument as a seed for a new sequence of pseudorandom numbers to be returned by subsequent calls to `rand`.

If `srand` is then called with the same seed value, the sequence of pseudorandom numbers is repeated.

If `rand` is called before any calls to `srand`, the same sequence of pseudorandom numbers is generated as when `srand` is first called with a seed value of 1.

srand48

srand48 — Initializes a 48-bit random-number generator.

Format

```
#include <stdlib.h>
void srand48 (long int seed_val);
```

Argument

seed_val

The initialization value to begin randomization. Changing this value changes the randomization pattern.

Description

The `srand48` function initializes the random-number generator. You can use this function in your program before calling the `drand48`, `lrand48`, or `mrand48` functions. (Although it is not recommended practice, constant default initializer values are automatically supplied if you call `drand48`, `lrand48`, or `mrand48` without calling an initialization function).

The function works by generating a sequence of 48-bit integer values, X*i*, according to the linear congruential formula:

```
Xn+1 = (aXn+c)mod m        n >= 0
```

The argument *m* equals 2^{48}, so 48-bit integer arithmetic is performed. Unless you invoke the `lcong48` function, the multiplier value *a* and the addend value *c* are:

$$a = 5DEECE66D_{16} = 273673163155_{8}$$
$$c = B_{16} = 13_{8}$$

The initializer function `srand48` sets the high-order 32 bits of X*i* to the low-order 32 bits contained in its argument. The low-order 16 bits of X*i* are set to the arbitrary value $330E_{16}$.

See also `drand48`, `lrand48`, and `mrand48`.

srandom

srandom — Initializes the pseudorandom-number generator `random`.

Format

```
#include <stdlib.h>
void srandom (unsigned seed);
```

Argument

seed

An initial seed value.

Description

The `srandom` function uses the argument as a seed for a new sequence of pseudorandom numbers to be returned by subsequent calls to `random`. This function has virtually the same calling sequence and initialization properties as the `srand` function, but produce sequences that are more random.

The `srandom` function initializes the current state with the initial seed value. The `srandom` function, unlike the `srand` function, does not return the old seed because the amount of state information used is more than a single word.

See also `rand`, `srand`, `random`, `setstate`, and `initstate`.

sscanf

sscanf — Reads input from a character string in memory, interpreting it according to the format specification.

Format

```
#include <stdio.h>
int sscanf (const char *str, const char *format_spec, ...);
```

Arguments

str

The address of the character string that provides the input text to `sscanf`.

format_spec

A pointer to a character string that contains the format specification. For more information about format specifications and conversion characters, see Chapter 2.

...

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 at least as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input pointers.

Conversion specifications are matched to input sources in left-to-right order. Excess input pointers, if any, are ignored.

Description

The following is an example of a conversion specification:

```
main ()
{
   char str[] = "4 17";
   int   temp,
         temp2;

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

This example produces the following output:

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

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

Return Values

x

The number of successfully matched and assigned input items.

EOF

Indicates that a read error occurred before any conversion. The function sets `errno`. For a list of the values set by this function, see `fscanf`.

ssignal

ssignal — Allows you to specify the action to take when a particular signal is raised.

Format

```
#include <signal.h>
void (*ssignal (int sig, void (*func) (int, ...))) (int, ...);
```

Arguments

sig

A number or mnemonic associated with a signal. The symbolic constants for signal values are defined in the `<signal.h>` header file (see Chapter 4).

func

The action to take when the signal is raised, or the address of a function that is executed when the signal is raised.

Description

The `ssignal` function is equivalent to the `signal` function except for the return value on error conditions.

Since the `signal` function is defined by the ANSI C standard and the `ssignal` function is not, use `signal` for greater portability.

See Section 4.2 for more information on signal handling.

Return Values

x

The address of the function previously established as the action for the signal. The address may be the value SIG_DFL (0) or SIG_IGN (1).

0

Indicates errors. For this reason, there is no way to know whether a return status of 0 indicates failure, or whether it indicates that a previous action was SIG_DFL (0).

[w]standend

[w]standend — Deactivate the boldface attribute for the specified window. The `standend` function operates on the `stdscr` window.

Format

```
#include <curses.h>
int standend (void);
int wstandend (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

The `standend` and `wstandend` functions are equivalent to `clrattr` and `wclrattr` called with the attribute _BOLD.

Return Values

OK

Indicates success.

ERR

Indicates an error.

[w]standout

[w]standout — Activate the boldface attribute of the specified window. The `standout` function acts on the `stdscr` window.

Format

```
#include <curses.h>
int standout (void);
int wstandout (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

The `standout` and `wstandout` functions are equivalent to `setattr` and `wsetattr` called with the attribute _BOLD.

Return Values

OK

Indicates success.

ERR

Indicates an error.

stat

stat — Accesses information about the specified file.

Format

```
#include <stat.h>
int stat (const char *file_spec, struct stat *buffer); (ISO POSIX-1)
int stat (const char *file_spec, struct stat *buffer, ...);
(VSI C Extension)
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `stat` function that is equivalent to the behavior before OpenVMS Version 7.0.

Compiling with the _USE_STD_STAT feature-test macro defined enables a variant of the `stat` function that uses an X/Open standard-compliant definition of the `stat` structure. The _USE_STD_STAT feature-test macro is mutually exclusive with the _DECC_V4_SOURCE and _VMS_V6_SOURCE macros.

Arguments

file_spec

A valid OpenVMS or UNIX style file specification (no wildcards). Read, write, or execute permission of the named file is not required, but you must be able to reach all directories listed in the file specification leading to the file. For more information about UNIX style file specifications, see Chapter 1.

buffer

A pointer to a structure of type `stat`. For convenience, a typedef `stat_t` is defined as `struct stat` in the `<stat.h>` header file.

This argument receives information about the particular file. The members of the structure pointed to by *buffer* are described in the Description section.

...

An optional default file-name string.

This is the only optional RMS keyword that can be specified for the `stat` function. See the description of the `creat` function for the full list of optional RMS keywords and their values.

Description

When the _USE_STD_STAT feature-test macro is not enabled, the legacy `stat` structure is used. When _USE_STD_STAT is enabled, the X/Open standard-compliant `stat` structure is used.

Legacy stat Structure

With the _USE_STD_STAT feature-test macro defined to DISABLE, the following legacy `stat` structure is used:

Member	Type	Definition
`st_dev`	`dev_t`	Pointer to the physical device name
`st_ino[3]`	`ino_t`	Three words to receive the file ID

Member	Type	Definition
st_mode	mode_t	File “mode” (prot, dir, . . .)
st_nlink	nlink_t	For UNIX system compatibility only
st_uid	uid_t	Owner user ID
st_gid	gid_t	Group member: fromst_uid
st_rdev	dev_t	UNIX system compatibility – always 0
st_size	off_t	File size, in bytes. For st_size to report a correct value, you need to flush both the C RTL and RMS buffers.
st_atime	time_t	File access time; always the same as st_mtime
st_mtime	time_t	Last modification time
st_ctime	time_t	File creation time
st_fab_rfm	char	Record format
st_fab_rat	char	Record attributes
st_fab_fsz	char	Fixed header size
st_fab_mrs	unsigned	Record size

The types dev_t, ino_t, off_t, mode_t, nlink_t, uid_t, gid_t, and time_t, are defined in the <stat.h> header file. However, when compiling for compatibility (/DEFINE=_DECC_V4_SOURCE), only dev_t, ino_t, and off_t are defined.

The off_t data type is either a 32-bit or 64-bit integer. The 64-bit interface allows for file sizes greater than 2 GB, and can be selected at compile time by defining the _LARGEFILE feature-test macro as follows: CC/DEFINE=_LARGEFILE

As of OpenVMS Version 7.0, times are given in seconds since the Epoch (00:00:00 GMT, January 1, 1970).

The st_mode structure member is the status information mode defined in the <stat.h> header file. The st_mode bits are described as follows:

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

The `stat` function does not work on remote network files.

If the file is a record file, the `st_size` field includes carriage-control information. Consequently, the `st_size` value will not correspond to the number of characters that can be read from the file.

Also be aware that for `st_size` to report a correct value, you need to flush both the C RTL and RMS buffers.

Standard-Compliant stat Structure

With OpenVMS Version 8.2, the _USE_STD_STAT feature-test macro and standard-compliant `stat` structure are introduced in support of UNIX compatibility.

With _USE_STD_STAT defined to ENABLE, you get the following behavior:

- Old `struct stat` definitions

  Old definitions of `struct stat` are obsolete. You must recompile your applications to access the new features. Existing applications will continue to access the old definitions and functions unless they are recompiled to use the new features.

- Function variants

  Calls to `stat`, `fstat`, `lstat`, and `ftw` accept pointers to structures of the new type. Calls to these functions are mapped to the new library entries `__std_stat`, `__std_fstat`, `__std_lstat`, and `__std_ftw`, respectively.

- Compatibilities with other feature macros

  _DECC_V4_SOURCE source-code compatibility is not supported. You must not enable _DECC_V4_SOURCE and _USE_STD_STAT at the same time.

  _VMS_V6_SOURCE binary compatibility is not supported. You must not enable _VMS_V6_SOURCE and _USE_STD_STAT at the same time. As a result, only UTC (rather than local-time) is supported for the `time_t` fields.

- Type changes

  The following type changes are in effect:

  - 32-bit gid type `gid_t` is used. _DECC_SHORT_GID_T is unsupported.
  - _LARGEFILE offsets are used. `off_t` is forced to 64 bits.
  - Type `ino_t`, representing the file number, is an `unsigned int` quadword (64 bits). Previously, it was an `unsigned short`.
  - Type `dev_t`, representing the device id, is an `unsigned int` quadword (64 bits). Previously, it was a 32-bit character pointer. The new type is standard because it is arithmetic.
  - Types `blksize_t` and `blkcnt_t` are added and defined as `unsigned int` quadwords (64 bits).

- Structure member Changes

  - Two members are added to `struct stat`:

```
blksize_t   st_blksize;
blkcnt_t    st_blocks;
```

According to the X/Open standard, `st_blksize` is the file system-specific preferred I/O blocksize for this file. On OpenVMS systems, `st_blksize` is set to the device buffer size multiplied by the disk cluster size. `st_blocks` is set to the allocated size of the file, in blocks. The blocksize used to calculate `st_blocks` is not necessarily the same as `st_blksize` and, in most cases, will not be the same.

- In `struct stat`, member `st_ino` is of type `ino_t`. In previous C RTL versions, it was of type `ino_t [3]` (array of 3 `ino_t`). Since `ino_t` has changed from a word to a quadword, the size of this member has increased by one word. The principal significance of this change is that it makes `st_ino` a scalar, which is how most open source applications define it.

- The new definition of `ino_t` also affects applications that include the `<dirent.h>` header file. In `struct dirent`, member `d_ino` changes in the same way as the `st_ino` member of `struct stat` in `<stat.h>`.

- Several macros that are not part of any standard were introduced in `<stat.h>` to facilitate access to the constituent parts of `ino_t` values:

  S_INO_NUM(ino), S_INO_SEQ(ino), and S_INO_RVN(ino) return the FILES-11 file number, sequence number, and relative volume number of ino, respectively, as `unsigned short`s.
  S_INO_RVN_RVN(ino) returns the byte of the RVN field containing the relative volume number;
  S_INO_RVN_NMX(ino) returns the byte of the RVN field containing the file number extension.

  Although individual components can be broken out like this, they are not part of the X/Open standard and should not be relied on in portable applications.

- Semantic changes

  Values of type `dev_t` are now unique for each device across clusters. An algorithm based on device name and allocation class or SCSSYSTEMID (for single-pathed devices) calculates the device id value having these characteristics, an X/Open standard requirement. Typically, the combination of file number and device id uniquely identifies a file in a cluster.

  This change affects `stat` structure members `st_dev` and `st_rdev`. For compatibility with previous releases, `st_rdev` is set to either 0 or `st_dev`.

Note

(Integrity servers, Alpha) On OpenVMS Alpha and Integrity server systems, the `stat`, `fstat`, `utime`, and `utimes` functions have been enhanced to take advantage of the new file-system support for POSIX compliant file timestamps.

This support is available only on ODS-5 devices on OpenVMS Alpha systems beginning with a version of OpenVMS Alpha after Version 7.3.

Before this change, the `stat` and `fstat` functions were setting the values of the `st_ctime`, `st_mtime`, and `st_atime` fields based on the following file attributes:

`st_ctime` - ATR$C_CREDATE (file creation time)
`st_mtime` - ATR$C_REVDATE (file revision time)

`st_atime` - was always set to st_mtime because no support for file access time was available

Also, for the file-modification time, `utime` and `utimes` were modifying the ATR$C_REVDATE file attribute, and ignoring the file-access-time argument.

After the change, for a file on an ODS-5 device, the `stat` and `fstat` functions set the values of the `st_ctime`, `st_mtime`, and `st_atime` fields based on the following new file attributes:

`st_ctime` - ATR$C_ATTDATE (last attribute modification time)
`st_mtime` - ATR$C_MODDATE (last data modification time)
`st_atime` - ATR$C_ACCDATE (last access time)

If ATR$C_ACCDATE is zero, as on an ODS-2 device, the `stat` and `fstat` functions set `st_atime` to `st_mtime`.

For the file-modification time, the `utime` and `utimes` functions modify both the ATR$C_REVDATE and ATR$C_MODDATE file attributes. For the file-access time, these functions modify the ATR$C_ACCDATE file attribute. Setting the ATR$C_MODDATE and ATR$C_ACCDATE file attributes on an ODS-2 device has no effect.

For compatibility, the old behavior of `stat`, `fstat`, `utime`, and `utimes` remains the default, regardless of the kind of device.

The new behavior must be explicitly enabled by defining the DECC$EFS_FILE_TIMESTAMPS logical name to ENABLE before invoking the application. Setting this logical does not affect the behavior of `stat`, `fstat`, `utime`, and `utimes` for files on an ODS-2 device.

Return Values

0

Indicates success.

-1

Indicates an error other than a privilege violation; `errno` is set to indicate the error.

-2

Indicates a privilege violation.

statvfs

statvfs — Gets information about a device containing the specified file.

Format

```
#include <statvfs.h>
int statvfs (const char *restrict path, struct statvfs *restrict buffer);
```

Arguments

path

Any file on a mounted device.

buffer

Pointer to a `statvfs` structure to hold the returned information.

Description

The `statvfs` function returns descriptive information about the device containing the specified file. Read, write, or execute permission of the specified file is not required. The returned information is in the format of a `statvfs` structure, which is defined in the `<statvfs.h>` header file and contains the following members:

`unsigned long f_bsize` - Preferred block size.
`unsigned long f_frsize` - Fundamental block size.
`fsblkcnt_t f_blocks` - Total number of blocks in units of `f_frsize`.
`fsblkcnt_t f_bfree` - Total number of free blocks. If `f_bfree` would assume a meaningless value due to the misreporting of free block count by $GETDVI for a DFS disk, then `f_bfree` is set to the maximum block count.
`fsblkcnt_t f_bavail` - Number of free blocks available. Set to the unused portion of the caller's disk quota.
`fsfilcnt_t f_files` - Total number of file serial numbers (for example, inodes).
`fsfilcnt_t f_ffree` - Total number of free file serial numbers. For OpenVMS systems, this value is calculated as freeblocks/clustersize.
`fsfilcnt_t f_favail` - Number of file serial numbers available to a non-privileged process (0 for OpenVMS systems).
`unsigned long f_fsid` - File system identifier. This identifier is based on the allocation-class device name. This gives a unique value based on device, as long as the device is locally mounted.
`unsigned long f_flag` - Bit mask representing one or more of the following flags:

- `ST_RONLY` - The volume is read-only.
- `ST_NOSUID` - The volume has protected subsystems enabled.

`unsigned long f_namemax` - Maximum length of a filename.
`char f_basetype[64]` - Device-type name.
`char f_fstr[64]` - Logical volume name.
`char __reserved[64]` - Media type name.

Upon successful completion, `statvfs` returns 0 (zero). Otherwise, it returns -1 and sets `errno` to indicate the error.

See also `fstatvfs`.

Return Value

0

Successful completion.

-1

Indicates an error. `errno` is set to one of the following:

- EACCES - Search permission is denied for a component of the path prefix.
- EIO - An I/O error occurred while reading the device.

- EINTR - A signal was intercepted during execution of the function.
- EOVERFLOW - One of the values to be returned cannot be represented correctly in the structure pointed to by *buffer*.
- ENAMETOOLONG - The length of a component of the path parameter exceeds NAME_MAX, or the length of the path parameter exceeds PATH_MAX.
- ENOENT - A component of *path* does not name an existing file, or *path* is an empty string.
- ENOTDIR - A component of the path prefix of the *path* parameter is not a directory.

stpcpy

stpcpy — Copies a string returning a pointer to its end.

Format

```
#include <string.h>
char *stpcpy(char *dest, const char *src);
```

Function Variants

The `stpcpy` function has variants named `_stpcpy32` and `_stpcpy64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions

Arguments

dest

A pointer to the destination character string.

src

A pointer to the character string to be copied.

Description

The function `stpcpy` uses `strlen` to determine the length of *src* and then copies the *src* to *dest*. The difference from the `strcpy` function is that `stpcpy` returns a pointer to the final '\0', and not to the beginning of the line.

Return Value

x

A pointer to the end of the string *dest*.

strcasecmp

strcasecmp — Does a case-insensitive comparison of two 7-bit ASCII strings.

Format

```
#include <strings.h>
int strcasecmp (const char *s1, const char *s2);
```

Arguments

s1

The first of two strings to compare.

s2

The second of two strings to compare.

Description

The `strcasecmp` function is case-insensitive. The returned lexicographic difference reflects a conversion to lowercase.

The `strcasecmp` function works for 7-bit ASCII compares only. Do not use this function for internationalized applications.

Return Value

n

An integer value greater than, equal to, or less than 0 (zero), depending on whether the *s1* string is greater than, equal to, or less than the *s2* string.

strcat

strcat — Concatenates *str_2*, including the terminating null character, to the end of *str_1*.

Format

```
#include <string.h>
char *strcat (char *str_1, const char *str_2);
```

Function Variants

The `strcat` function has variants named `_strcat32` and `_strcat64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

str_1, str_2

Pointers to null-terminated character strings.

Description

See `strncat`.

Return Value

x

The address of the first argument, *str_1*, which is assumed to be large enough to hold the concatenated result.

Example

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

/* This program concatenates two strings using the strcat       */
/* function, and then manually compares the result of strcat    */
/* to the expected result.                                      */

#define S1LENGTH 10
#define S2LENGTH 8

main()
{
    static char s1buf[S1LENGTH + S2LENGTH] = "abcmnexyz";
    static char s2buf[] = " orthis";
    static char test1[] = "abcmnexyz orthis";

    int i;
    char *status;

    /* Take static buffer s1buf, concatenate static buffer      */
    /* s2buf to it, and compare the answer in s1buf with the    */
    /* static answer in test1.                                  */

    status = strcat(s1buf, s2buf);
    for (i = 0; i <= S1LENGTH + S2LENGTH - 2; i++) {
        /* Check for correct returned string.   */

        if (test1[i] != s1buf[i])
            printf("error in strcat");
    }
}
```

strchr

strchr — Returns the address of the first occurrence of a given character in a null-terminated string.

Format

```
#include <string.h>
char *strchr (const char *str, int character);
```

Function Variants

The `strchr` function has variants named `_strchr32` and `_strchr64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

str

A pointer to a null-terminated character string.

character

An object of type `int`.

Description

This function returns the address of the *first* occurrence of a given character in a null-terminated string. The terminating null character is considered to be part of the string.

Compare with `strrchr`, which returns the address of the *last* occurrence of a given character in a null-terminated string.

Return Values

x

The address of the first occurrence of the specified character.

NULL

Indicates that the character does not occur in the string.

Example

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

main()
{

    static char s1buf[] = "abcdefghijkl lkjihgfedcba";

    int i;

    char *status;

/*  This program checks the strchr function by incrementally   */
/*  going through a string that ascends to the middle and then */
/*  descends towards the end.                                  */

    for (i = 0; s1buf[i] != '\0' && s1buf[i] != ' '; i++) {
        status = strchr(s1buf, s1buf[i]);

/* Check for pointer to leftmost character - test 1.           */

        if (status != &s1buf[i])
            printf("error in strchr");
    }
```

```
}
```

strcmp

strcmp — Compares two ASCII character strings and returns a negative, 0, or positive integer, indicating that the ASCII values of the individual characters in the first string are less than, equal to, or greater than the values in the second string.

Format

```
#include <string.h>
int strcmp (const char *str_1, const char*str_2);
```

Arguments

str_1, str_2

Pointers to character strings.

Description

The strings are compared until a null character is encountered or until the strings differ.

Return Values

< 0

Indicates that *str_1* is less than *str_2*.

= 0

Indicates that *str_1* equals *str_2*.

> 0

Indicates that *str_1* is greater than *str_2*.

strcoll

strcoll — Compares two strings and returns an integer that indicates if the strings differ and how they differ. The function uses the collating information in the LC_COLLATE category of the current locale to determine how the comparison is performed.

Format

```
#include <string.h>
int strcoll (const char *s1, const char *s2);
```

Arguments

s1, s2

Pointers to character strings.

Description

The `strcoll` function, unlike `strcmp`, compares two strings in a locale-dependent manner. Because no value is reserved for error indication, the application must check for one by setting `errno` to 0 before the function call and testing it after the call.

See also `strxfrm`.

Return Values

< 0

Indicates that *s1* is less than *s2*.

= 0

Indicates that the strings are equal.

> 0

Indicates that *s1* is greater than *s2*.

strcpy

strcpy — Copies all of *source*, including the terminating null character, into *dest*.

Format

```
#include <string.h>
char *strcpy (char *dest, const char *source);
```

Function Variants

The `strcpy` function has variants named `_strcpy32` and `_strcpy64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

Pointer to the destination character string.

source

Pointer to the source character string.

Description

The `strcpy` function copies *source* into *dest*, and stops after copying *source*'s null character.

The behavior of this function is undefined if the area pointed to by *dest* overlaps the area pointed to by *source*.

Return Value

x

The address of *dest*.

strcspn

strcspn — Returns the length of the prefix of a string that consists entirely of characters not in a specified set of characters.

Format

```
#include <string.h>
size_t strcspn (const char *str, const char *charset);
```

Arguments

str

A pointer to a character string. If this character string is a null string, 0 is returned.

charset

A pointer to a character string containing the set of characters.

Description

The `strcspn` function scans the characters in the string, stops when it encounters a character found in *charset*, and returns the length of the string's initial segment formed by characters not found in *charset*.

If none of the characters match in the character strings pointed to by *str* and *charset*, `strcspn` returns the length of string.

Return Value

x

The length of the segment.

strdup

strdup — Duplicates the specified string.

Format

```
#include <string.h>
char *strdup (const char *s1);
```

Function Variants

The `strdup` function has variants named `_strdup32` and `_strdup64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

s1

The string to be duplicated.

Description

The `strdup` function returns a pointer to a string that is an exact duplicate of the string pointed to by *s1*. The `malloc` function is used to allocate space for the new string. The `strdup` function is provided for compatibility with existing systems.

Return Values

x

A pointer to the resulting string.

NULL

Indicates an error.

strerror

strerror — Maps the error number in *error_code* to a locale-dependent error message string.

Format

```
#include <string.h>
char *strerror (int error_code); (ANSI C)
char *strerror (int error_code[, int vms_error_code]); (VSI C Extension)
```

Arguments

error_code

An error code.

vms_error_code

An OpenVMS error code.

Description

The `strerror` function uses the error number in *error_code* to retrieve the appropriate locale-dependent error message. The contents of the error message strings are determined by the LC_MESSAGES category of the program's current locale.

When a program is not compiled with any standards-related feature-test macros (see Section 1.4.1), `strerror` has a second argument (*vms_error_code*), which is used in the following way:

- If *error_code* is EVMSERR and there is a second argument, then that second argument is used as the `vaxc$errno` value.
- If *error_code* is EVMSERR and there is no second argument, look at `vaxc$errno` to get the OpenVMS error condition.

See the Example section.

Use of the second argument is not included in the ANSI C definition of `strerror` and is, therefore, not portable.

Because no return value is reserved to indicate an error, applications should set the value of `errno` to 0, call `strerror`, and then test the value of `errno`; a nonzero value indicates an error condition.

Return Value

x

A pointer to a buffer containing the appropriate error message. Do not modify this buffer in your programs. Moreover, calls to the `strerror` function may overwrite this buffer with a new message.

Example

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <ssdef.h>

main()
{
    puts(strerror(EVMSERR));
    errno = EVMSERR;
    vaxc$errno = SS$_LINKEXIT;
    puts(strerror(errno));
    puts(strerror(EVMSERR, SS$_ABORT));
    exit(1);
}
```

Running this example produces the following output:

```
nontranslatable vms error code: <none>
network partner exited
abort
```

strerror_r

strerror_r — Maps the error number in *error_code* to a locale-dependent error message string (reentrant).

Format

```
#include <string.h>
int strerror_r(int error_code, char *buf, size_t buflen);
```

Arguments

error_code

An error code.

buf

A pointer to a buffer where the function can store the error message.

buflen

The length of the buffer (in characters).

Description

The `strerror_r` function is the reentrant version of `strerror`. The `strerror_r` function uses the error number in *error_code* to retrieve the appropriate locale dependent error message. The contents of the error message strings are determined by the LC_MESSAGES category of the program's current locale.

If *error_code* is EVMSERR, the function looks at `vaxc$errno` to get the OpenVMS error condition.

Return Values

0

On success, the error message is in the character array pointed to by *buf*. The array is *buflen* characters long and should have space for the error message and the terminating null character.

strfmon

strfmon — Converts a number of monetary values into a string. The conversion is controlled by a format string.

Format

```
#include <monetary.h>
ssize_t strfmon (char *s, size_t maxsize, const char *format, ...);
```

Arguments

s

A pointer to the resultant string.

maxsize

The maximum number of bytes to be stored in the resultant string.

format

A pointer to a string that controls the format of the output string.

. . .

The monetary values of type `double` that are to be formatted for the output string. There should be as many values as there are conversion specifications in the format string pointed to by *format*. The function fails if there are insufficient values. Excess arguments are ignored.

Description

The `strfmon` function creates a string pointed to by *s*, using the monetary values supplied. A maximum of *maxsize* bytes is copied to *s*.

The format string pointed to by *format* consists of ordinary characters and conversion specifications. All ordinary characters are copied unchanged to the output string. A conversion specification defines how one of the monetary values supplied is formatted in the output string.

A conversion specification consists of a percent character (%), followed by a number of optional characters (see Table 37), and concluding with a conversion specifier (see Table 38).

If any of the optional characters listed in Table 37 is included in a conversion specification, they must appear in the order shown.

Table 37. Optional Characters in strfmon Conversion Specifications

Character	Meaning
= *character*	Use *character* as the numeric fill character if a left precision is specified. The default numeric fill character is the space character. The fill character must be representable as a single byte in order to work with precision and width count. This conversion specifier is ignored unless a left precision is specified, and it does not affect width filling, which always uses the space character.
^	Do not use separator characters to format the number. By default, the digits are grouped according to the *mon_grouping* field in the LC_MONETARY category of the current locale.
+	Add the string specified by the *positive_sign* or *negative_sign* fields in the current locale. If *p_sign_posn* or *n_sign_posn* is set to 0, then parentheses are used by default to indicate negative values. Otherwise, sign strings are used to indicate the sign of the value. You cannot use a + and a (in the same conversion specification.
(	Enclose negative values within parentheses. The default is taken from the *p_sign_posn* and *n_sign_posn* fields in the current locale. If *p_sign_posn* or *n_sign_posn* is set to 0, then parentheses are used by default to indicate negative values. Otherwise, sign strings are used to indicate the sign of the value. You cannot use a + and (in the same conversion specification.
!	Suppress the currency symbol. By default, the currency symbol is included.
--	Left-justify the value within the field. By default, values are right-justified.
field width	A decimal integer that specifies the minimum field width in which to align the result of the conversion. The default field width is the smallest field that can contain the result.

#left_precision	A # followed by a decimal integer specifies the number of digits to the left of the radix character. Extra positions are filled by the fill character. By default the precision is the smallest required for the argument. If grouping is not suppressed with the ^ conversion specifier, and if grouping is defined for the current locale, grouping separators are inserted before any fill characters are added. Grouping separators are not applied to fill characters even if the fill character is defined as a digit.
.right_precision	A period (.) followed by a decimal integer specifies the number of digits to the right of the radix character. Extra positions are filled with zeros. The amount is rounded to this number of decimal places. If the right precision is zero, the radix character is not included in the output. By default the right precision is defined by the *frac_digits* or *int_frac_digits* field of the current locale.

Table 38. strfmon Conversion Specifiers

Specifier	Meaning
i	Use the international currency symbol defined by the *int_currency_symbol* field in the current locale, unless the currency symbol has been suppressed.
n	Use the local currency symbol defined by the *currency_symbol* field in the current locale, unless the currency symbol has been suppressed.
%	Output a % character. The conversion specification must be %%; none of the optional characters is valid with this specifier.

Return Values

x

The number of bytes written to the string pointed to by *s*, not including the null-terminating character.

-1

Indicates an error. The function sets `errno` to one of the following values:

- EINVAL – A conversion specification is syntactically incorrect.
- E2BIG – Processing the complete format string would produce more than *maxsize* bytes.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <locale.h>
#include <monetary.h>
#include <errno.h>

#define MAX_BUF_SIZE 124

main()
{
  size_t ret;
  char buffer[MAX_BUF_SIZE];
```

```
  double amount = 102593421;

 /* Display a monetary amount using the en_US.ISO8859-1 */
 /* locale and a range of different display formats.    */

  if (setlocale(LC_ALL, "en_US.ISO8859-1") == (char *) NULL) {
      perror("setlocale");
      exit(EXIT_FAILURE);
  }
  ret = strfmon(buffer, MAX_BUF_SIZE, "International: %i\n", amount);
  printf(buffer);

  ret = strfmon(buffer, MAX_BUF_SIZE, "National:      %n\n", amount);
  printf(buffer);

  ret = strfmon(buffer, MAX_BUF_SIZE, "National:      %=*#10n\n", amount);
  printf(buffer);

  ret = strfmon(buffer, MAX_BUF_SIZE, "National:     %(n\n", -1 * amount);
  printf(buffer);

  ret = strfmon(buffer, MAX_BUF_SIZE, "National:      %^!n\n", amount);
  printf(buffer);
}
```

Running the example program produces the following result:

```
International: USD 102,593,421.00
National:      $102,593,421.00
National:      $**102,593,421.00
National:      ($102,593,421.00)
National:      102593421.00
```

strftime

strftime — Uses date and time information stored in a `tm` structure to create an output string. The format of the output string is controlled by a format string.

Format

```
#include <time.h>
size_t strftime (char *s, size_t maxsize, const char *format,
const struct tm *timeptr);
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `strftime` function that is equivalent to the behavior before OpenVMS Version 7.0.

Arguments

s

A pointer to the resultant string.

maxsize

The maximum number of bytes to be stored in the resultant string, including the null terminator.

format

A pointer to a string that controls the format of the output string.

timeptr

A pointer to the local time (`tm`) structure. The `tm` structure is defined in the `<time.h>` header file.

Description

The `strftime` function uses data in the structure pointed to by *timeptr* to create the string pointed to by *s*. A maximum of *maxsize* bytes is copied to *s*.

The format string consists of zero or more conversion specifications and ordinary characters. All ordinary characters (including the terminating null character) are copied unchanged into the output string. A conversion specification defines how data in the `tm` structure is formatted in the output string.

A conversion specification consists of a percent (%) character followed by one or more optional characters (see Table 39), and concluding with a conversion specifier (see Table 40). If any of the optional characters listed in Table 39 are specified, they must appear in the order shown in the table.

The `strftime` function behaves as if it called `tzset`.

Table 39. Optional Elements of `strftime` Conversion Specifications

Element	Meaning
–	Optional with the field width to specify that the field is left-justified and padded with spaces. This cannot be used with the 0 element.
0	Optional with the field width to specify that the field is right-justified and padded with zeros. This cannot be used with the – element.
field width	A decimal integer that specifies the maximum field width
.precision	A decimal integer that specifies the precision of data in a field. For the `d, H, I, j, m, M, o, S, U, w, W, y,` and `Y` conversion specifiers, the precision specifier is the minimum number of digits to appear in the field. If the conversion specification has fewer digits than that specified by the precision, leading zeros are added. For the `a, A, b, B, c, D, E, h, n, N, p, r, t, T, x, X, Z,` and % conversion specifiers, the precision specifier is the maximum number of characters to appear in the field. If the conversion specification has more characters than that specified by the precision, characters are truncated on the right. The default precision for the `d, H, I, m, M, o, S, U, w, W, y` and `Y` conversion specifiers is 2; the default precision for the `j` conversion specifier is 3.

Note that the list of conversion specifications in Table 39 are extensions to the XPG4 specification.

Table 40 lists the conversion specifiers. The `strftime` function uses fields in the LC_TIME category of the program's current locale to provide a value. For example, if `%B` is specified, the function accesses the *mon* field in LC_TIME to find the full month name for the month specified in the `tm` structure. The result of using invalid conversion specifiers is undefined.

Table 40. `strftime` Conversion Specifiers

Specifier	Replaced by
`%a`	The locale's abbreviated weekday name.
`%A`	The locale's full weekday name.
`%b`	The locale's abbreviated month name.
`%B`	The locale's full month name.
`%c`	The locale's appropriate date and time representation.
`%C`	The century number (the year divided by 100 and truncated to an integer) as a decimal number (00 – 99).
`%d`	The day of the month as a decimal number (01 – 31).
`%D`	Same as `%m/%d/%y`.
`%e`	The day of the month as a decimal number (1 – 31) in a 2-digit field with the leading space character fill.
`%Ec`	The locale's alternative date and time representation.
`%EC`	The name of the base year (period) in the locale's alternative representation.
`%Ex`	The locale's alternative date representation.
`%EX`	The locale's alternative time representation.
`%Ey`	The offset from the base year (`%EC`) in the locale's alternative representation.
`%EY`	The locale's full alternative year representation.
`%F`	Same as "`%Y#%m#%d`" (the ISO 8601 date format). [tm_year, tm_mon, tm_mday]
`%g`	The last 2 digits of the week-based year as a decimal number (00–99). [tm_year, tm_wday, tm_yday]
`%G`	The week-based year as a decimal number (for example, 1997). [tm_year, tm_wday, tm_yday]
`%h`	Same as `%b`.
`%H`	The hour (24-hour clock) as a decimal number (00 – 23).
`%I`	The hour (12-hour clock) as a decimal number (01 – 12).
`%j`	The day of the year as a decimal number (001 – 366).
`%k`	The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank.
`%l`	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank.
`%m`	The month as a decimal number (01 – 12).
`%M`	The minute as a decimal number (00 – 59).
`%n`	The new-line character.
`%Od`	The day of the month using the locale's alternative numeric symbols.
`%Oe`	The date of the month using the locale's alternative numeric symbols.

Specifier	Replaced by
%OH	The hour (24-hour clock) using the locale's alternative numeric symbols.
%OI	The hour (12-hour clock) using the locale's alternative numeric symbols.
%Om	The month using the locale's alternative numeric symbols.
%OM	The minutes using the locale's alternative numeric symbols.
%OS	The seconds using the locale's alternative numeric symbols.
%Ou	The weekday as a number in the locale's alternative representation (Monday=1).
%OU	The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.
%OV	The week number of the year (Monday as the first day of the week) as a decimal number (01– 53) using the locale's alternative numeric symbols. If the week containing January 1 has four or more days in the new year, it is considered as week 1. Otherwise, it is considered as week 53 of the previous year, and the next week is week 1.
%Ow	The weekday as a number (Sunday=0) using the locale's alternative numeric symbols.
%OW	The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%Oy	The year without the century using the locale's alternative numeric symbols.
%p	The locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%P	Same as %p but in lowercase; "am" or "pm" or a corresponding string for the current locale.
%r	The time in AM/PM notation.
%R	The time in 24-hour notation (%H:%M).
%s	The number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).
%S	The second as a decimal number (00 – 61).
%t	The tab character.
%T	The time (%H:%M:%S).
%u	The weekday as a decimal number between 1 and 7 (Monday=1).
%U	The week number of the year (the first Sunday as the first day of week 1) as a decimal number (00 – 53).
%V	The week number of the year (Monday as the first day of the week) as a decimal number (00 – 53). If the week containing January 1 has four or more days in the new year, it is considered as week 1. Otherwise, it is considered as week 53 of the previous year, and the next week is week 1.
%w	The weekday as a decimal number (0 [Sunday] – 6).
%W	The week number of the year (the first Monday as the first day of week 1) as a decimal number (00– 53).
%x	The locale's appropriate date representation.
%X	The locale's appropriate time representation.
%y	The year without century as a decimal number (00 – 99).
%Y	The year with century as a decimal number.
%z	The +hhmm or -hhmm numeric timezone (that is, the hour and minute offset from UTC).

Specifier	Replaced by
%Z	The time-zone name or abbreviation. If time-zone information is not available, no character is output.
%+	The date and time in date format.
%%	Literal % character.

Return Values

x

The number of characters placed into the array pointed to by *s*, not including the terminating null character.

0

Indicates an error occurred. The contents of the array are indeterminate.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <locale.h>
#include <errno.h>

#define NUM_OF_DATES  7
#define BUF_SIZE 256

/* This program formats a number of different dates, once */
/* using the C locale and then using the fr_FR.ISO8859-1  */
/* locale. Date and time formatting is done using strftime(). */

main()
{
    int count,
        i;
    char buffer[BUF_SIZE];
    struct tm *tm_ptr;
    time_t time_list[NUM_OF_DATES] =
    {500, 68200000, 694223999, 694224000,
     704900000, 705000000, 705900000};

    /* Display dates using the C locale */
    printf("\nUsing the C locale:\n\n");

    setlocale(LC_ALL, "C");

    for (i = 0; i < NUM_OF_DATES; i++) {
        /* Convert to a tm structure */
        tm_ptr = localtime(&time_list[i]);

        /* Format the date and time */
        count = strftime(buffer, BUF_SIZE,
               "Date: %A %d %B %Y%nTime: %T%n%n", tm_ptr);
```

```
        if (count == 0) {
            perror("strftime");
            exit(EXIT_FAILURE);
        }

        /* Print the result */
       printf(buffer);
    }

    /* Display dates using the fr_FR.ISO8859-1 locale */
    printf("\nUsing the fr_FR.ISO8859-1 locale:\n\n");

    setlocale(LC_ALL, "fr_FR.ISO8859-1");

    for (i = 0; i < NUM_OF_DATES; i++) {
        /* Convert to a tm structure */
        tm_ptr = localtime(&time_list[i]);

        /* Format the date and time */
        count = strftime(buffer, BUF_SIZE,
               "Date: %A %d %B %Y%nTime: %T%n%n", tm_ptr);
        if (count == 0) {
            perror("strftime");
            exit(EXIT_FAILURE);
        }

        /* Print the result */
        printf(buffer);
    }
}
```

Running the example program produces the following result:

```
Using the C locale:

Date: Thursday 01 January 1970
Time: 00:08:20

Date: Tuesday 29 February 1972
Time: 08:26:40

Date: Tuesday 31 December 1991
Time: 23:59:59

Date: Wednesday 01 January 1992
Time: 00:00:00

Date: Sunday 03 May 1992
Time: 13:33:20

Date: Monday 04 May 1992
Time: 17:20:00

Date: Friday 15 May 1992
Time: 03:20:00


Using the fr_FR.ISO8859-1 locale:
```

```
Date: jeudi 01 janvier 1970
Time: 00:08:20

Date: mardi 29 février 1972
Time: 08:26:40

Date: mardi 31 décembre 1991
Time: 23:59:59

Date: mercredi 01 janvier 1992
Time: 00:00:00

Date: dimanche 03 mai 1992
Time: 13:33:20

Date: lundi 04 mai 1992
Time: 17:20:00

Date: vendredi 15 mai 1992
Time: 03:20:00
```

strlen

strlen — Returns the length of a string of ASCII characters. The returned length does not include the terminating null character (\0).

Format

```
#include <string.h>
size_t strlen (const char *str);
```

Argument

str

A pointer to the character string.

Return Value

x

The length of the string.

strncasecmp

strncasecmp — Does a case-insensitive comparison between two 7-bit ASCII strings.

Format

```
#include <strings.h>
int strncasecmp (const char *s1, const char *s2, size_t n);
```

Arguments

s1

The first of two strings to compare.

s2

The second of two strings to compare.

n

The maximum number of bytes in a string to compare.

Description

The `strncasecmp` function is case-insensitive. The returned lexicographic difference reflects a conversion to lowercase. The `strncasecmp` function is similar to the `strcasecmp` function, but also compares size. If the size specified by *n* is read before a NULL, the comparison stops.

The `strcasecmp` function works for 7-bit ASCII compares only. Do not use this function for internationalized applications.

Return Value

n

An integer value greater than, equal to, or less than 0 (zero), depending on whether *s1* is greater than, equal to, or less than *s2*.

strncat

strncat — Appends not more than *maxchar* characters from *str_2* to the end of *str_1*.

Format

```
#include <string.h>
char *strncat (char *str_1, const char *str_2, size_t maxchar);
```

Function Variants

The `strncat` function has variants named `_strncat32` and `_strncat64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

str_1, str_2

Pointers to null-terminated character strings.

maxchar

The number of characters to concatenate from *str_2*, unless `strncat` first encounters a null terminator in *str_2*. If *maxchar* is 0, no characters are copied from *str_2*.

Description

A null character is always appended to the result of the `strncat` function. If `strncat` reaches the specified maximum, it sets the next byte in *str_1* to the null character.

Return Value

x

The address of the first argument, *str_1*, which is assumed to be large enough to hold the concatenated result.

strncmp

strncmp — Compares not more than *maxchar* characters of two ASCII character strings and returns a negative, 0, or positive integer, indicating that the ASCII values of the individual characters in the first string are less than, equal to, or greater than the values in the second string.

Format

```
#include <string.h>
int strncmp (const char *str_1, const char *str_2, size_t maxchar);
```

Arguments

str_1, str_2

Pointers to character strings.

maxchar

The maximum number of characters (beginning with the first) to search in both *str_1* and *str_2*.If *maxchar* is 0, no comparison is performed and 0 is returned (the strings are considered equal).

Description

The `strncmp` function compares no more than *maxchar* characters from the string pointed to by *str_1* to the string pointed to by *str_2*. The strings are compared until a null character is encountered, the strings differ, or *maxchar* is reached. Characters that follow a difference or a null character are not compared.

Return Values

< 0

Indicates that *str_1* is less than *str_2*.

= 0

Indicates that *str_1* equals *str_2*.

> 0

Indicates that *str_1* is greater than *str_2*.

Examples

1. ```
   #include <string.h>
   #include <stdio.h>

   main()
   {
       printf( "%d\n", strncmp("abcde", "abc", 3));
   }
   ```

   When linked and executed, this example returns 0, because the first 3 characters of the 2 strings are equal:

   ```
   $ run tmp
   0
   ```

2. ```
   #include <string.h>
   #include <stdio.h>

       main()
       {
           printf( "%d\n", strncmp("abcde", "abc", 4));
       }
   ```

   When linked and executed, this example returns a value greater than 0 because the first 4 characters of the 2 strings are not equal (The "d" in the first string is not equal to the null character in the second):

   ```
   $ run tmp
           100
   ```

strncpy

strncpy — Copies not more than *maxchar* characters from *source* into *dest*.

Format

```
#include <string.h>
char *strncpy (char *dest, const char *source, size_t maxchar);
```

Function Variants

The `strncpy` function has variants named `_strncpy32` and `_strncpy64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

Pointer to the destination character string.

source

Pointer to the source character string.

maxchar

The maximum number of characters to copy from *source* to *dest* up to but not including the null terminator of *source*.

Description

The `strncpy` function copies no more than *maxchar* characters from *source* to *dest*, up to but not including the null terminator of *source*. If *source* contains less than *maxchar* characters, *dest* is padded with null characters. If *source* contains greater than or equal to *maxchar* characters, as many characters as possible are copied to *dest*. Be aware that the *dest* argument might not be terminated by a null character after a call to `strncpy`.

Return Value

x

The address of *dest*.

strndup

strndup — Duplicates a specific number of bytes from a string.

Format

```
#include <string.h>
char *strndup(const char *s, size_t size);
```

Function Variants

The `strndup` function has variants named `_strndup32` and `_strndup64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s

A pointer to the null-terminated byte string.

size

The number of bytes to duplicate from *s*.

Description

The `strndup` function duplicates a specific number of bytes from a string. The `strndup` function is equivalent to the `strdup` function, duplicating the provided string in a new block of memory allocated as if by using `malloc`, with the exception that `strndup` copies at most *size* plus one bytes into the newly allocated memory, terminating the new string with a NUL character.

If the length of *s* is larger than *size*, only *size* bytes will be duplicated. If *size* is larger than the length of *s*, all bytes in *s* will be copied into the new memory buffer, including the terminating NUL character. The newly created string will always be properly terminated.

Return Values

x

A pointer to the resulting string.

NULL

Indicates an error.

strnlen

strnlen — Returns the number of bytes in a string.

Format

```
#include <string.h>
size_t strnlen (const char *s, size_t n);
```

Arguments

s

Pointer to the string.

n

The maximum number of characters to examine.

Description

The `strnlen` function returns the number of bytes in the string pointed to by *s*. The string length value does not include the terminating null character. The `strnlen` function counts bytes until the first null byte or until *n* bytes have been examined.

Return Value

n

The length of the string.

strpbrk

strpbrk — 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);
```

Function Variants

The `strpbrk` function has variants named `_strpbrk32` and `_strpbrk64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

str

A pointer to a character string. If this character string is a null string, 0 is returned.

charset

A pointer to a character string containing the set of characters for which the function will search.

Description

The `strpbrk` function scans the characters in the string, stops when it encounters a character found in *charset*, and returns the address of the first character in the string that appears in the character set.

Return Values

x

The address of the first character in the string that is in the set.

NULL

Indicates that no character is in the set.

strptime

strptime — Converts a character string into date and time values that are stored in a `tm` structure. Conversion is controlled by a format string.

Format

```
#include <time.h>
char *strptime (const char *buf, const char *format, struct tm *timeptr);
```

Function Variants

The `strptime` function has variants named `_strptime32` and `_strptime64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

buf

A pointer to the character string to convert.

format

A pointer to the string that defines how the input string is converted.

timeptr

A pointer to the local time structure. The `tm` structure is defined in the `<time.h>` header file.

Description

The `strptime` function converts the string pointed to by *buf* into values that are stored in the structure pointed to by *timeptr*. The string pointed to by *format* defines how the conversion is performed.

The `strptime` function modifies only those fields in the `tm` structure that have corresponding conversion specifications in the format. In particular, `strptime` never sets the `tm_isdst` member of the `tm` structure.

The format string consists of zero or more directives. A directive is composed of one of the following:

- One or more white-space characters (as defined by the `isspace` function). This directive causes the function to read input up to the first character that is not a white-space character.
- Any character other than the percent character (%) or a white-space character. This directive causes the function to read the next character. The character read must be the same as the character that comprises the directive. If the character is different, the function fails.
- A conversion specification. A conversion specification defines how characters in the input string are interpreted as values that are then stored in the `tm` structure. A conversion specification consists of a percent (%) character followed by a conversion specifier. Table 41 lists the valid conversion specifications.

The `strptime` function uses fields in the LC_TIME category of the program's current locale to provide a value.

Note

To be compliant with X/Open CAE Specification System Interfaces and Headers Issue 5 (commonly known as XPG5), the `strptime` function processes the "%y" directive differently than in previous versions of the C RTL.

With Version 6.4 and higher of the C compiler, for a two-digit year within the century if no century is specified, "%y" directive values range from:

- 69 to 99 refer to years in the twentieth century (1969 to 1999 inclusive)
- 00 to 68 refer to years in the twenty-first century (2000 to 2068 inclusive)

In previous (XPG4-compliant) versions of the C RTL, `strptime` interpreted a two-digit year with no century specified as a year within the twentieth century.

The XPG5-compliant `strptime` is now the default version in the C RTL.

To obtain the old, XPG4-compliant `strptime` function behavior, specify one of the following:

- Define the DECC$XPG4_STRPTIME logical name as follows:

  ```
  $ DEFINE DECC$XPG4_STRPTIME ENABLE
  ```

  or:

- Call the XPG4 `strptime` directly as the function `decc$strptime_xpg4`.

To return to using the XPG5 `strptime` version, DEASSIGN the DECC$XPG4_STRPTIME logical name:

```
$ DEASSIGN DECC$XPG4_STRPTIME
```

Table 41. `strptime` Conversion Specifiers

Specification	Replaced by
%a	The weekday name. This is either the abbreviated or the full name.
%A	Same as %a.
%b	The month name. This is either the abbreviated or the full name.
%B	Same as %b.
%c	The date and time using the locale's date format.
%Ec	The locale's alternative date and time representation.
%C	The century number (the year divided by 100 and truncated to an integer) as a decimal number (00 – 99). Leading zeros are permitted.
%EC	The name of the base year (period) in the locale's alternative representation.
%d	The day of the month as a decimal number (01 – 31). Leading zeros are permitted.
%Od	The day of the month using the locale's alternative numeric symbols.
%D	Same as %m/%d/%y.
%e	Same as %d.
%Oe	The date of the month using the locale's alternative numeric symbols.
%F	Same as "%Y#%m#%d" (the ISO 8601 date format). [tm_year, tm_mon, tm_mday]
%g	The last 2 digits of the week-based year as a decimal number (00–99). [tm_year, tm_wday, tm_yday]
%G	The week-based year as a decimal number (for example, 1997). [tm_year, tm_wday, tm_yday]
%h	Same as %b.
%H	The hour (24-hour clock) as a decimal number (00 – 23). Leading zeros are permitted.
%OH	The hour (24-hour clock) using the locale's alternative numeric symbols.
%I	The hour (12-hour clock) as a decimal number (01 – 12). Leading zeros are permitted.
%OI	The hour (12-hour clock) using the locale's alternative numeric symbols.
%j	The day of the year as a decimal number (001 – 366).
%k	The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank.
%l	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank.
%m	The month as a decimal number (01 – 12). Leading zeros are permitted.
%Om	The month using the locale's alternative numeric symbols.
%M	The minute as a decimal number (00 – 59). Leading zeros are permitted.
%OM	The minutes using the locale's alternative numeric symbols.

Specification	Replaced by
%n	Any white-space character.
%p	The locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%P	Same as %p but in lowercase: "am" or "pm" or a corresponding string for the current locale.
%r	The time in AM/PM notation (%I:%M:%S %p).
%R	The time in 24-hour notation (%H:%M).
%s	The number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).
%S	The second as a decimal number (00 – 61). Leading zeros are permitted.
%OS	The seconds using the locale's alternative numeric symbols.
%t	Any white-space character.
%T	The time (%H:%M:%S).
%u	The weekday as a decimal number between 1 and 7 (Monday=1).
%U	The week number of the year (the first Sunday as the first day of week 1) as a decimal number (00 – 53). Leading zeros are permitted.
%OU	The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.
%w	The weekday as a decimal number (0 [Sunday] – 6). Leading zeros are permitted.
%Ow	The weekday as a number (Sunday=0) using the locale's alternative numeric symbols.
%W	The week number of the year (the first Monday as the first day of week 1) as a decimal number (00 – 53). Leading zeros are permitted.
%OW	The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%x	The locale's appropriate date representation.
%Ex	The locale's alternative date representation.
%EX	The locale's alternative time representation.
%X	The locale's appropriate time representation.
%y	The year without century as a decimal number (00– 99).
%Ey	The offset from the base year (%EC) in the locale's alternative representation.
%Oy	The year without the century using the locale's alternative numeric symbols.
%Y	The year with century as a decimal number.
%EY	The locale's full alternative year representation.
%z	The +hhmm or -hhmm numeric timezone (that is, the hour and minute offset from UTC).
%Z	The time-zone name or abbreviation. If time-zone information is not available, no character is output.
%+	The date and time in date format.
%%	Literal % character.

Return Values

x

A pointer to the character following the last character parsed.

NULL

Indicates that an error occurred. The contents of the `tm` structure are undefined.

Example

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <locale.h>
#include <errno.h>

#define NUM_OF_DATES  7
#define BUF_SIZE 256

/* This program takes a number of date and time strings and     */
/* converts them into tm structs using strptime().  These tm    */
/* structs are then passed to strftime() which will reverse the */
/* process.  The resulting strings are then compared with the   */
/* originals and if a difference is found then an error is      */
/* displayed.                                                   */

main()
{
int count,
    i;
char buffer[BUF_SIZE];
char *ret_val;
struct tm time_struct;
char dates[NUM_OF_DATES][BUF_SIZE] =
{
    "Thursday 01 January 1970 00:08:20",
    "Tuesday 29 February 1972 08:26:40",
    "Tuesday 31 December 1991 23:59:59",
    "Wednesday 01 January 1992 00:00:00",
    "Sunday 03 May 1992 13:33:20",
    "Monday 04 May 1992 17:20:00",
    "Friday 15 May 1992 03:20:00"};

for (i = 0; i < NUM_OF_DATES; i++) {
    /* Convert to a tm structure */
 ret_val = strptime(dates[i], "%A %d %B %Y %T", &time_struct);

    /* Check the return value */
    if (ret_val == (char *) NULL) {
        perror("strptime");
        exit(EXIT_FAILURE);
    }
```

```
    /* Convert the time structure back to a formatted string */
count = strftime(buffer, BUF_SIZE, "%A %d %B %Y %T",&time_struct);

    /* Check the return value */
    if (count == 0) {
        perror("strftime");
        exit(EXIT_FAILURE);
    }

    /* Check the result */
    if (strcmp(buffer, dates[i]) != 0) {
    printf("Error: Converted string differs from the original\n");
    }
    else {
        printf("Successfully converted <%s>\n", dates[i]);
    }
 }
}
```

Running the example program produces the following result:

```
Successfully converted <Thursday 01 January 1970 00:08:20>
Successfully converted <Tuesday 29 February 1972 08:26:40>
Successfully converted <Tuesday 31 December 1991 23:59:59>
Successfully converted <Wednesday 01 January 1992 00:00:00>
Successfully converted <Sunday 03 May 1992 13:33:20>
Successfully converted <Monday 04 May 1992 17:20:00>
Successfully converted <Friday 15 May 1992 03:20:00>
```

strrchr

strrchr — Returns the address of the last occurrence of a given character in a null-terminated string.

Format

```
#include <string.h>
char *strrchr (const char *str, int character);
```

Function Variants

The `strrchr` function has variants named `_strrchr32` and `_strrchr64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

str

A pointer to a null-terminated character string.

character

An object of type `int`.

Description

This function returns the address of the *last* occurrence of a given character in a null-terminated string. The terminating null character is considered to be part of the string.

Compare with `strchr`, which returns the address of the *first* occurrence of a given character in a null-terminated string.

Return Values

x

The address of the last occurrence of the specified character.

NULL

Indicates that the character does not occur in the string.

strsep

strsep — Separates strings.

Format

```
#include <string.h>
char *strsep (char **stringp, char *delim);
```

Function Variants

The `strsep` function has variants named `_strsep32` and `_strsep64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

stringp

A pointer to a pointer to a character string.

delim

A pointer to a string containing characters to be used as delimiters.

Description

The `strsep` function locates in *stringp*, the first occurrence of any character in *delim* (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or NULL, if the end of the string is reached) is stored in the *stringp* argument. The original value of the *stringp* argument is returned.

You can detect an "empty" field; one caused by two adjacent delimiter characters, by comparing the location referenced by the pointer returned in the *stringp* argument to '\0'.

The *stringp* argument is initially NULL, `strsep` returns NULL.

Return Values

x

The address of the string pointed to by *stringp*.

NULL

Indicates that *stringp* is NULL.

Example

The following example uses `strsep` to parse a string, containing token delimited by white space, into an argument vector:

```
char **ap, **argv[10], *inputstring;

for (ap = argv; (*ap = strsep(&inputstring, " \t")) != NULL;)
     if (**ap != '\0')
         ++ap;
```

strspn

strspn — Returns the length of the prefix of a string that consists entirely of characters from a set of characters.

Format

```
#include <string.h>
size_t strspn (const char *str, const char *charset);
```

Arguments

str

A pointer to a character string. If this string is a null string, 0 is returned.

charset

A pointer to a character string containing the characters for which the function will search.

Description

The `strspn` function scans the characters in the string, stops when it encounters a character not found in *charset*, and returns the length of the string's initial segment formed by characters found in *charset*.

Return Value

x

The length of the segment.

strstr

strstr — 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);
```

Function Variants

The `strstr` function has variants named `_strstr32` and `_strstr64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s1, s2

Pointers to character strings.

Return Values

Pointer

A pointer to the located string.

NULL

Indicates that the string was not found.

Example

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

main()
{
    static char lookin[]="that this is a test was at the end";

    putchar('\n');
    printf("String: %s\n", &lookin[0] );
    putchar('\n');
    printf("Addr: %s\n", &lookin[0] );
    printf("this: %s\n", strstr( &lookin[0] ,"this") );
    printf("that: %s\n", strstr( &lookin[0] , "that" ) );
    printf("NULL: %s\n", strstr( &lookin[0], "" ) );
    printf("was: %s\n", strstr( &lookin[0], "was" ) );
    printf("at: %s\n", strstr( &lookin[0], "at" ) );
    printf("the end: %s\n", strstr( &lookin[0], "the end") );
    putchar('\n');
```

```
    exit(0);
}
```

This example produces the following results:

```
$ RUN STRSTR_EXAMPLE
String: that this is a test was at the end
Addr: that this is a test was at the end
this: this is a test was at the end
that: that this is a test was at the end
NULL: that this is a test was at the end
was: was at the end
at: at this is a test was at the end
the end: the end
$
```

strtod

strtod — Converts a given string to a double-precision number.

Format

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr);
```

Function Variants

The `strtod` function has variants named `_strtod32` and `_strtod64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

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

endptr

The address of an object where the function can store the address of the first unrecognized character that terminates the scan. If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

Description

The `strtod` function recognizes an optional sequence of white-space characters (as defined by `isspace`), then an optional plus or minus sign, then a sequence of digits optionally containing a radix character, then an optional letter (e or E) followed by an optionally signed integer. The first unrecognized character ends the conversion.

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

The radix character is defined the program's current locale (category LC_NUMERIC).

This function returns the converted value. For `strtod`, overflows are accounted for in the following manner:

- 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.
- If the correct value causes an underflow, 0 is returned and `errno` is set to ERANGE.

If the string starts with an unrecognized character, then the conversion is not performed, **endptr* is set to *nptr*, a 0 value is returned, and `errno` is set to EINVAL.)

Return Values

x

The converted string.

0

Indicates the conversion could not be performed. `errno` is set to one of the following:

- EINVAL - No conversion could be performed.
- ERANGE - The value would cause an underflow.
- ENOMEM - Not enough memory available for internal conversion buffer.

±HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

strtof

strtof — Converts a string to a `float`.

Format

```
#include <stdlib.h>
float strtof (const char * restrict nptr,
char ** restrict endptr);
```

Function Variants

The `strtof` function has variants named `_strtof32` and `_strtof64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argumants

nptr

A pointer to the character string to be converted.

endptr

If this argument is *not* NULL, the function stores in it a pointer to the character that follows the last character used in the conversion.

Description

The `strtof` function converts the initial portion of the string pointed to by *nptr* to `float` representation.

First, this function decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function); a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to a floating-point number, and returns the result as a value of type `float`.

The expected form of the (initial portion of the) string is optional leading white space, an optional plus ('+') or minus sign ('-') and then either (i) a decimal number, or (ii) a hexadecimal number, or (iii) an infinity, or (iv) a NaN (not-a-number).

Return Values

x

The converted value.

±HUGE_VALF

If the correct value is outside the range of representable values of the type, ±HUGE_VALF is returned according to the sign of the value; and `errno` is set to ERANGE.

n

If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number; whether `errno` acquires the value ERANGE is implementation-defined.

0

If no valid conversion could be performed.

strtoimax, strtoumax

strtoimax, strtoumax — Convert a string into an integer.

Format

```
#include <inttypes.h>
intmax_t strtoimax(const char *nptr, char **endptr, int base);
uintmax_t strtoumax(const char *nptr, char **endptr, int base);
```

Function Variants

The `strtoimax` function has variants named `_strtoimax32` and `_strtoimax64` for use with 32-bit and 64-bit pointer sizes, respectively.

The `strtoumax` function has variants named `_strtoumax32` and `_strtoumax64` for use with 32-bit and 64-bit pointer sizes, respectively.

See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the string to be converted.

endptr

If this argument is *not* NULL, the function stores in it a pointer to the character that follows the last character used in the conversion.

base

The base of the conversion.

Description

The `strtoimax` and `strtoumax` functions convert a string of ASCII characters pointed to by *nptr* to the appropriate signed and unsigned numeric values. `strtoimax` is a synonym for `strtoll`; `strtoumax` is a synonym for `strtoull`.

The functions recognize strings in various formats, depending on the value of the *base.* Any leading white-space characters (as defined by `isspace` in `<ctype.h>`) in the given string are ignored. The functions recognize 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 and is pointed to by *endptr.*

If *base* is 16, leading zeros after the optional sign are ignored, and 0x or 0X is ignored.

If *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

n

If successful, the function returns an integer value corresponding to the contents of *nptr*.

INTMAX_MAX, INTMAX_MIN, UINTMAX_MAX

If the correct value is outside the range of representable values, a *range error* occurs; `errno` is set to ERANGE.

0

If no conversion can be performed.

strtok, strtok_r

strtok, strtok_r — Split strings into tokens.

Format

```
#include <string.h>
char *strtok (char *s1, const char *s2);
char *strtok_r (char *s, const char *sep, char **lasts);
```

Function Variants

The `strtok` function has variants named `_strtok32` and `_strtok64` for use with 32-bit and 64-bit pointer sizes, respectively. Likewise, the `strtok_r` function has variants named `_strtok_r32` and `_strtok_r64`. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s1

On the first call, a pointer to a string containing zero or more text tokens. On all subsequent calls for that string, a NULL pointer.

s2

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

s

A null-terminated string that is a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *sep*.

sep

A null-terminated string of separator characters. This separator string can be different from call to call.

lasts

A pointer that points to a user-provided pointer to stored information needed for `strtok_r` to continue scanning the same string.

Description

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.

A token in *s1* starts at the first character that is not a character in the separator string *s2* and ends either at the end of the string or at (but not including) a separator character.

The first call to the `strtok` function returns a pointer to the first character in the first token and writes a null character into *s1* immediately following the returned token. Each subsequent call (with the value of the first argument remaining NULL) returns a pointer to a subsequent token in the string originally pointed to by *s1*. When no tokens remain in the string, the `strtok` function returns a NULL pointer. (This can occur on the first call to `strtok` if the string is empty or contains only separator characters.)

Since `strtok` inserts null characters into *s1* to delimit tokens, *s1* cannot be a `const` object.

The `strtok_r` function is the reentrant version of `strtok`. The function `strtok_r` considers the null-terminated string *s* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *sep*. The *lasts* argument points to a user-provided pointer to stored information needed for `strtok_r` to continue scanning the same string.

In the first call to `strtok_r`, *s* points to a null-terminated string, *sep* points to a null-terminated string of separator characters, and the value pointed to by *lasts* is ignored. The `strtok_r` function returns a pointer to the first character of the first token, writes a null character into *s* immediately following the returned token, and updates the pointer to which *lasts* points.

In subsequent calls, *s* is a NULL pointer and *lasts* is unchanged from the previous call so that subsequent calls move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* can be different from call to call. When no token remains in *s*, a NULL pointer is returned.

Return Values

x

A pointer to the first character of the parsed token in the string.

NULL

Indicates that there are no tokens remaining in the string.

Examples

1.
```
#include <stdio.h>
#include <string.h>

main()
{
    static char str[] = "...ab..cd,,ef.hi";

    printf("|%s|\n", strtok(str, "."));
    printf("|%s|\n", strtok(NULL, ","));
    printf("|%s|\n", strtok(NULL, ",."));
    printf("|%s|\n", strtok(NULL, ",."));
}
```

Running this example program produces the following results:

```
$ RUN STRTOK_EXAMPLE1
|ab|
|.cd|
```

```
|ef|
|hi|
$
```

2.
```
#include <stdio.h>
#include <string.h>

main()
{
   char *ptr,
        string[30];

   /* The first character not in the string "-" is "A".  The   */
   /* token ends at "C.                                        */

    strcpy(string, "ABC");
    ptr = strtok(string, "-");
    printf("|%s|\n", ptr);

    /* Returns NULL because no characters not in separator      */
    /* string "-" were found (i.e.  only separator characters   */
    /* were found)                                              */

    strcpy(string, "-");
    ptr = strtok(string, "-");
    if (ptr == NULL)
        printf("ptr is NULL\n");

}
```

   Running this example program produces the following results:

```
$ RUN STRTOK_EXAMPLE2
|abc|
ptr is NULL
$
```

strtol

strtol — Converts strings of ASCII characters to the appropriate numeric values.

Format

```
#include <stdlib.h>
long int strtol (const char *nptr, char **endptr, int base);
```

Function Variants

The `strtol` function has variants named `_strtol32` and `_strtol64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

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

endptr

The address of an object where the function can store a pointer to the first unrecognized character encountered in the conversion process (that is, the character that follows the last character in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base

The value, 2 through 36, to use as the base for the 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.

Leading zeros after the optional sign are ignored, and 0x or 0X is ignored if the base is 16.

If *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.

Truncation from `long` to `int` can take place after assignment or by an explicit cast (arithmetic exceptions not withstanding). The function call `atol (str)` is equivalent to `strtol (str, (char**)NULL, 10)`.

Return Values

x

The converted value.

LONG_MAX or LONG_MIN

Indicates that the converted value would cause an overflow.

0

Indicates that the string starts with an unrecognized character or that the value for *base* is invalid. If the string starts with an unrecognized character, **endptr* is set to *nptr*.

strtold

strtold — Converts a string to a `long double`.

Format

```
#include <stdlib.h>
long double strtold (const char * restrict nptr,
```

```
char ** restrict endptr);
```

Function Variants

The `strtold` function has variants named `_strtold32` and `_strtold64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the character string to be converted.

endptr

If this argument is *not* NULL, the function stores in it a pointer to the character that follows the last character used in the conversion.

Description

The `strtold` function converts the initial portion of the string pointed to by *nptr* to `long double` representation.

First, this function decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function); a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to a floating-point number, and returns the result as a value of type `long double`.

The expected form of the (initial portion of the) string is optional leading white space, an optional plus ('+') or minus sign ('-') and then either (i) a decimal number, or (ii) a hexadecimal number, or (iii) an infinity, or (iv) a NaN (not-a-number).

Return Values

x

The converted value.

±HUGE_VALF

If the correct value is outside the range of representable values of the type, ±HUGE_VALL is returned according to the sign of the value; and `errno` is set to ERANGE.

n

If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number; whether `errno` acquires the value ERANGE is implementation-defined.

0

If no valid conversion could be performed.

strtoq, strtoll

strtoq, strtoll — Convert strings of ASCII characters to the appropriate numeric values. `strtoll` is a synonym for `strtoq`.

Format

```
#include <stdlib.h>
__int64 strtoq (const char *nptr, char **endptr, int base);
__int64 strtoll (const char *nptr, char **endptr, int base);
```

Function Variants

These functions have variants named `_strtoq32`, `_strtoll32` and `_strtoq64`, `_strtoll64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the character string to be converted to an `__int64`.

endptr

The address of an object where the function can store a pointer to the first unrecognized character encountered in the conversion process (that is, the character that follows the last character in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base

The value, 2 through 36, to use as the base for the conversion.

Description

The `strtoq` and `strtoll` functions recognize strings in various formats, depending on the value of the base. Any leading white-space characters (as defined by `isspace` in `<ctype.h>`) in the given string are ignored. The functions recognize 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.

Leading zeros after the optional sign are ignored, and 0x or 0X is ignored if the base is 16.

If *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.

The function call `atoq (str)` is equivalent to `strtoq (str, (char**)NULL, 10)`.

Return Values

x

The converted value.

__INT64_MAX or __INT64_MIN

Indicates that the converted value would cause an overflow.

0

Indicates that the string starts with an unrecognized character or that the value for *base* is invalid. If the string starts with an unrecognized character, **endptr* is set to *nptr*.

strtoul

strtoul — Converts the initial portion of the string pointed to by *nptr* to an `unsigned long` integer.

Format

```
#include <stdlib.h>
unsigned long int strtoul (const char *nptr, char **endptr, int base);
```

Function Variants

The `strtoul` function has variants named `_strtoul32` and `_strtoul64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the character string to be converted to an `unsigned long`.

endptr

The address of an object where the function can store a pointer to a pointer to the first unrecognized character encountered in the conversion process (that is, the character that follows the last character in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base

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

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

Return Values

x

The converted value.

0

Indicates that the string starts with an unrecognized character or that the value for *base* is invalid. If the string starts with an unrecognized character, **endptr* is set to *nptr*.

ULONG_MAX

Indicates that the converted value would cause an overflow.

strtouq, strtoull

strtouq, strtoull — Convert the initial portion of the string pointed to by *nptr* to an `unsigned __int64` integer. `strtoull` is a synonym for `strtouq`.

Format

```
#include <stdlib.h>
unsigned __int64 strtouq const char *nptr, char **endptr, int base);
unsigned __int64 strtoull (const char *nptr, char **endptr, int base);
```

Function Variants

These functions have variants named `_strtouq32`, `_strtoull32` and `_strtouq64`, `_strtoull64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the character string to be converted to an `unsigned __int64`.

endptr

The address of an object where the function can store a pointer to a pointer to the first unrecognized character encountered in the conversion process (that is, the character that follows the last character in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base

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

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

Return Values

x

The converted value.

0

Indicates that the string starts with an unrecognized character or that the value for *base* is invalid. If the string starts with an unrecognized character, **endptr* is set to *nptr*.

__UINT64_MAX

Indicates that the converted value would cause an overflow.

strxfrm

strxfrm — Changes a string such that the changed string can be passed to the `strcmp` function, and produce the same result as passing the unchanged string to the `strcoll` function.

Format

```
#include <string.h>
size_t strxfrm (char *s1, const char *s2, size_t maxchar);
```

Arguments

s1, s2

Pointers to character strings.

maxchar

The maximum number of bytes (including the null terminator) to be stored in *s1*.

Description

The *strxfrm* function transforms the string pointed to by *s2*, and stores the resulting string in the array pointed to by *s1*. No more than *maxchar* bytes, including the null terminator, are placed into the array pointed to by *s1*.

If the value of *maxchar* is less than the required size to store the transformed string (including the terminating null), the contents of the array pointed to by *s1* is indeterminate. In such a case, the function returns the size of the transformed string.

If *maxchar* is 0, then *s1* is allowed to be a NULL pointer, and the function returns the required size of the *s1* array before making the transformation.

The string comparison functions, `strcoll` and `strcmp`, can produce different results given the same two strings to compare. The reason for this is that `strcmp` does a straightforward comparison of the code point values of the characters in the strings, whereas `strcoll` uses the locale information to do the comparison. Depending on the locale, the `strcoll` comparison can be a multipass operation, which is slower than `strcmp`.

The purpose of the `strxfrm` function is to transform strings in such a way that if you pass two transformed strings to the `strcmp` function, the result is the same as passing the two original strings to the `strcoll` function. The `strxfrm` function is useful in applications that need to do a large number of comparisons on the same strings using `strcoll`. In this case, it might be more efficient (depending on the locale) to transform the strings once using `strxfrm`, and then do comparisons using `strcmp`.

Return Value

x

Length of the resulting string pointed to by *s1*, not including the terminating null character.

No return value is reserved for error indication. However, the function can set `errno` to EINVAL – The string pointed to by *s2* contains characters outside the domain of the collating sequence.

Example

```
/* This program verifies that two transformed strings when      */
/* passed through strxfrm and then compared, provide the same   */
/* result as if passed through strcoll without any              */
/* transformation.

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

#define  BUFF_SIZE  256

main()
{
   char string1[BUFF_SIZE];
   char string2[BUFF_SIZE];
   int errno;
   int coll_result;
   int strcmp_result;
   size_t strxfrm_result1;
   size_t strxfrm_result2;

   /* setlocale to French locale */

   if (setlocale(LC_ALL, "fr_FR.ISO8859-1") == NULL) {
       perror("setlocale");
       exit(EXIT_FAILURE);
   }

   /* collate string 1 and string 2 and store the result */

   errno = 0;
   coll_result = strcoll("<a`>bcd", "abcz");
   if (errno) {
       perror("strcoll");
       exit(EXIT_FAILURE);
   }

   else {
       /* Transform the strings (using strxfrm) into string1   */
       /* and string2                                          */

       strxfrm_result1 = strxfrm(string1, "<a`>bcd", BUFF_SIZE);

       if (strxfrm_result1 == ((size_t) - 1)) {
```

```
            perror("strxfrm");
            exit(EXIT_FAILURE);
        }

        else if (strxfrm_result1 > BUFF_SIZE) {
            perror("\n** String is too long **\n");
            exit(EXIT_FAILURE);
        }

        else {
            strxfrm_result2 = strxfrm(string2, "abcz", BUFF_SIZE);
            if (strxfrm_result2 == ((size_t) - 1)) {
                perror("strxfrm");
                exit(EXIT_FAILURE);
            }

            else if (strxfrm_result2 > BUFF_SIZE) {
                perror("\n** String is too long **\n");
                exit(EXIT_FAILURE);
            }

            /* Compare the two transformed strings and verify   */
            /* that the result is the same as the result from   */
            /* strcoll on the original strings                  */
            else {
                strcmp_result = strcmp(string1, string2);
                if (strcmp_result == 0 && (coll_result == 0)) {
                    printf("\nReturn value from strcoll() and "
                      "return value from strcmp() are both zero.");
                   printf("\nThe program was successful\n\n");
                }

                else if ((strcmp_result < 0) && (coll_result < 0)) {
                    printf("\nReturn value from strcoll() and "
                 "return value from strcmp() are less than zero.");
                    printf("\nThe program successful\n\n");
                }

                else if ((strcmp_result > 0) && (coll_result > 0)) {
                    printf("\nReturn value from strcoll() and "
              "return value from strcmp() are greater than zero.");
                    printf("\nThe program was successful\n\n");
                }

                else {
                printf("** Error **\n");
                printf("\nReturn values are not of the same type");
                }
             }
         }
   }
}
```

Running the example program produces the following result:

```
Return value from strcoll() and return value
               from strcmp() are less than zero.
The program was successful
```

subwin

subwin — Creates a new subwindow with *numlines* lines and *numcols* columns starting at the coordinates (*begin_y*, *begin_x*) on the terminal screen.

Format

```
#include <curses.h>
WINDOW *subwin (WINDOW *win, int numlines, int numcols, int begin_y,
int begin_x);
```

Arguments

win

A pointer to the parent window.

numlines

The number of lines in the subwindow. If *numlines* is 0, then the function sets that dimension to LINES - *begin_y*. To get a subwindow of dimensions LINES by COLS, use the following format:

```
subwin (win, 0, 0, 0, 0)
```

numcols

The number of columns in the subwindow. If *numcols* is 0, then the function sets that dimension to COLS - *begin_x*. To get a subwindow of dimensions LINES by COLS, use the following format:

```
subwin (win, 0, 0, 0, 0)
```

begin_y

A window coordinate at which the subwindow is to be created.

begin_x

A window coordinate at which the subwindow is to be created.

Description

When creating the subwindow, *begin_y* and *begin_x* are relative to the entire terminal screen. If either *numlines* or *numcols* is 0, then the `subwin` function sets that dimension to (LINES - *begin_y*) or (COLS - *begin_x*), respectively.

The window pointed to by *win* must be large enough to contain the entire area of the subwindow. Any changes made to either window within the coordinates of the subwindow appear on both windows.

Return Values

window pointer

A pointer to an instance of the structure window corresponding to the newly created subwindow.

ERR

Indicates an error.

swab

swab — Swaps bytes.

Format

```
#include <unistd.h>
void swab (const void *src, void *dest, ssize_t nbytes);
```

Arguments

src

A pointer to the location of the string to copy.

dest

A pointer to where you want the results copied.

nbytes

The number of bytes to copy. Make this argument an even value. When it is an odd value, the `swab` function uses *nbytes* -1 instead.

Description

The `swab` function copies the number of bytes specified by *nbytes* from the location pointed to by *src*to the array pointed to by *dest*. The function then exchanges adjacent bytes. If a copy takes place between objects that overlap, the result is undefined.

swprintf

swprintf — Writes output to an array of wide characters under control of the wide-character format string.

Format

```
#include <wchar.h>
int swprintf (wchar_t *s, size_t n, const wchar_t *format, ...);
```

Arguments

s

A pointer to the resulting wide-character sequence.

n

The maximum number of wide characters that can be written to an array pointed to by *s*, including a terminating null wide character.

format

A pointer to a wide-character string containing the format specifications. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

. . .

Optional expressions whose resultant types correspond to conversion specifications given in the format specification.

If no conversion specifications are given, the output sources can be omitted. Otherwise, the function calls must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources.

Conversion specifications are matched to output sources in left-to-right order. Excess output pointers, if any, are ignored.

Description

The `swprintf` function is equivalent to the `fwprintf` function, except that the first argument specifies an array of wide characters instead of a stream.

No more than *n* wide characters are written, including a terminating null wide character, which is always added (unless *n* is 0).

See also `fwprintf`.

Return Values

x

The number of wide characters written, not counting the terminating null wide character.

Negative value

Indicates an error. Either *n* or more wide characters were requested to be written, or a conversion error occurred, in which case `errno` is set to EILSEQ.

swscanf

swscanf — Reads input from a wide-character string under control of the wide-character format string.

Format

```
#include <wchar.h>
int swscanf (const wchar_t *s, const wchar_t *format, ...);
```

Arguments

s

A pointer to a wide-character string from which the input is to be obtained.

format

A pointer to a wide-character string containing the format specifications. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

. . .

Optional expressions whose results correspond to conversion specifications given in the format specification.

If no conversion specifications are given, you can omit the input pointers. Otherwise, the function calls must have exactly as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input pointers.

Conversion specifications are matched to input sources in left-to-right order. Excess input pointers, if any, are ignored.

Description

The `swscanf` function is equivalent to the `fwscanf` function, except that the first argument specifies a wide-character string rather than a stream. Reaching the end of the wide-character string is the same as encountering EOF for the `fwscanf` function.

See also `fwscanf`.

Return Values

x

The number of input items assigned, sometimes fewer than provided for, or even 0 in the event of an early matching failure.

EOF

Indicates an error. An input failure occurred before any conversion.

symlink

symlink — Creates a symbolic link containing the specified contents.

Format

```
#include <unistd.h>
int symlink (const char *link_contents, const char *link_name);
```

Arguments

link_contents

Contents of the symbolic link file, specified as a text string representing the pathname to which the symbolic link will point.

link_name

The text string representing the name of the symbolic link file.

Description

A symbolic link is a special kind of file that points to another file. It is a directory entry that associates a filename with a text string that is interpreted as a POSIX pathname when accessed by certain services. A symbolic link is implemented on OpenVMS systems as a file of organization SPECIAL and type SYMBOLIC_LINK.

The `symlink` function creates a symbolic link (*link_name*) containing the specified contents (*link_contents*). No attempt is made at link creation time to interpret the symbolic link contents.

See also `readlink`, `unlink`, `realpath`, `lchown`, and `lstat`.

Return Values

0

Successful completion

-1

Indicates an error. `errno` is set to indicate the error:

- EACCES– Write permission is denied in the directory where the symbolic link is being created, or search permission is denied for a component of the path prefix of *link_name*.
- EEXIST– The *link_name* argument names an existing file or symbolic link.
- ENAMETOOLONG– The length of the *link_name* argument exceeds PATH_MAX, or a pathname component is longer than NAME_MAX, or the length of the *link_contents* argument is longer than SYMLINK_MAX.
- ENOSPC – The directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the file system containing the directory, or the new symbolic link cannot be created because no space is left on the file system that would contain the link, or the file system is out of file-allocation resources.
- Any `errno` value from `creat`, `fsync`, `lstat`, or `write`.

sysconf

sysconf — Gets configurable system variables.

Format

```
#include <unistd.h>
long int sysconf (int name);
```

Argument

name

Specifies the system variable to be queried.

Description

The `sysconf` function provides a method for determining the current value of a configurable system limit or whether optional features are supported.

You supply a symbolic constant in the *name* argument, and `sysconf` returns a value for the corresponding system variable:

- The symbolic constants defined in the `<unistd.h>` header file.
- The system variables are defined in the `<limits.h>` and `<unistd.h>` header files.

Table 42 lists the system variables returned by the `sysconf` function, and the symbolic constants that you can supply as the *name* value.

Table 42. sysconf Argument and Return Values

System Variable Returned	Symbolic Constant for *name*	Meaning
ISO POSIX-1		
ARG_MAX	_SC_ARG_MAX	The maximum length, in bytes, of the arguments for one of the `exec` functions, including environment data.
CHILD_MAX	_SC_CHILD_MAX	The maximum number of simultaneous processes for each real user ID.
CLK_TCK	_SC_CLK_TCK	The number of clock ticks per second. The value of CLK_TCK can be variable. Do not assume that CLK_TCK is a compile-time constant.
NGROUPS_MAX	_SC_NGROUPS_MAX	The maximum number of simultaneous supplementary group IDs for each process.
OPEN_MAX	_SC_OPEN_MAX	The maximum number of files that one process can have open at one time.
STREAM_MAX	_SC_STREAM_MAX	The number of streams that one process can have open at one time.
TZNAME_MAX	_SC_TZNAME_MAX	The maximum number of bytes supported for the name of a time zone (not the length of the `TZ` environmental variable).
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL	This variable has a value of 1 if the system supports job control; otherwise, -1 is returned.
_POSIX_SAVED_IDS	_SC_SAVED_IDS	This variable has a value of 1 if each process has a saved set user ID and a saved set group ID; otherwise, -1 is returned.
_POSIX_VERSION	_SC_VERSION	The date of approval of the most current version of the POSIX-1 standard that the system supports. The date is a 6-digit number, with the first 4 digits signifying the year and the last 2 digits the month. If _POSIX_VERSION is not defined, -1 is returned. Different versions of the POSIX-1 standard are periodically approved by the IEEE Standards Board, and the date of approval is used to distinguish between different versions.
ISO POSIX-2		

System Variable Returned	Symbolic Constant for *name*	Meaning
BC_BASE_MAX	_SC_BC_BASE_MAX	The maximum value allowed for the `obase` variable with the `bc` command.
BC_DIM_MAX	_SC_BC_DIM_MAX	The maximum number of elements permitted in an array by the `bc` command.
BC_SCALE_MAX	_SC_BC_SCALE_MAX	The maximum value allowed for the scale variable with the `bc` command.
BC_STRING_MAX	_SC_BC_STRING_MAX	The maximum length of string constants accepted by the `bc` command.
COLL_WEIGHTS_MAX	_SC_COLL_WEIGHTS_ MAX	The maximum number of weights that can be assigned to an entry in the LC_COLLATE locale-dependent information in a locale definition file.
EXPR_NEST_MAX	_SC_EXPR_NEST_MAX	The maximum number of expressions that you can nest within parentheses by the `expr` command.
LINE_MAX	_SC_LINE_MAX	The maximum length, in bytes, of a command input line (either standard input or another file) when the utility is described as processing text files. The length includes room for the trailing new-line character.
RE_DUP_MAX	_SC_RE_DUP_MAX	The maximum number of repeated occurrences of a regular expression permitted when using the interval notation arguments, such as the *m* and *n* arguments with the `ed` command.
_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM	This variable has a value of 1 if the system supports at least one terminal type; otherwise, -1 is returned.
_POSIX2_C_BIND	_SC_2_C_BIND	This variable has a value of 1 if the system supports the C language binding option; otherwise, -1 is returned.
_POSIX2_C_DEV	_SC_2_C_DEV	This variable has a value of 1 if the system supports the optional C Language Development Utilities from the ISO POSIX-2 standard; otherwise, -_POSIX_THREAD_ATTR_STAC_ KSSCI_ZT1 is returned.
_POSIX2_C_VERSION	_SC_2_C_VERSION	Integer value indicating the version of the ISO POSIX-2 standard (C language binding). It changes with each new version of the ISO POSIX-2 standard.
_POSIX2_VERSION	_SC_2_VERSION	Integer value indicating the version of the ISO POSIX-2 standard (Commands). It changes with each new version of the ISO POSIX-2 standard.

System Variable Returned	Symbolic Constant for *name*	Meaning
_POSIX2_FORT_DEV	_SC_2_FORT_DEV	The variable has a value of 1 if the system supports the Fortran Development Utilities Option from the ISO POSIX-2 standard; otherwise, -1 is returned.
_POSIX2_FORT_RUN	_SC_2_FORT_RUN	The variable has a value of 1 if the system supports the Fortran Runtime Utilities Option from the ISO POSIX-2 standard; otherwise, -1 is returned.
_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF	The variable has a value of 1 if the system supports the creation of new locales with the `localedef` command; otherwise, -1 is returned.
_POSIX2_SW_DEV	_SC_2_SW_DEV	The variable has a value of 1 if the system supports the Software Development Utilities Option from the ISO POSIX-2 standard; otherwise, -1 is returned.
_POSIX2_UPE	_SC_2_UPE	The variable has a value of 1 if the system supports the User Portability Utilities Option; otherwise, -1 is returned.
POSIX 1003.1c-1995		
_POSIX_THREADS	_SC_THREADS	This variable has a value of 1 if the system supports POSIX threads; otherwise, -1 is returned.
_POSIX_THREAD_ATTR _STACKSIZE	_SC_THREAD_ATTR _STACKSIZE	This variable has a value of 1 if the system supports the POSIX threads stack size attribute; otherwise, -1 is returned.
_POSIX_THREAD_ PRIORITY _SCHEDULING	_SC_THREAD_ PRIORITY _SCHEDULING	The 1003.1c implementation supports the real time scheduling functions.
_POSIX_THREAD_SAFE_ FUNCTIONS	_SC_THREAD_SAFE _FUNCTIONS	TRUE if the implementation supports the thread-safe ANSI C functions in POSIX 1003.1c.
PTHREAD_DESTRUCTOR _ITERATIONS	_SC_THREAD_ DESTRUCTOR_ ITERATIONS	When a thread terminates, DECthreads iterates through all non-NULL thread-specific data values in the thread, and calls a registered destructor routine (if any) for each. It is possible for a destructor routine to create new values for one or more thread-specific data keys. In that case, DECthreads goes through the entire process again.

System Variable Returned	Symbolic Constant for *name*	Meaning
		_SC_THREAD_DESTRUCTOR_ ITERATIONS is the maximum number of times the implementation loops before it terminates the thread even if there are still non-NULL values.
PTHREAD_KEYS_MAX	_SC_THREAD_KEYS_ MAX	The maximum number of thread-specific data keys that an application can create.
PTHREAD_STACK_MIN	_SC_THREAD_STACK_ MIN	The minimum allowed size of a stack for a new thread. Any lower value specified for the "stacksize" thread attribute is rounded up.
UINT_MAX	_SC_THREAD_THREADS _MAX	The maximum number of threads an application is allowed to create. Since DECthreads does not enforce any fixed limit, this value is -1.
X/Open		
_XOPEN_VERSION	_SC_XOPEN_VERSION	An integer indicating the most current version of the X/Open standard that the system supports.
PASS_MAX	_SC_PASS_MAX	Maximum number of significant bytes in a password (not including terminating null).
XOPEN_CRYPT	_SC_XOPEN_CRYPT	This variable has a value of 1 if the system supports the X/Open Encryption Feature Group; otherwise, -1 is returned.
XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N	This variable has a value of 1 if the system supports the X/Open enhanced Internationalization Feature Group; otherwise, -1 is returned.
XOPEN_SHM	_SC_XOPEN_SHM	This variable has a value of 1 if the system supports the X/Open Shared Memory Feature Group; otherwise, -1 is returned.
X/Open Extended		
ATEXIT_MAX	_SC_ATEXIT_MAX	The maximum number of functions that you can register with `atexit` per process.
PAGESIZE	_SC_PAGESIZE	Size, in bytes, of a page.
PAGE_SIZE	_SC_PAGE_SIZE	Same as PAGESIZE. If either PAGESIZE or PAGE_SIZE is defined, the other is defined with the same value.
IOV_MAX	_SC_IOV_MAX	Maximum number of `iovec` structures that one process has available for use with `readv` or `writev`.

System Variable Returned	Symbolic Constant for *name*	Meaning
XOPEN_UNIX	_SC_XOPEN_UNIX	This variable has a value of 1 if the system supports the X/Open CAE Specification, August 1994, System Interfaces and Headers, Issue 4, Version 2,(ISBN: 1-85912-037-7, C435); otherwise, -1 is returned.
Other		
N/A	_SC_CPU_CHIP_TYPE	Returns information for the processor type. See the description after this table.

For the _SC_CPU_CHIP_TYPE symbolic constant:

- On Alpha servers, `sysconf` returns the architecture type (2), as given by the $GETSYI system service.
- Integrity processor information is stored in CPUID register 3. This register contains a 64-bit integer divided into 1-byte fields indicating version information related to the processor implementation. The `sysconf` function returns the low-order longword with the following information:

```
31     24 23    16 15    8 7     0
---------------------------------
| family | model  | rev   |number|
---------------------------------
```

These fields are described in the following table:

Field	Bits	Description
number	7:0	Index of the largest implemented CPUID register (one less than the number of implemented CPUID registers). This value will be at least 4.
rev	15:8	Processor revision number. An 8-bit value that represents the revision or stepping of this processor implementation within the processor model.
model	23:16	Processor model number. A unique 8-bit value representing the processor model within the processor family.
family	31:24	Processor family number. A unique 8-bit value representing the processor family.

Return Values

x

The current variable value on the system. The value does not change during the lifetime of the calling process.

-1

Indicates an error.

If the value of the *name* argument is invalid, `errno` is set to indicate the error.

If the value of the *name* argument is undefined, `errno` is unchanged.

system

system — Passes a given string to the host environment to be executed by a command processor. This function is non-reentrant.

Format

```
#include <stdlib.h>
int system (const char *string);
```

Argument

string

A pointer to the string to be executed. If *string* is NULL, a nonzero value is returned. The string is a DCL command, not the name of an image. To execute an image, use one of the `exec` routines.

Description

The `system` function spawns a subprocess and executes the command specified by *string* in that subprocess. The `system` function waits for the subprocess to complete before returning the subprocess status as the return value of the function.

The subprocess is spawned within the `system` call by a call to `vfork`. Because of this, a call to `system` should not be made after a call to `vfork` and before the corresponding call to an `exec` function.

For OpenVMS Version 7.0 and higher systems, if you include `<stdlib.h>` and compile with the _POSIX_EXIT feature-test macro set, then the `system` function returns the status as if it called `waitpid` to wait for the child. Therefore, use the WIFEXITED and WEXITSTATUS macros (described in the `wait*` routines) to retrieve the exit status in the range of 0 to 255.

You set the _POSIX_EXIT feature-test macro by using /DEFINE=_POSIX_EXIT or #define _POSIX_EXIT at the top of your file, before any file inclusions.

Return Value

nonzero value

If *string* is NULL, a value of 1 is returned, indicating that the `system` function is supported. If *string* is not NULL, the value is the subprocess OpenVMS return status.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>     /* write, close */
#include <fcntl.h>      /* Creat */

main()
{
    int status,
        fd;
```

```
    /* Creat a file we are sure is there        */

    fd = creat("system.test", 0);
    write(fd, "this is an example of using system", 34);
    close(fd);

    if (system(NULL)) {
        status = system("DIR/NOHEAD/NOTRAIL/SIZE SYSTEM.TEST");
        printf("system status = %d\n", status);
    }
    else
        printf("system() not supported.\n");
}
```

Running this example program produces the following result:

```
DISK3$:[JONES.CRTL.2059.SRC]SYSTEM.TEST;1
                            1
system status = 1
```

tan

tan — Returns a `double` value that is the tangent of its radian argument.

Format

```
#include <math.h>
double tan (double x);
float tanf (float x);
long double tanl (long double x);
double tand (double x);
float tandf (float x);
long double tandl (long double x);
```

Argument

x

A radian expressed as a real number.

Description

The `tan` functions compute the tangent of *x*, measured in radians.

The `tand` functions compute the tangent of *x*, measured in degrees.

Return Values

x

The tangent of the argument.

HUGE_VAL

x is a singular point (. . . -3 π/2, - π/2, π/2 . . .).

NaN

x is NaN; `errno` is set to EDOM.

0

x is ±Infinity; `errno` is set to EDOM.

±HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

0

Underflow occurred; `errno` is set to ERANGE.

tanh

tanh — Returns the hyperbolic tangent of its argument.

Format

```
#include <math.h>
double tanh (double x);
float tanhf (float x);
long double tanhl (long double x);
```

Argument

x

A real number.

Description

The `tanh` functions return the hyperbolic tangent their argument, calculated as (e** *x* - e**(- *x*))/(e**x + e**(- *x*)).

Return Values

n

The hyperbolic tangent of the argument.

HUGE_VAL

The argument is too large; `errno` is set to ERANGE.

NaN

x is NaN; `errno` is set to EDOM.

0

Underflow occurred; `errno` is set to ERANGE.

telldir

telldir — Returns the current location associated with a specified directory stream. Performs operations on directories.

Format

```
#include <dirent.h>
long int telldir (DIR *dir_pointer);
```

Argument

dir_pointer

A pointer to the `DIR` structure of an open directory.

Description

The `telldir` function returns the current location associated with the specified directory stream.

Return Values

x

The current location.

-1

Indicates an error and is further specified in the global `errno`.

tempnam

tempnam — Constructs the name for a temporary file.

Format

```
#include <stdio.h>
char *tempnam (const char *directory, const char *prefix, ...;)
```

Arguments

directory

A pointer to the pathname of the directory where you want to create a file.

prefix

A pointer to an initial character sequence of the filename. The *prefix* argument can be null, or it can point to a string of up to five characters used as the first characters of the temporary filename.

. . .

An optional argument that can be either 1 or 0. If you specify 1, `tempnam` returns the file specification in OpenVMS format. If you specify 0, `tempnam` returns the file specification in UNIX style format. For more information about UNIX style directory specifications, see Section 1.3.3.

Description

The `tempnam` function generates filenames for temporary files. It allows you to control the choice of a directory.

If the *directory* argument is null or points to a string that is not a pathname for an appropriate directory, the pathname defined as `P_tmpdir` in the `<stdio.h>` header file is used. For programs running under a detached process, the *directory* argument cannot be null.

You can bypass the selection of a pathname by providing the `TMPDIR` environment variable in the user environment. The value of the `TMPDIR` variable is a pathname for the desired temporary file directory.

Use the *prefix* argument to specify a prefix of up to five characters for the temporary filename.

The `tempnam` function returns a pointer to the generated pathname, suitable for use in a subsequent call to the `free` function.

See also `free`.

Note

In contrast to `tmpnam`, `tempnam` does not have to generate a different filename on each call. `tempnam` generates a new filename only if the file with the specified name exists. If you need a unique filename on each call, use `tmpnam` instead of `tempnam`.

Return Values

x

A pointer to the generated pathname, suitable for use in a subsequent call to the `free` function.

NULL

An error occurred; `errno` is set to indicate the error.

tgamma

tgamma — Returns the gamma function of its argument.

Format

```
#include <math.h>
double tgamma (double x);
float tgammaf (float x);
long double tgammal (long double x);
```

Argument

x

A real value.

Description

The `tgamma` functions compute the gamma function of *x*.

Return Values

n

Upon success, the gamma function of *x*.

-1

If *x* is negative. `errno` is set to EDOM.

±HUGE_VAL

Overflow occurred or *x* is ±0. `errno` is set to ERANGE.

NaN

If *x* is NaN or -Inf. `errno` is set to EDOM.

x

If *x* is +Inf.

time

time — Returns the time (expressed as Universal Coordinated Time) elapsed since 00:00:00, January 1, 1970, in seconds.

Format

```
#include <time.h>
time_t time (time_t *time_location);
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `time` function that is equivalent to the behavior before OpenVMS Version 7.0.

Argument

time_location

Either NULL or a pointer to the place where the returned time is also stored. The `time_t` type is defined in the `<time.h>` header file as follows:

```
typedef unsigned long int time_t;
```

Return Values

x

The time elapsed past the Epoch.

(`time_t`)(-1)

Indicates an error. If the value of SYS$TIMEZONE_DIFFERENTIAL logical is wrong, the function will fail with `errno` set to EINVAL.

times

times — Passes back the accumulated times of the current process and its terminated child processes.

Format

```
#include <times.h>
clock_t times (struct tms *buffer); (OpenVMS V7.0 and higher)
void times (tbuffer_t *buffer); (pre OpenVMS V7.0)
```

Argument

buffer

A pointer to the terminal buffer.

Description

For both process and children times, the structure breaks down the time by user and system time. Since the OpenVMS system does not differentiate between system and user time, all system times are returned as 0. Accumulated CPU times are returned in 10-millisecond units.

Only the accumulated times for child processes running a C main program or a program that calls `VAXC$CRTL_INIT` or `DECC$CRTL_INIT` are included.

On OpenVMS Version 7.0 and higher systems, the `times` function returns the elapsed real time in clock ticks since an arbitrary reference time in the past (for example, system startup time). This reference time does not change from one `times` function call to another. The return value can overflow the possible range of type `clock_t` values. When `times` fails, it returns a value of -1. The C RTL uses system-boot time as its reference time.

Return Values

x

The elapsed real time in clock ticks since system-boot time.

(`clock_t`)(-1)

Indicates an error.

tmpfile

tmpfile — Creates a temporary file that is opened for update.

Format

```
#include <stdio.h>
FILE *tmpfile (void);
```

Description

The file exists only for the duration of the process, or until the file is closed and is preserved across calls to `vfork`.

Return Values

x

The address of a file pointer (defined in the `<stdio.h>` header file).

NULL

Indicates an error.

tmpnam

tmpnam — Generates filenames that can be safely used for a temporary file.

Format

```
#include <stdio.h>
char *tmpnam (char *name);
```

Function Variants

The `tmpnam` function has variants named `_tmpnam32` and `_tmpnam64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argument

name

A character string containing a name to use in place of file-name arguments in functions or macros. Successive calls to `tmpnam` with a null argument cause the function to overwrite the current name.

Return Value

x

If the *name* argument is the NULL pointer value NULL, `tmpnam` returns the address of an internal storage area. If *name* is not NULL, then it is considered the address of an area of length `L_tmpnam` (defined in the `<stdio.h>` header file). In this case, `tmpnam` returns the *name* argument as the result.

toascii

toascii — Converts its argument, an 8-bit ASCII character, to a 7-bit ASCII character.

Format

```
#include <ctype.h>
int toascii (char character);
```

Argument

character

An object of type `char`.

Return Value

x

A 7-bit ASCII character.

tolower

tolower — Converts a character to lowercase.

Format

```
#include <ctype.h>
int tolower (int character);
```

Argument

character

An object of type `int` representable as an `unsigned char` or the value of EOF. For any other value, the behavior is undefined.

Description

If the argument represents an uppercase letter, and there is a corresponding lowercase letter, as defined by character type information in the program locale category LC_CTYPE, the corresponding lowercase letter is returned.

If the argument is not an uppercase character, it is returned unchanged.

Return Value

x

The lowercase letter corresponding to the argument. Or, the unchanged argument, if it is not an uppercase character.

_tolower

_tolower — Converts an uppercase character to lowercase.

Format

```
#include <ctype.h>
int _tolower (int character);
```

Argument

character

This argument must be an uppercase letter.

Description

The `_tolower` macro is equivalent to the `tolower` function except that its argument *must* be an uppercase letter (not lowercase, not EOF).

As of OpenVMS Version 8.3 and to comply with the C99 ANSI standard and X/Open Specification, the `_tolower` macro by default does not evaluate its parameter more than once. It simply calls the `tolower` function. This avoids side effects (such as i++ or function calls) where the user can tell how many times an expression is evaluated.

To keep the older, optimized `_tolower` macro behavior, compile with /DEFINE=_FAST_TOUPPER. Then, as in previous releases, `_tolower` optimizes the call to avoid the overhead of a runtime call. The parameters are checked to determine how to calculate the result, thereby creating unwanted side effects. Therefore, when compiling with /DEFINE=_FAST_TOUPPER, do not use the `_tolower` macro with arguments that contain side-effect operations. For instance, the following example will not return the expected result:

```
d = _tolower (C++);
```

Return Value

x

The lowercase letter corresponding to the argument.

touchwin

touchwin — Places the most recently edited version of the specified window on the terminal screen.

Format

```
#include <curses.h>
int touchwin (WINDOW *win);
```

Argument

win

A pointer to the window.

Description

The `touchwin` function is normally used only to refresh overlapping windows.

Return Values

OK

Indicates success.

ERR

Indicates an error.

toupper

toupper — Converts a character to uppercase.

Format

```
#include <ctype.h>
int toupper (int character);
```

Argument

character

An object of type `int` representable as an `unsigned char` or the value of EOF. For any other value, the behavior is undefined.

Description

If the argument represents a lowercase letter, and there is a corresponding uppercase letter, as defined by character type information in the program locale category LC_CTYPE, the corresponding uppercase letter is returned.

If the argument is not a lowercase character, it is returned unchanged.

Return Value

x

The uppercase letter corresponding to the argument. Or, the unchanged argument, if the argument is not a lowercase character.

_toupper

_toupper — Converts a lowercase character to uppercase.

Format

```
#include <ctype.h>
int _toupper (int character);
```

Argument

character

This argument must be a lowercase letter.

Description

The `_toupper` macro is equivalent to the `toupper` function except that its argument *must* be a lowercase letter (not uppercase, not EOF).

As of OpenVMS Version 8.3 and to comply with the C99 ANSI standard and X/Open Specification, the `_toupper` macro by default does not evaluate parameters more than once. It simply calls the `toupper` function. This avoids side effects (such as i++ or function calls) where the user can tell how many times an expression is evaluated.

To keep the older, optimized `_toupper` macro behavior, compile with /DEFINE=_FAST_TOUPPER. Then, as in previous releases, `_toupper` optimizes the call to avoid the overhead of a runtime call. The parameters are checked to determine how to calculate the result, thereby creating unwanted side effects. So when compiling with /DEFINE=_FAST_TOUPPER, do not use the `_toupper` macro with arguments that contain side-effect operations. For instance, the following example will not return the expected result:

```
d = _toupper (c++);
```

Return Value

x

The uppercase letter corresponding to the argument.

towctrans

towctrans — Maps one wide character to another according to a specified mapping descriptor.

Format

```
#include <wctype.h>
wint_t towctrans (wint_t wc, wctrans_t desc);
```

Arguments

wc

The wide character that you want to map.

desc

Description of the mapping obtained through a call to the `wctrans` function.

Description

The `towctrans` function maps the wide character specified in *wc*, using the mapping described by `desc`.

The current setting of the `LC_CTYPE` category must be the same as during the call to the `wctrans` function that returned the value of *desc*.

Return Value

x

The mapped value of the *wc* wide character, if this character exists in the mapping described by *desc*. Otherwise, the value of *wc* is returned.

towlower

towlower — Converts the argument, a wide-character code, to lowercase. If the argument is not an uppercase character, it is returned unchanged.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int towlower (wint_t wc);
```

Argument

wc

An object of type `wint_t` representable as a valid wide character in the current locale, or the value of WEOF. For any other value, the behavior is undefined.

Description

If the argument is an uppercase wide character, the corresponding lowercase wide character (as defined in the LC_CTYPE category of the locale) is returned, if it exists. If it does not exist, the function returns the input argument unchanged.

towupper

towupper — Converts the argument, a wide character, to uppercase. If the argument is not a lowercase character, it is returned unchanged.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
int towupper (wint_t wc);
```

Argument

wc

An object of type `wint_t` representable as a valid wide character in the current locale, or the value of WEOF. For any other value, the behavior is undefined.

Description

If the argument is a lowercase wide character, the corresponding uppercase wide character (as defined in the LC_CTYPE category of the locale) is returned, if it exists. If it does not exist, the function returns the input argument unchanged.

trunc

trunc — Truncates the argument to an integral value.

Format

```
#include <math.h>
double trunc (double x);
float truncf (float x,);
long double truncl (long double x);
```

Argument

x

A floating-point number.

Return Value

n

The truncated, integral value of the argument.

truncate

truncate — Changes file length to a specified length, in bytes.

Format

```
#include <unistd.h>
int truncate (const char *path, off_t length);
```

Arguments

path

The name of a file that is to be truncated. This argument must point to a pathname that names a regular file for which the calling process has write permission.

length

The new length of the file, in bytes. The `off_t` type of *length* is either a 64-bit or 32-bit integer. The 64-bit interface allows for file sizes greater than 2 GB, and can be selected at compile time by defining the _LARGEFILE feature-test macro as follows:

```
CC/DEFINE=_LARGEFILE
```

Description

The `truncate` function changes the length of a file to the size, in bytes, specified by the *length* argument.

If the new length is less than the previous length, the function removes all data beyond *length* bytes from the specified file. All file data between the new End-of-File and the previous End-of-File is discarded.

For stream files, if the new length is greater than the previous length, new file data between the previous End-of-File and the new End-of-File is added, consisting of all zeros. For record files, it is not possible to extend the file in this manner.

Return Values

0

Indicates success.

-1

An error occurred; `errno` is set to indicate the error.

ttyname, ttyname_r

ttyname, ttyname_r — Find the pathname of a terminal.

Format

```
#include <unixio.h> (Compatibility)
char *ttyname (void); (Compatibility)
#include <unistd.h> (OpenVMS V7.3-2 and higher)
char *ttyname (int filedes); (OpenVMS V7.3-2 and higher)
int ttyname_r (int filedes, char name, size_t namesize);
(OpenVMS V7.3-2 and higher)
```

Arguments

filedes

An open file descriptor.

name

Pointer to a buffer in which the terminal name is stored.

namesize

The length of the buffer pointed to by the *name* argument.

Description

The implementation of the `ttyname` function that takes no argument is provided only for backward compatibility. This legacy implementation returns a pointer to the null-terminated name of the terminal device associated with file descriptor 0, the default input device (`stdin`). A value of 0 is returned if SYS$INPUT is not a TTY device.

The `ttyname_r` function and the implementation of `ttyname` that takes a *filedes* argument are UNIX standard compliant and are available with only OpenVMS Version 7.3-2 and higher.

The standard compliant `ttyname` function returns a pointer to a string containing a null-terminated pathname of the terminal associated with file descriptor *filedes*. The return value might point to static data whose content is overwritten by each call. The `ttyname` interface need not be reentrant.

The `ttyname_r` function returns a pointer to store the null-terminated pathname of the terminal associated with the file descriptor *filedes* in the character array referenced by *name*. The array is *namesize* characters long and should have space for the name and the terminating null character. The maximum length of the terminal name is TTY_NAME_MAX.

If successful, `ttyname` returns a pointer to a string. Otherwise, a NULL pointer is returned and `errno` is set to indicate the error.

If successful, `ttyname_r` stores the terminal name as a null-terminated string in the buffer pointed to by *name* and returns 0. Otherwise, an error number is returned to indicate the error.

Return Values

x

Upon successful completion, `ttyname` returns a pointer to a null-terminated string.

NULL

Upon failure, `ttyname` returns a NULL pointer and sets `errno` to indicate the failure:

- EBADF – The *fildes* argument is not a valid file descriptor.
- ENOTTY– The *fildes* argument does not refer to a terminal device.

0

Upon successful completion, `ttyname_r` returns 0.

n

Upon failure, `ttyname_r` sets `errno` to indicate the failure, and returns the same `errno` code:

- EBADF – The *fildes* argument is not a valid file descriptor.
- ENOTTY– The *fildes* argument does not refer to a TTY device.
- ERANGE– The value of *namesize* is smaller than the length of the string to be returned including the terminating null character.

0

For the legacy `ttyname`, indicates that SYS$INPUT is not a TTY device.

tzset

tzset — Sets and accesses time-zone conversion.

Format

```
#include <time.h>
void tzset (void);
extern char *tzname[];
extern long int timezone;
extern int daylight;
```

Description

The `tzset` function initializes time-conversion information used by the `ctime`, `localtime`, `mktime`, `strftime`, and `wcsftime` functions.

The `tzset` function sets the following external variables:

- `tzname` is set as follows, where "std" is a 3-byte name for the standard time zone, and "dst" is a 3-byte name for the Daylight Savings Time zone:

  ```
  tzname[0] = "std" tzname[1] = "dst"
  ```

- `daylight` is set to 0 if Daylight Savings Time should never be applied to the time zone. Otherwise, `daylight` is set to 1.

- `timezone` is set to the difference between UTC and local standard time.

The environment variable `TZ` specifies how `tzset` initializes time conversion information:

- If `TZ` is absent from the environment, the implementation-dependent time-zone information is used, as follows:

  The best available approximation to local wall-clock time is used, as defined by the SYS$LOCALTIME system logical, which points to a `tzfile` format file that describes default time-zone rules.
  This system logical is set during the installation of OpenVMS Version 7.0 or higher to define a time-zone file based off the root directory SYS$COMMON:[SYS$ZONEINFO.SYSTEM].[1]

- If `TZ` appears in the environment but its value is a null string, Coordinated Universal Time (UTC) is used (without leap-second correction).

- If `TZ` appears in the environment and its value is not a null string, the value has one of three formats, as described in Table 43.

[1]The C RTL uses a public-domain, time-zone handling package that puts time-zone conversion rules in easily accessible and modifiable files. These files reside in the SYS$COMMON:[SYS$ZONEINFO.SYSTEM.SOURCES] directory.

The time-zone compiler zic converts these files to a special format described by the <tzfile.h> header file. The converted files are created with a root directory of SYS$COMMON:[SYS$ZONEINFO.SYSTEM], which is pointed to by the SYS$TZDIR system logical. This format is readable by the C library functions that handle time-zone information. For example, in the eastern United Stated, SYS$LOCALTIME is defined to be SYS$COMMON:[SYS$ZONEINFO.SYSTEM.US]EASTERN.)

Table 43. Time-Zone Initialization Rules

TZ Format	Meaning
:	UTC is used.
: *pathname*	The characters following the colon specify the pathname of a `tzfile` format file from which to read the time-conversion information. A pathname beginning with a slash (/) represents an absolute pathname; otherwise, the pathname is relative to the system time-conversion information directory specified by SYS$TZDIR, which by default is SYS$COMMON: [SYS$ZONEINFO.SYSTEM].
stdoffset[*dst*[*offset*] [,rule]]	The value is first used as the pathname of a file (as described for the :*pathname* format) from which to read the time-conversion information. If that file cannot be read, the value is then interpreted as a direct specification of the time-conversion information, as follows:
	std and *dst* – Three or more characters that are the designation for the time zone: • *std* – Standard time zone. Required. • *dst* – Daylight Savings Time zone. Optional. If *dst* is omitted, Daylight Savings Time does not apply. Uppercase and lowercase letters are explicitly allowed. Any characters are allowed, except the following: • digits • leading colon (:) • comma (,) • minus (-) • plus (+) • ASCII null character
	offset – The value added to the local time to arrive at UTC. The offset has the following format: `hh[:mm[:ss]]` In this format: • *hh* (hours) is a one-or two-digit value of 0–24. • *mm* (minutes) is a value of 0–59. (optional) • *ss* (seconds) is a value of 0–59. (optional)
	The offset following *std* is required. If no offset follows *dst*, summer time is assumed, one hour ahead of standard time. You

TZ Format	Meaning
	can use one or more digits; the value is always interpreted as a decimal number. If the time zone is preceded by a minus sign (-), the time zone is East of Greenwich; otherwise, it is West, which can also be indicated by a preceding plus sign (+).
	rule – Indicates when to change to and return from summer time. The rule has the form: `start[/time], end[/time]` where: • *start* is the date when the change from standard time to summer time occurs. • *end* is the date for returning from summer time to standard time.
	If *start* and *end* are omitted, the default is the US Daylight Savings Time start and end dates. The format for *start* and *end* must be one of the following: • *Jn* – The Julian day *n* ($1 < n < 365$). Leap days are not counted. That is, in all years, including leap years, February 28 is day 59 and March 1 is day 60. You cannot explicitly refer to February 29. • *n* – The zero based Julian day ($0 < n < 365$). Leap days are counted, making it possible to refer to February 29. • *Mm.n.d* – The *n*th *d* day of month *m*, where: $0 < n < 5$ $0 < d < 6$ $1 < m < 12$ When *n* is 5, it refers to the last *d* day of month *m*. Sunday is day 0.
	time – The time when, in current time, the change to or return from summer time occurs. The *time* argument has the same format as *offset*, except that you cannot use a leading minus (-) or plus (+) sign. If *time* is not specified, the default is 02:00:00. If no rule is present in the `TZ` specification, the rules used are those specified by the `tzfile` format file defined by the SYS$POSIXRULES system logical in the system time-conversion information directory, with the standard and summer time offsets from UTC replaced by those specified by the offset values in `TZ`. If `TZ` does not specify a `tzfile` format file and cannot be interpreted as a direct specification, UTC is used.

Note

The UTC-based time functions, introduced in OpenVMS Version 7.0, had degraded performance compared with the non-UTC-based time functions.

OpenVMS Version 7.1 added a cache for time-zone files to improve performance. The size of the cache is determined by the logical name DECC$TZ_CACHE_SIZE. To accommodate most countries changing the time twice per year, the default cache size is large enough to hold two time-zone files.

See also `ctime`, `localtime`, `mktime`, `strftime`, and `wcsftime`.

Sample TZ Specification

```
EST5EDT4,M4.1.0,M10.5.0
```

This sample TZ specification describes the rule defined in 1987 for the Eastern time zone in the US:

- EST (Eastern Standard Time) is the designation for standard time, which is 5 hours behind UTC.
- EDT (Eastern Daylight Time) is the designation for summer time, which is 4 hours behind UTC. EDT starts on the first Sunday in April and ends on the last Sunday in October.

Because *time* was not specified in either case, the changes occur at the default time, which is 2:00 A.M. The start and end dates did not need to be specified, because they are the defaults.

ualarm

ualarm — Sets or changes the timeout of interval timers.

Format

```
#include <unistd.h>
useconds_t ualarm (useconds_t mseconds, useconds_t interval);
```

Arguments

mseconds

Specifies a number of real-time microseconds.

interval

Specifies the interval for repeating the timer.

Description

The `ualarm` function causes the `SIGALRM` signal to be generated for the calling process after the number of real-time microseconds specified by *useconds* has elapsed. When the *interval* argument is nonzero, repeated timeout notification occurs with a period in microseconds specified by *interval*. If the notification signal `SIGALRM` is not intercepted or is ignored, the calling process is terminated.

If you call a combination of `ualarm` and `setitimer` functions, and the AST status is disabled, the return value is invalid.

If you call a combination of `ualarm` and `setitimer` functions, and the AST status is enabled, the return value is valid.

This is because you cannot invoke an AST handler to clear the previous value of the timer when ASTs are disabled or invoked from a handler that was invoked at AST level.

Note

Interactions between `ualarm` and either `alarm`, or `sleep` are unspecified.

See also `setitimer`.

Return Values

n

The number of microseconds remaining from the previous `ualarm` or `setitimer` call.

0

No timeouts are pending or `ualarm` not previously called.

-1

Indicates an error.

umask

umask — Creates a file protection mask that is used when a new file is created, and returns the previous mask value.

Format

```
#include <stat.h>
mode_t umask (mode_t mode_complement);
```

Argument

mode_complement

Shows which bits to turn off when a new file is created. See the description of `chmod` to determine what the bits represent.

Description

Initially, the file protection mask is set from the current process's default file protection. This is done when the C main program starts up or when `DECC$CRTL_INIT` (or `VAXC$CRTL_INIT`) is called. You can change this for all files created by your program by calling `umask` or you can use `chmod` to change the file protection on individual files. The file protection of a file created by `open` or `creat` is the bitwise AND of the `open` and `creat` mode argument with the complement of the value passed to `umask` on the previous call.

Note

The way to create files with OpenVMS RMS default protections using the UNIX system-call functions `umask`, `mkdir`, `creat`, and `open` is to call `mkdir`, `creat`, and `open` with a file-protection mode argument of 0777 in a program that never specifically calls `umask`. These default protections include correctly establishing protections based on ACLs, previous versions of files, and so on.

In programs that do `vfork`/`exec` calls, the new process image inherits whether `umask` has ever been called or not from the calling process image. The `umask` setting and whether the `umask` function has ever been called are both inherited attributes.

Return Value

x

The old mask value.

uname

uname — Gets system identification information.

Format

```
#include <utsname.h>
int uname (struct utsname *name);
```

Argument

name

The current system identifier.

Description

The `uname` function stores null-terminated strings of information identifying the current system into the structure referenced by the *name* argument.

The `utsname` structure is defined in the `<utsname.h>` header file and contains the following members:

`sysname`	Name of the operating system implementation
`nodename`	Network name of this machine
`release`	Release level of the operating system
`version`	Version level of the operating system
`machine`	Machine hardware platform

Return Values

0

Indicates success.

-1

Indicates an error; `errno` or `vaxc$errno` is set as appropriate.

ungetc

ungetc — Pushes a character back into the input stream and leaves the stream positioned before the character.

Format

```
#include <stdio.h>
int ungetc (int character, FILE *file_ptr);
```

Arguments

character

A value of type `int`.

file_ptr

A file pointer.

Description

When using the `ungetc` function, the character is pushed back onto the file indicated by *file_ptr*.

One push-back is guaranteed, even if there has been no previous activity on the file. The `fseek` function erases all memory of pushed-back characters. The pushed-back character is not written to the underlying file. If the character to be pushed back is EOF, the operation fails, the input stream is left unchanged, and EOF is returned.

See also `fseek` and `getc`.

Return Values

x

The push-back character.

EOF

Indicates it cannot push the character back.

ungetwc

ungetwc — Pushes a wide character back into the input stream.

Format

```
#include <wchar.h>
wint_t ungetwc (wint_t wc, FILE *file_ptr);
```

Arguments

wc

A value of type `wint_t`.

file_ptr

A file pointer.

Description

When using the `ungetwc` function, the wide character is pushed back onto the file indicated by *file_ptr*.

One push-back is guaranteed, even if there has been no previous activity on the file. If a file positioning function (such as `fseek`) is called before the pushed back character is read, the bytes representing the pushed back character are lost.

If the character to be pushed back is WEOF, the operation fails, the input stream is left unchanged, and WEOF is returned.

See also `getwc`.

Return Values

x

The push-back character.

WEOF

Indicates that the function cannot push the character back. `errno` is set to one of the following:

- EBADF – The file descriptor is not valid.
- EALREADY – Operation is already in progress on the same file.
- EILSEQ – Invalid wide-character code detected.

unlink

unlink — Deletes the specified symbolic link from the system.

Format

```
#include <unistd.h>
int unlink (const char *link_name);
```

Arguments

link_name

The name of the symbolic link to be deleted.

Description

The `unlink` function deletes the specified symbolic link (*link_name*) from the system. The contents of the symbolic link are not examined, and no action is performed on the file specified in the contents. For other files, the `unlink` function behaves the same as the C RTL `remove` function.

See also `symlink`, `readlink`, `realpath`, `lchown`, and `lstat`.

Return Values

0

Successful completion.

-1

Indicates an error. The named file (*link_name*) is unchanged, and `errno` is set to any `errno` value from `remove`.

unordered

unordered — Returns the value 1 (TRUE) if either or both of the arguments is a NaN. Otherwise, it returns the value 0 (FALSE).

Format

```
#include <math.h>
double unordered (double x, double y);
float unorderedf (float x, float y);
long double unorderedl (long double x, long double y);
```

Arguments

x

A real number.

y

A real number.

Return Values

1

Either or both of the arguments is a NaN.

0

Neither argument is a NaN.

unsetenv

unsetenv — Deletes all instances of the environment variable name from the environment list.

Format

```
#include <stdlib.h>
void unsetenv (const char *name);
```

Argument

name

The environment variable to delete from the environment list.

Description

The `unsetenv` function deletes all instances of the variable name pointed to by the *name* argument from the environment list.

usleep

usleep — Suspends execution for an interval.

Format

```
#include <unistd.h>
int usleep (unsigned int mseconds);
```

Argument

mseconds

The number of microseconds to suspend execution for.

Description

The `usleep` function suspends the current process from execution for the number of microseconds specified by the *mseconds* argument. This argument must be less than 1,000,000. However, if its value is 0, then the call has no effect.

Be aware that `usleep` time specifications are rounded up approximately to the next millisecond because that is the finest time interval granularity possible on OpenVMS systems.

There is one real-time interval timer for each process. The `usleep` function does not interfere with a previous setting of this timer. If the process set this timer before calling `usleep` and if the time specified by *mseconds* equals or exceeds the interval timer's prior setting, then the process is awakened shortly before the timer was set to expire.

Return Values

0

Indicates success.

-1

Indicates an error occurred; `errno` is set to EINVAL.

utime

utime — Sets file access and modification times.

Format

```
#include <utime.h>
int utime (const char *path, const struct utimbuf *times);
```

Arguments

path

A pointer to a file.

times

A NULL pointer or a pointer to a `utimbuf` structure.

Description

The `utime` function sets the access and modification times of the file named by the *path* argument. The file must be openable for write-access to use this function.

If *times* is a NULL pointer, the access and modification times of the file are set to the current time. To use `utime` in this way, the effective user ID of the process must match the owner of the file, or the process must have write permission to the file or have appropriate privileges.

If *times* is not a NULL pointer, it is interpreted as a pointer to a `utimbuf` structure, and the access and modification times are set to the values in the specified structure. Only a process with an effective user ID equal to the user ID of the file or a process with appropriate privileges can use `utime` this way.

The `utimbuf` structure is defined by the `<utime.h>` header. The times in the `utimbuf` structure are measured in seconds since the Epoch.

Upon successful completion, `utime` marks the time of the last file status change, `st_ctime`, to be updated. See the `<stat.h>` header file.

Note

(Integrity servers, Alpha) On OpenVMS Alpha and Integrity server systems, the `stat`, `fstat`, `utime`, and `utimes` functions have been enhanced to take advantage of the new file-system support for POSIX compliant file timestamps.

This support is available only on ODS-5 devices on OpenVMS Alpha systems beginning with a version of OpenVMS Alpha after Version 7.3.

Before this change, `stat` and `fstat` set the values of the `st_ctime`, `st_mtime`, and `st_atime` fields based on the following file attributes:

`st_ctime` – ATR$C_CREDATE (file creation time)
`st_mtime` – ATR$C_REVDATE (file revision time)
`st_atime` – was always set to `st_mtime` because no support for file access time was available

Also, for the file-modification time, `utime` and `utimes` were modifying the ATR$C_REVDATE file attribute, and ignoring the file-access-time argument.

After the change, for a file on an ODS-5 device, the `stat` and `fstat` functions set the values of the `st_ctime`, `st_mtime`, and `st_atime` fields based on the following new file attributes:

`st_ctime` – ATR$C_ATTDATE (last attribute modification time)
`st_mtime` – ATR$C_MODDATE (last data modification time)
`st_atime` – ATR$C_ACCDATE (last access time)

If ATR$C_ACCDATE is 0, as on an ODS-2 device, the `stat` and `fstat` functions set `st_atime` to `st_mtime`.

For the file-modification time, the `utime` and `utimes` functions modify both the ATR$C_REVDATE and ATR$C_MODDATE file attributes. For the file-access time, these functions modify the ATR$C_ACCDATE file attribute. Setting the ATR$C_MODDATE and ATR$C_ACCDATE file attributes on an ODS-2 device has no effect.

For compatibility, the old behavior of `stat`, `fstat`, `utime`, and `utimes` remains the default, regardless of the kind of device.

The new behavior must be explicitly enabled by defining the DECC$EFS_FILE_TIMESTAMPS logical name to "ENABLE" before invoking the application. Setting this logical does not affect the behavior of `stat`, `fstat`, `utime`, and `utimes` for files on an ODS-2 device.

Return Values

0

Successful execution.

-1

Indicates an error. The function sets `errno` to one of the following values:

The `utime` function *will* fail if:

- EACCES– Search permission is denied by a component of the *path* prefix; or the *times* argument is a NULL pointer and the effective user ID of the process does not match the owner of the file and write access is denied.
- ELOOP– Too many symbolic links were encountered in resolving *path*.
- ENAMETOOLONG– The length of the *path* argument exceeds PATH_MAX, a pathname component is longer than NAME_MAX, or a pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
- ENOENT– *path* does not name an existing file, or *path* is an empty string.
- ENOTDIR– A component of the *path* prefix is not a directory.

- EPERM– *times* is not a NULL pointer and the calling process's effective user ID has write-access to the file but does not match the owner of the file, and the calling process does not have the appropriate privileges.
- EROFS– The file system containing the file is read-only.

utimes

utimes — Sets file access and modification times.

Format

```
#include <time.h>
int utimes (const char *path, const struct timeval times[2]);
```

Arguments

path

A pointer to a file.

times

an array of `timeval` structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the `timeval` structure are measured in seconds and microseconds since the Epoch, although rounding toward the nearest second may occur.

Description

The `utimes` function sets the access and modification times of the file pointed to by the *path* argument to the value of the times argument. The `utimes` function allows time specifications accurate to the microsecond.

If the *times* argument is a NULL pointer, the access and modification times of the file are set to the current time. The effective user ID of the process must be the same as the owner of the file, or must have write access to the file or appropriate privileges to use this call in this manner.

Upon completion, `utimes` marks the time of the last file status change, `st_ctime`, for update.

Note

(Integrity servers, Alpha) On OpenVMS Alpha and Integrity server systems, the `stat`, `fstat`, `utime`, and `utimes` functions have been enhanced to take advantage of the new file-system support for POSIX compliant file timestamps.

This support is available only on ODS-5 devices on OpenVMS Alpha systems beginning with a version of OpenVMS Alpha after Version 7.3.

Before this change, the `stat` and `fstat` functions were setting the values of the `st_ctime`, `st_mtime`, and `st_atime` fields based on the following file attributes:

`st_ctime` – ATR$C_CREDATE (file creation time)

`st_mtime` – ATR$C_REVDATE (file revision time)
`st_atime` – was always set to `st_mtime` because no support for file access time was available

Also, for the file-modification time, `utime` and `utimes` were modifying the ATR$C_REVDATE file attribute, and ignoring the file-access-time argument.

After the change, for a file on an ODS-5 device, the `stat` and `fstat` functions set the values of the `st_ctime`, `st_mtime`, and `st_atime` fields based on the following new file attributes:

`st_ctime` – ATR$C_ATTDATE (last attribute modification time)
`st_mtime` – ATR$C_MODDATE (last data modification time)
`st_atime` – ATR$C_ACCDATE (last access time)

If ATR$C_ACCDATE is 0, as on an ODS-2 device, the `stat` and `fstat` functions set `st_atime` to `st_mtime`.

For the file-modification time, the `utime` and `utimes` functions modify both the ATR$C_REVDATE and ATR$C_MODDATE file attributes. For the file-access time, these functions modify the ATR$C_ACCDATE file attribute. Setting the ATR$C_MODDATE and ATR$C_ACCDATE file attributes on an ODS-2 device has no effect.

For compatibility, the old behavior of `stat`, `fstat`, `utime`, and `utimes` remains the default, regardless of the kind of device.

The new behavior must be explicitly enabled by defining the DECC$EFS_FILE_TIMESTAMPS logical name to "ENABLE" before invoking the application. Setting this logical does not affect the behavior of `stat`, `fstat`, `utime`, and `utimes` for files on an ODS-2 device.

Return Values

0

Successful execution.

-1

Indicates an error. The file times do not change and the function sets `errno` to one of the following values:

The `utimes` function *will* fail if:

- EACCES– Search permission is denied by a component of the *path* prefix; or the *times* argument is a NULL pointer and the effective user ID of the process does not match the owner of the file and write access is denied.
- ELOOP– Too many symbolic links were encountered in resolving *path*.
- ENAMETOOLONG– The length of the *path* argument exceeds PATH_MAX, a pathname component is longer than NAME_MAX, or a pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
- ENOENT– A component of *path* does not name an existing file, or *path* is an empty string.
- ENOTDIR– A component of the *path* prefix is not a directory.

- EPERM–The *times* argument is not a NULL pointer and the calling process's effective user ID has write-access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.
- EROFS– The file system containing the file is read-only.

VAXC$CRTL_INIT

VAXC$CRTL_INIT — Allows you to call the C RTL from other languages or to use the C RTL when your main function is not in C. It initializes the run-time environment and establishes both an exit and condition handler. `VAXC$CRTL_INIT` is a synonym for `DECC$CRTL_INIT`. Either name invokes the same routine.

Format

```
#include <signal.h>
void VAXC$CRTL_INIT();
```

Description

The following example shows a Pascal program that calls the C RTL using the `VAXC$CRTL_INIT` function:

On OpenVMS Alpha systems:

```
$ PASCAL EXAMPLE
$ LINK EXAMPLE,SYS$LIBRARY:VAXCRTL/LIB
$ TY EXAMPLE.PAS
PROGRAM TESTC(input, output);
PROCEDURE VAXC$CRTL_INIT; extern;
BEGIN
   VAXC$CRTL_INIT;
END
$
```

A shareable image need only call this function if it contains an VSI C function for signal handling, environment variables, I/O, exit handling, a default file protection mask, or if it is a child process that should inherit context.

Although many of the initialization activities are performed only once, `DECC$CRTL_INIT` can safely be called multiple times. On OpenVMS VAX systems, `DECC$CRTL_INIT` establishes the C RTL internal OpenVMS exception handler in the frame of the routine that calls `DECC$CRTL_INIT` each time `DECC$CRTL_INIT` is called.

At least one frame in the current call stack must have that handler established for OpenVMS exceptions to get mapped to UNIX signals.

VAXC$ESTABLISH

VAXC$ESTABLISH — Used for establishing an OpenVMS exception handler for a particular routine. This function establishes a special C RTL exception handler in the routine that called it. This special handler catches all RTL-related exceptions that occur in later routines, and passes on all other exceptions to your handler.

Format

```
#include <signal.h>
void VAXC$ESTABLISH (unsigned int (*exception_handler)(void *sigarr,
void *mecharr));
```

Arguments

exception_handler

The name of the function that you want to establish as an OpenVMS exception handler. You pass a pointer to this function as the parameter to VAXC$ESTABLISH.

sigarr

A pointer to the signal array.

mecharr

A pointer to the mechanism array.

Description

VAXC$ESTABLISH must be used in place of LIB$ESTABLISH when programs use the C RTL routines `setjmp` or `longjmp`. See `setjmp` and `longjmp`, or `sigsetjmp` and `siglongjmp`.

You can only invoke the VAXC$ESTABLISH function from a VSI C for OpenVMS function, because it relies on the allocation of data space on the run-time stack by the VSI C compiler. Calling the OpenVMS system library routine LIB$ESTABLISH directly from an VSI C function results in undefined behavior from the `setjmp` and `longjmp` functions.

To cause an OpenVMS exception to generate a UNIX style signal, user exception handlers must return SS$_RESIGNAL upon receiving any exception that they do not want to handle. Returning SS$_NORMAL prevents the generation of a UNIX style signal. UNIX signals are generated as if by an exception handler in the stack frame of the main C program. Not all OpenVMS exceptions correspond to UNIX signals. See Chapter 4 for more information on the interaction of OpenVMS exceptions and UNIX style signals.

Calling VAXC$ESTABLISH with an argument of NULL cancels an existing handler in that routine.

Note

On OpenVMS Alpha systems, VAXC$ESTABLISH is implemented as a compiler built-in function, not as a C RTL function. (Alpha only)

---

va_arg

va_arg — Returns the next item in the argument list.

Format

```
#include <stdarg.h> (ANSI C)
```

```
#include <varargs.h> (VSI C Extension)
type va_arg (va_list ap, type);
```

Arguments

ap

A variable list containing the next argument to be obtained.

type

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

Description

The `va_arg` macro interprets the object at the address specified by the list increment or according to type. If there is no corresponding argument, the behavior is undefined.

When using `va_arg` to write portable applications, include the `<stdarg.h>` header file (defined by the ANSI C standard), not the `<varargs.h>` header file, and use `va_arg` only in conjunction with other functions and macros defined in `<stdarg.h>`.

For an example of argument-list processing using the `<stdarg.h>` functions and definitions, see Example 3.6.

va_copy

va_copy — Copies one argument list to another.

Format

```
#include <stdarg.h>
void va_copy (va_list dest, va_list src);
```

Arguments

dest

Argument list that is a copy of *src*.

src

Source argument list to be copied to *dest*.

Description

The `va_copy` macro initializes *dest* as a copy of *src*, as if the `va_start` macro had been applied to *dest* followed by the same sequence of uses of the `va_arg` macro as had previously been used to reach the present state of *src*. Neither the `va_copy` nor `va_start` macro should be invoked to reinitialize *dest* without an intervening invocation of the `va_end` macro for the same *dest*.

Note

The `va_copy` macro requires VSI C Compiler Version 7.5 or later.

va_count

va_count — Returns the number of quadwords (Alpha only) in the argument list.

Format

```
#include <stdarg.h> (ANSI C)
OR
#include <varargs.h> (VSI C Extension)
void va_count (int count);
```

Argument

count

An integer variable name in which the number of quadwords (Alpha only) is returned.

Description

The `va_count` macro places the number of quadwords (Alpha only) in the argument list into *count*. The value returned in *count* is the number of quadwords (Alpha only) in the function argument block not counting the *count* field itself.

If the argument list contains items whose storage requirements are a quadword (Alpha only) of memory or less, the number in the *count* argument is also the number of arguments. However, if the argument list contains items that are longer than a quadword (Alpha only), *count* must be interpreted to obtain the number of arguments. Because a `double` is 8 bytes, it occupies one argument-list position on OpenVMS Alpha and Integrity server systems.

The `va_count` macro is specific to VSI C for OpenVMS systems and is not portable.

va_end

va_end — Finishes the <varargs.h> or <stdarg.h> session.

Format

```
#include <stdarg.h> (ANSI C)
#include <varargs.h> (VSI C Extension)
void va_end (va_list ap);
```

Argument

ap

The object used to traverse the argument list length. You must declare and use the argument *ap* as shown in this format section.

Description

You can execute multiple traversals of the argument list, each delimited by va_start . . . va_end. The va_end function sets *ap* equal to NULL.

When using this function to write portable applications, include the <stdarg.h> header file (defined by the ANSI C standard), not the <varargs.h> header file, and use va_end only in conjunction with other routines defined in <stdarg.h>.

For an example of argument-list processing using the <stdarg.h> functions and definitions, see Example 3.6.

va_start, va_start_1

va_start, va_start_1 — Used for initializing a variable to the beginning of the argument list.

Format

```
#include <varargs.h> (VSI C Extension)
void va_start (va_list ap);
void va_start_1 (va_list ap, int offset);
```

Arguments

ap

An object pointer. You must declare and use the argument *ap* as shown in the format section.

offset

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

Description

The va_start macro initializes the variable *ap* to the beginning of the argument list.

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

When determining the value of the offset argument used in `va_start_1`, the implications of the OpenVMS calling standard must be considered.

On OpenVMS Alpha and Integrity server systems, each argument item is a quadword.

Note

When accessing argument lists, especially those passed to a subroutine (written in C) by a program written in another programming language, consider the implications of the OpenVMS calling standard. For more information about the OpenVMS calling standard, see the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/] or the *VSI OpenVMS Calling Standard*.

The preceding version of `va_start` and `va_start_1` is specific to the C RTL, and is not portable.

The following syntax describes the `va_start` macro in the `<stdarg.h>` header file, as defined in the ANSI C standard.

Format

```
#include <stdarg.h> (ANSI C)
void va_start (va_list ap, parmN);
```

Arguments

ap

An object pointer. You must declare and use the argument *ap* as shown in the format section.

parmN

The name of the last of the known fixed arguments.

Description

The pointer *ap* is initialized to point to the first of the optional arguments that follow *parmN* in the argument list.

Always use this version of `va_start` in conjunction with functions that are declared and defined with function prototypes. Also use this version of `va_start` to write portable programs.

For an example of argument-list processing using the `<stdarg.h>` functions and definitions, see Example 3.6.

vfork

vfork — Creates an independent child process. This function is nonreentrant.

Format

```
#include <unistd.h>
int vfork (void); (_DECC_V4_SOURCE)
pid_t vfork (void); (not _DECC_V4_SOURCE)
```

Description

The `vfork` function provided by VSI C for OpenVMS systems differs from the `fork` function provided by other C implementations. Table 44 shows the two major differences.

Table 44. The vfork and fork Functions

The vfork Function	The fork Function
Used with the `exec` functions.	Can be used without an `exec` function for asynchronous processing.
Creates an independent child process that shares some of the parent's characteristics.	Creates an exact duplicate of the parent process that branches at the point where `vfork` is called, as if the parent and the child are the same process at different stages of execution.

The `vfork` function provides the setup necessary for a subsequent call to an `exec` function. Although no process is created by `vfork`, it performs the following steps:

- It saves the return address (the address of the `vfork` call) to be used later as the return address for the call to an `exec` function.
- It saves the current context.
- It returns the integer 0 the first time it is called (before the call to an `exec` function is made). After the corresponding `exec` function call is made, the `exec` function returns control to the parent process, at the point of the `vfork` call, and it returns the process ID of the child as the return value. Unless the `exec` function fails, control appears to return twice from `vfork` even though one call was made to `vfork` and one call was made to the `exec` function.

The behavior of the `vfork` function is similar to the behavior of the `setjmp` function. Both `vfork` and `setjmp` establish a return address for later use, both return the integer 0 when they are first called to set up this address, and both pass back the second return value as though it were returned by them rather than by their corresponding `exec` or `longjmp` function calls.

However, unlike `setjmp`, with `vfork`, all local automatic variables, even those with volatile-qualified type, can have indeterminate values if they are modified between the call to `vfork` and the corresponding call to an `exec` routine.

Return Values

0

Indicates successful creation of the context.

nonzero

Indicates the process ID (PID) of the child process.

-1

Indicates an error – failure to create the child process.

vfprintf

vfprintf — Prints formatted output based on an argument list.

Format

```
#include <stdio.h>
int vfprintf (FILE *file_ptr, constchar *format, va_list ap);
```

Arguments

file_ptr

A pointer to the file to which the output is directed.

format

A pointer to a string containing the format specification. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

ap

A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

See also `vprintf` and `vsprintf`.

Return Values

x

The number of bytes written.

Negative value

Indicates an output error. The function sets `errno`. For a list of possible `errno` values set, see `fprintf`.

vfscanf

vfscanf — Reads formatted input based on an argument list.

Format

```
#include <stdio.h>
int vfscanf (FILE *file_ptr, constchar *format, va_list ap);
```

Arguments

file_ptr

A pointer to the file that provides input text.

format

A pointer to a string containing the format specification.

ap

A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The `vfscanf` function is the same as the `fscanf` function except that instead of being called with a variable number of arguments, it is called with an argument list that has been initialized by `va_start` (and possibly subsequent `va_arg` calls).

If no conversion specifications are given, you can omit the input pointers. Otherwise, the function calls must have exactly as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input pointers.

Conversion specifications are matched to input sources in left-to-right order. Excess input pointers, if any, are ignored.

For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

This function returns the number of successfully matched and assigned input items.

See also `vscanf` and `vsscanf`.

Return Values

n

The number of successfully matched and assigned input items.

EOF

Indicates that the end-of-file was encountered or a read error occurred. If a read error occurs, the function sets `errno` to one of the following:

- EILSEQ– Invalid character detected.
- EINVAL– Insufficient arguments.
- ENOMEM– Not enough memory available for conversion.
- ERANGE– Floating-point calculations overflow.
- EVMSERR– Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This can indicate that conversion to a numeric value failed due to overflow.

- The function can also set `errno` to the following as a result of errors returned from the I/O subsystem:
  - EBADF– The file descriptor is not valid.
  - EIO– I/O error.
  - ENXIO– Device does not exist.
  - EPIPE– Broken pipe.
  - EVMSERR– Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This indicates that an I/O error occurred for which there is no equivalent C error code.

vfwprintf

vfwprintf — Writes output to the stream under control of the wide-character format string.

Format

```
#include <wchar.h>
int vfwprintf (FILE *stream, const wchar_t *format, va_list ap);
```

Arguments

stream

A file pointer.

format

A pointer to a wide-character string containing the format specifications. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

ap

A variable list of the items needed for output.

Description

The `vfwprintf` function is equivalent to the `fwprintf` function, with the variable argument list replaced by the *ap* argument. Initialize *ap* with the `va_start` macro (and possibly with subsequent `va_arg` calls) from `<stdarg.h>`.

If the stream pointed to by *stream* has no orientation, `vfwprintf` makes the stream wide-oriented.

See also `fwprintf`.

Return Values

n

The number of wide characters written.

Negative value

Indicates an error. The function sets `errno` to one of the following:

- EILSEQ – Invalid character detected.
- EINVAL – Insufficient arguments.
- ENOMEM – Not enough memory available for conversion.
- ERANGE – Floating-point calculations overflow.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This might indicate that conversion to a numeric value failed because of overflow.

The function can also set `errno` to the following as a result of errors returned from the I/O subsystem:

- EBADF – The file descriptor is not valid.
- EIO – I/O error.
- ENOSPC – No free space on the device containing the file.
- ENXIO – Device does not exist.
- EPIPE – Broken pipe.
- ESPIPE – Illegal seek in a file opened for append.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This indicates that an I/O error occurred for which there is no equivalent C error code.

Examples

The following example shows the use of the `vfwprintf` function in a general error reporting routine:

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

void error(char *function_name, wchar_t *format, ...);
{
   va_list args;

   va_start(args, format);
   /* print out name of function causing error */
   fwprintf(stderr, L"ERROR in %s: ", function_name);
   /* print out remainder of message */
   vfwprintf(stderr, format, args);
   va_end(args);
}
```

vfwscanf

vfwscanf — Reads input from the stream under control of a wide-character format string.

Format

```
#include <wchar.h>
int vfwscanf (FILE *stream, const wchar_t *format, va_list ap);
```

Arguments

stream

A file pointer.

format

A pointer to a wide-character string containing the format specifications.

ap

A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The `vfwscanf` function is equivalent to the `fwscanf` function, except that instead of being called with a variable number of arguments, it is called with an argument list (*ap*) that has been initialized by `va_start` (and possibly with subsequent `va_arg` calls).

If the stream pointed to by *stream* has no orientation, `vfwscanf` makes the stream wide-oriented.

For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

Return Values

n

The number of successfully matched and assigned wide-character input items.

EOF

Indicates that a read error occurred before any conversion. The function sets `errno`. For a list of the values set by this function, see `vfscanf`.

vprintf

vprintf — Prints formatted output based on an argument list. This function is the same as the `printf` function except that instead of being called with a variable number of arguments, it is called with an argument list that has been initialized by the `va_start` macro (and possibly with subsequent `va_arg` calls) from `<stdarg.h>`.

Format

```
#include <stdio.h>
int vprintf (const char *format, va_list ap);
```

Arguments

format

A pointer to the string containing the format specification. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

ap

A variable list of the items needed for output.

Description

See the `vfprintf` and `vsprintf` functions.

Return Values

x

The number of bytes written.

Negative value

Indicates an output error. The function sets `errno`. For a list of possible `errno` values set, see `fprintf`.

vscanf

vscanf — Reads formatted input based on an argument list.

Format

```
#include <stdio.h>
int vscanf (const char *format, va_list ap);
```

Arguments

format

A pointer to the string containing the format specification.

ap

A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The `vscanf` function is the same as the `scanf` function except that instead of being called with a variable number of arguments, it is called with an argument list (*ap*) that has been initialized by the `va_start` macro (and possibly with subsequent `va_arg` calls).

For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

See also `scanf`, `vfscanf`, and `vsscanf`.

Return Values

n

The number of successfully matched and assigned input items.

EOF

Indicates that a read error occurred before any conversion. The function sets `errno`. For a list of the values set by this function, see `vfscanf`.

vsnprintf

vsnprintf — Prints formatted output based on an argument list.

Format

```
#include <stdio.h>
int vsnprintf (char *str, size_t n, const char *format, va_list ap);
```

Arguments

str

A pointer to a string that will receive the formatted output.

format

A pointer to a character string that contains the format specification. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

ap

A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The `vsnprintf` function is the same as the `snprintf` function, but instead of being called with a variable number of arguments, it is called with an argument list that has been initialized by `va_start` (and possibly with subsequent `va_arg` calls).

This function does not invoke the `va_end` macro. Because the function invokes the `va_arg` macro, the value of *ap* after the return is unspecified.

Applications using `vsnprintf` should call `va_end` (*ap*) afterwards to clean up.

Return Values

x

The number of bytes (excluding the terminating null byte) that would be written to *str* if *n* is sufficiently large.

Negative value

Indicates an output error occurred. The function sets `errno`. For a list of possible `errno` values set, see `fprintf`.

vsprintf

vsprintf — Prints formatted output based on an argument list. This function is the same as the `sprintf` function except that instead of being called with a variable number of arguments, it is called with an argument list that has been initialized by `va_start` (and possibly with subsequent `va_arg` calls).

Format

```
#include <stdio.h>
int vsprintf (char *str, const char *format, va_list ap);
```

Arguments

str

A pointer to a string that will receive the formatted output. This string is assumed to be large enough to hold the output.

format

A pointer to a character string that contains the format specification. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

ap

A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Return Value

x

The number of bytes written.

Negative value

Indicates an output error occurred. The function sets `errno`. For a list of possible `errno` values set, see `fprintf`.

vsscanf

vsscanf — Reads formatted input based on an argument list.

Format

```
#include <stdio.h>
int vsscanf (char *str, const char *format, va_list ap);
```

Arguments

str

The address of the character string that provides the input text to `sscanf`.

format

A pointer to a character string that contains the format specification.

ap

A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The `vsscanf` function is the same as the `sscanf` function except that instead of being called with a variable number of arguments, it is called with an argument list that has been initialized by `va_start` (and possibly with subsequent `va_arg` calls).

The `vsscanf` function is also equivalent to the `vfscanf` function, except that the first argument specifies a wide-character string rather than a stream. Reaching the end of the wide-character string is the same as encountering EOF for the `vfscanf` function.

For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

See also `vsscanf` and `sscanf`.

Return Values

n

The number of successfully matched and assigned input items.

EOF

Indicates that a read error occurred before any conversion. The function sets `errno`. For a list of the values set by this function, see `vfscanf`.

vswprintf

vswprintf — Writes output to the stream under control of the wide-character format string.

Format

```
#include <wchar.h>
int vswprintf (wchar_t *s, size_t n, const wchar_t *format, va_list ap);
```

Arguments

s

A pointer to a multibyte character sequence.

n

The maximum number of bytes that comprise the multibyte character.

format

A pointer to a wide-character string containing the format specifications. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

ap

A variable list of the items needed for output.

Description

The `vswprintf` function is equivalent to the `swprintf` function, with the variable argument list replaced by the *ap* argument. Initialize *ap* with the `va_start` macro, and possibly with subsequent `va_arg` calls.

See also `swprintf`.

Return Values

n

The number of wide characters written.

Negative value

Indicates an error. The function sets `errno` to one of the following:

- EILSEQ – Invalid character detected.
- EINVAL – Insufficient arguments.
- ENOMEM – Not enough memory available for conversion.
- ERANGE – Floating-point calculations overflow.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This might indicate that conversion to a numeric value failed because of overflow.

The function can also set `errno` to the following as a result of errors returned from the I/O subsystem:

- EBADF – The file descriptor is not valid.
- EIO – I/O error.
- ENOSPC – No free space on the device containing the file.
- ENXIO – Device does not exist.
- EPIPE – Broken pipe.

- ESPIPE – Illegal seek in a file opened for append.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This indicates that an I/O error occurred for which there is no equivalent C error code.

vswscanf

vswscanf — Reads input from the stream under control of the wide-character format string.

Format

```
#include <wchar.h>
int vswscanf (wchar_t *s, const wchar_t *format, va_list ap);
```

Arguments

s

A pointer to a wide-character string from which the input is to be obtained.

format

A pointer to a wide-character string containing the format specifications.

ap

A list of expressions whose results correspond to conversion specifications given in the format specification.

Description

The `vswscanf` function is equivalent to the `swscanf` function, except that instead of being called with a variable number of arguments, it is called with an argument list (*ap*) that has been initialized by `va_start` (and possibly with subsequent `va_arg` calls).

The `vswscanf` function is also equivalent to the `vfwscanf` function, except that the first argument specifies a wide-character string rather than a stream. Reaching the end of the wide-character string is the same as encountering EOF for the `vfwscanf` function.

For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

See also `vfwscanf` and `swscanf`.

Return Values

n

The number of wide characters read.

EOF

Indicates that a read error occurred before any conversion. The function sets `errno`. For a list of the values set by this function, see `vfscanf`.

vwprintf

vwprintf — Writes output to an array of wide characters under control of the wide-character format string.

Format

```
#include <wchar.h>
int vwprintf (const wchar_t *format, va_list ap);
```

Arguments

format

A pointer to a wide-character string containing the format specifications. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

ap

The variable list of items needed for output.

Description

The `vwprintf` function is equivalent to the `wprintf` function, with the variable argument list replaced by the *ap* argument. Initialize `ap` with the `va_start` macro, and possibly with subsequent `va_arg` calls. The `vwprintf` function does not invoke the `va_end` macro.

See also `wprintf`.

Return Values

x

The number of wide characters written, not counting the terminating null wide character.

Negative value

Indicates an error. Either *n* or more wide characters were requested to be written, or a conversion error occurred, in which case `errno` is set to EILSEQ.

vwscanf

vwscanf — Reads input from an array of wide characters under control of a wide-character format string.

Format

```
#include <wchar.h>
int vwscanf (const wchar_t *format, va_list ap);
```

Arguments

format

A pointer to a wide-character string containing the format specifications.

ap

A list of expressions whose resultant types correspond to the conversion specifications given in the format specifications.

Description

The `vwscanf` function is equivalent to the `wscanf` function, except that instead of being called with a variable number of arguments, it is called with an argument list (*ap*) that has been initialized by `va_start` (and possibly with subsequent `va_arg` calls).

For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

See also `wscanf`.

Return Values

n

The number of wide characters read.

EOF

Indicates that a read error occurred before any conversion. The function sets `errno`. For a list of the values set by this function, see `vfscanf`.

wait

wait — Checks the status of the child process before exiting. A child process is terminated when the parent process terminates.

Format

```
#include <wait.h>
pid_t wait (int *status);
```

Argument

status

The address of a location to receive the final status of the terminated child. The child can set the status with the `exit` function and the parent can retrieve this value by specifying *status*.

Description

The `wait` function suspends the parent process until the final status of a terminated child is returned from the child.

On OpenVMS Version 7.0 and higher systems, the `wait` function is equivalent to `waitpid( 0, status, 0)` if you include `<wait.h>` and compile with the `_POSIX_EXIT` feature-test macro

set (either with /DEFINE=_POSIX_EXIT or with `#define _POSIX_EXIT` at the top of your file, before any file inclusions).

Return Values

x

The process ID (PID) of the terminated child. If more than one child process was created, `wait` will return the PID of the terminated child that was most recently created. Subsequent calls will return the PID of the next most recently created, but terminated, child.

-1

No child process was spawned.

wait3

wait3 — Waits for a child process to stop or terminate.

Format

```
#include <wait.h>
pid_t wait3 (int *status_location, int options,
struct rusage *resource_usage);
```

Arguments

status_location

A pointer to a location that contains the termination status of the child process as defined in the `<wait.h>` header file.

Beginning with OpenVMS Version 7.2, when compiled with the _VMS_WAIT macro defined, the `wait3` function puts the OpenVMS completion code of the child process at the address specified in the *status_location* argument.

options

Flags that modify the behavior of the function. These flags are defined in the Description section.

resource_usage

The location of a structure that contains the resource utilization information for terminated child processes.

Description

The `wait3` function suspends the calling process until the request is completed, and redefines it so that only the calling thread is suspended.

The *options* argument modifies the behavior of the function. You can combine the flags for the *options* argument by specifying their bitwise inclusive OR. The flags are:

WNOWAIT	Specifies that the process whose status is returned in *status_location* is kept in a waitable state. You can wait for the process again with the same results.
WNOHANG	Prevents the suspension of the calling process. If there are child processes that stopped or terminated, one is chosen and the waitpid function returns its process ID, as when you do not specify the WNOHANG flag. If there are no terminated processes (that is, if waitpid suspends the calling process without the WNOHANG flag), 0 (zero) is returned. Because you can never wait for process 0, there is no confusion arising from this return.
WUNTRACED	Specifies that the call return additional information when the child processes of the current process stop because the child process received a SIGTTIN, SIGTTOU, SIGSTOP, or SIGTSTOP signal.

If the wait3 function returns because the status of a child process is available, the process ID of the child process is returned. Information is stored in the location pointed to by *status_location*, if this pointer is not null.

The value stored in the location pointed to by *status_location* is 0 (zero) only if the status is returned from a terminated child process that did one of the following:

- Returned 0 from the main function.
- Passed 0 as the *status* argument to the _exit or exit function.

Regardless of the *status_location* value, you can define this information using the macros defined in the <wait.h> header file, which evaluate to integral expressions. In the following macro descriptions, the *status_value* argument is equal to the integer value pointed to by the *status_location* argument:

WIFEXITED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that terminated normally.
WEXITSTATUS(*status_value*)	If the value of WIFEXITED (*status_value*) is nonzero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to the _exit or exit function, or to the value the child process returned from the main function.
WIFSIGNALED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that terminated due to the receipt of a signal that was not intercepted.
WTERMSIG(*status_value*)	If the value of WIFSIGNALED (*status_value*) is nonzero, this macro evaluates to the number of the signal that caused the termination of the child process.
WIFSTOPPED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that is currently stopped.
WSTOPSIG(*status_value*)	If the value of WIFSTOPPED (*status_value*) is nonzero, this macro evaluates to the number of the signal that caused the child process to stop.
WIFCONTINUED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that has continued.

If the information stored at the location pointed to by *status_location* was stored there by a call to wait3 that specified the WUNTRACED flag, one of the following macros evaluates to a nonzero value:

- WIFEXITED(**status_value*)

- `WIFSIGNALED`(**status_value*)
- `WIFSTOPPED`(**status_value*)
- `WIFCONTINUED`(**status_value*)

If the information stored in the location pointed to by *status_location* resulted from a call to `wait3` without the `WUNTRACED` flag specified, one of the following macros evaluates to a nonzero value:

- `WIFEXITED`(**status_value*)
- `WIFSIGNALED`(**status_value*)

The `wait3` function provides compatibility with BSD systems. The *resource_usage* argument points to a location that contains resource usage information for the child processes as defined in the `<resource.h>` header file.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes is assigned a parent process ID equal to the process ID of the `init` process.

See also `exit`, `-exit`, and `init`.

Return Values

0

Indicates success. There are no stopped or exited child processes, the `WNOHANG` option is specified.

x

The *process_id* of the child process. The status of a child process is available.

-1

Indicates an error; `errno` is set to one of the following values:

- ECHILD – There are no child processes to wait for.
- EINTR – Terminated by receipt of a signal intercepted by the calling process.
- EFAULT– The *status_location* or *resource_usage* argument points to a location outside of the address space of the process.
- EINVAL – The value of the *options* argument is not valid.

wait4

wait4 — Waits for a child process to stop or terminate.

Format

```
#include <wait.h>
pid_t wait4 (pid_t process_id union wait *status_location, int options,
struct rusage *resource_usage);
```

Arguments

status_location

A pointer to a location that contains the termination status of the child process as defined in the `<wait.h>` header file.

Beginning with OpenVMS Version 7.2, when compiled with the _VMS_WAIT macro defined, the `wait4` function puts the OpenVMS completion code of the child process at the address specified in the *status_location* argument.

process_id

The child process or set of child processes.

options

Flags that modify the behavior of the function. These flags are defined in the Description section.

resource_usage

The location of a structure that contains the resource utilization information for terminated child processes.

Description

The `wait4` function suspends the calling process until the request is completed.

The *process_id* argument allows the calling process to gather status from a specific set of child processes, according to the following rules:

If the *process_id* is	**Then status is requested**
Equal to -1	For any child process. In this respect, the `waitpid` function is equivalent to the `wait` function.
Greater than 0	For a single child process and specifies the process ID.

The `wait4` function only returns the status of a child process from this set.

The *options* argument to the `wait4` function modifies the behavior of the function. You can combine the flags for the *options* argument by specifying their bitwise-inclusive OR. The flags are:

`WNOWAIT`	Specifies that the process whose status is returned in *status_location* is kept in a waitable state. You can wait for the process again with the same results.
`WNOHANG`	Prevents the suspension of the calling process. If there are child processes that stopped or terminated, one is chosen and the `waitpid` function returns its process ID, as when you do not specify the `WNOHANG` flag. If there are no terminated processes (that is, if `waitpid` suspends the calling process without the `WNOHANG` flag), 0 is returned. Because you can never wait for process 0, there is no confusion arising from this return.
`WUNTRACED`	Specifies that the call return additional information when the child processes of the current process stop because the child process received a SIGTTIN, SIGTTOU, SIGSTOP, or SIGTSTOP signal.

If the wait4 function returns because the status of a child process is available, the process ID of the child process is returned. Information is stored in the location pointed to by *status_location*, if this pointer is not null.

The value stored in the location pointed to by *status_location* is 0 only if the status is returned from a terminated child process that did one of the following:

- Returned 0 from the main function.
- Passed 0 as the *status* argument to the _exit or exit function.

Regardless of the *status_location* value, you can define this information using the macros defined in the <wait.h> header file, which evaluate to integral expressions. In the following macro descriptions, *status_value* is equal to the integer value pointed to by *status_location*:

WIFEXITED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that terminated normally.
WEXITSTATUS(*status_value*)	If the value of WIFEXITED (*status_value*) is nonzero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to the _exit or exit function, or to the value the child process returned from the main function.
WIFSIGNALED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that terminated due to the receipt of a signal that was not intercepted.
WTERMSIG(*status_value*)	If the value of WIFSIGNALED (*status_value*) is nonzero, this macro evaluates to the number of the signal that caused the termination of the child process.
WIFSTOPPED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that is currently stopped.
WSTOPSIG(*status_value*)	If the value of WIFSTOPPED (*status_value*) is nonzero, this macro evaluates to the number of the signal that caused the child process to stop.
WIFCONTINUED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that has continued.

If the information stored at the location pointed to by *status_location* was stored there by a call to wait4 that specified the WUNTRACED flag, one of the following macros evaluates to a nonzero value:

- WIFEXITED(**status_value*)
- WIFSIGNALED(**status_value*)
- WIFSTOPPED(**status_value*)
- WIFCONTINUED(**status_value*)

If the information stored in the location pointed to by *status_location* resulted from a call to wait4 without the WUNTRACED flag specified, one of the following macros evaluates to a nonzero value:

- WIFEXITED(**status_value*)
- WIFSIGNALED(**status_value*)

The `wait4` function is similar to the `wait3` function. However, the `wait4` function waits for a specific child as indicated by the *process_id* argument. The *resource_usage* argument points to a location that contains resource usage information for the child processes as defined in the `<resource.h>` header file.

See also `exit` and `_exit`.

Return Values

0

Indicates success. There are no stopped or exited child processes, the `WNOHANG` option is specified.

x

The *process_id* of the child process. The status of a child process is available.

-1

Indicates an error; `errno` is set to one of the following values:

- ECHILD – There are no child processes to wait for.
- EINTR – Terminated by receipt of a signal intercepted by the calling process.
- EFAULT– The *status_location* or *resource_usage* argument points to a location outside of the address space of the process.
- EINVAL – The value of the *options* argument is not valid.

waitpid

waitpid — Waits for a child process to stop or terminate.

Format

```
#include <wait.h>
pid_t waitpid (pid_t process_id, int *status_location, int options);
```

Arguments

process_id

The child process or set of child processes.

status_location

A pointer to a location that contains the termination status of the child process as defined in the `<wait.h>` header file.

Beginning with OpenVMS Version 7.2, when compiled with the _VMS_WAIT macro defined, the `waitpid` function puts the OpenVMS completion code of the child process at the address specified in the *status_location* argument.

options

Flags that modify the behavior of the function. These flags are defined in the Description section.

Description

The `waitpid` function suspends the calling process until the request is completed. It is redefined so that only the calling thread is suspended.

If the *process_id* argument is -1 and the *options* argument is 0, the `waitpid` function behaves the same as the `wait` function. If these arguments have other values, the `waitpid` function is changed as specified by those values.

The *process_id* argument allows the calling process to gather status from a specific set of child processes, according to the following rules:

If the *process_id* is	**Then status is requested**
Equal to -1	For any child process. In this respect, the `waitpid` function is equivalent to the `wait` function.
Greater than 0	For a single child process and specifies the process ID.

The `waitpid` function only returns the status of a child process from this set.

The *options* argument to the `waitpid` function modifies the behavior of the function. You can combine the flags for the *options* argument by specifying their bitwise-inclusive OR. The flags are:

`WCONTINUED`	Specifies that the following is reported to the calling process: the status of any continued child process specified by the *process_id* argument whose status is unreported since it continued.
`WNOWAIT`	Specifies that the process whose status is returned in *status_location* is kept in a waitable state. You can wait for the process again with the same results.
`WNOHANG`	Prevents the calling process from being suspended. If there are child processes that stopped or terminated, one is chosen and `waitpid` returns its PID, as when you do not specify the `WNOHANG` flag. If there are no terminated processes (that is, if `waitpid` suspends the calling process without the `WNOHANG` flag), 0 (zero) is returned. Because you can never wait for process 0, there is no confusion arising from this return.
`WUNTRACED`	Specifies that the call return additional information when the child processes of the current process stop because the child process received a SIGTTIN, SIGTTOU, SIGSTOP, or SIGTSTOP signal.

If the `waitpid` function returns because the status of a child process is available, the process ID of the child process is returned. Information is stored in the location pointed to by *status_location*, if this pointer is not null. The value stored in the location pointed to by *status_location* is 0 only if the status is returned from a terminated child process that did one of the following:

- Returned 0 from the `main` function.
- Passed 0 as the *status* argument to the `_exit` or `exit` function.

Regardless of the value of *status_location*, you can define this information using the macros defined in the `<wait.h>` header file, which evaluate to integral expressions. In the following function descriptions, *status_value* is equal to the integer value pointed to by *status_location*:

WIFEXITED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that terminated normally.
WEXITSTATUS(*status_value*)	If the value of WIFEXITED (*status_value*) is nonzero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to the _exit or exit function, or to the value the child process returned from the main function.
WIFSIGNALED(*status_value*)	Evaluates to a nonzero value if status returned for a child process that terminated due to the receipt of a signal not intercepted.
WTERMSIG(*status_value*)	If the value of WIFSIGNALED (*status_value*) is nonzero, this macro evaluates to the number of the signal that caused the termination of the child process.
WIFSTOPPED(*status_value*)	Evaluates to a nonzero value if status was returned for a child process that is currently stopped.
WSTOPSIG(*status_value*)	If the value of WIFSTOPPED (*status_value*) is nonzero, this macro evaluates to the number of the signal that caused the child process to stop.
WIFCONTINUED(*status_value*)	Evaluates to a nonzero value if status returned for a child process that continued.

If the information stored at the location pointed to by *status_location* is stored there by a call to waitpid that specified the WUNTRACED flag, one of the following macros evaluates to a nonzero value:

- WIFEXITED(**status_value*)
- WIFSIGNALED(**status_value*)
- WIFSTOPPED(**status_value*)
- WIFCONTINUED(**status_value*)

If the information stored in the buffer pointed to by *status_location* resulted from a call to waitpid without the WUNTRACED flag specified, one of the following macros evaluates to a nonzero value:

- WIFEXITED(**status_value*)
- WIFSIGNALED(**status_value*)

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes is assigned a parent process ID equal to the process ID of the in it process.

See also exit, _exit, and wait.

Return Values

0

Indicates success. If the WNOHANG option was specified, and there are no stopped or exited child processes, the waitpid function also returns a value of 0.

-1

Indicates an error; errno is set to one of the following values:

- ECHILD – The calling process has no existing unwaited-for child processes. The process or process group ID specified by the *process_id* argument does not exist or is not a child process of the calling process.
- EINTR – The function was terminated by receipt of a signal.

  If the `waitpid` function returns because the status of a child process is available, the process ID of the child is returned to the calling process. If they return because a signal was intercepted by the calling process, -1 is returned.
- EFAULT – The *status_location* argument points to a location outside of the address space of the process.
- EINVAL – The value of the *options* argument is not valid.

wcrtomb

wcrtomb — Converts the wide character to its multibyte character representation.

Format

```
#include <wchar.h>
size_t wcrtomb (char *s, wchar_t wc, mbstate_t *ps);
```

Arguments

s

A pointer to the resulting multibyte character.

wc

A wide character.

ps

A pointer to the `mbstate_t` object. If a NULL pointer is specified, the function uses its internal `mbstate_t` object. `mbstate_t` is an opaque datatype intended to keep the conversion state for the state-dependent codesets.

Description

If *s* is a NULL pointer, the `wcrtomb` function is equivalent to the call:

```
wcrtomb (buf, L'\0', ps)
```

where *buf* is an internal buffer.

If *s* is not a NULL pointer, the `wcrtomb` function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character specified by *wc* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most `MB_CUR_MAX` bytes are stored.

If *wc* is a null wide character, a null byte is stored preceded by any shift sequence needed to restore the initial shift state. The resulting state described is the initial conversion state.

Return Values

n

The number of bytes stored in the resulting array, including any shift sequences to represent the multibyte character.

-1

Indicates an encoding error. The *wc* argument is not a valid wide character. The global `errno` is set to `EILSEQ`; the conversion state is undefined.

wcscat

wcscat — Concatenates two wide-character strings.

Format

```
#include <wchar.h>
wchar_t *wcscat (wchar_t *wstr_1, const wchar_t *wstr_2);
```

Function Variants

The `wcscat` function has variants named `_wcscat32` and `_wcscat64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

wstr_1, wstr_2

Pointers to null-terminated wide-character strings.

Description

The `wcscat` function appends the wide-character string *wstr_2*, including the terminating null character, to the end of *wstr_1*.

See also `wcsncat`.

Return Value

x

The first argument, *wstr_1*, which is assumed to be large enough to hold the concatenated result.

Example

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

```
#include <string.h>

/* This program concatenates two wide-character strings using */
/* the wcscat function, and then manually compares the result */
/* to the expected result                                     */

#define S1LENGTH 10
#define S2LENGTH 8

main()
{
    int i;
    wchar_t s1buf[S1LENGTH + S2LENGTH];
    wchar_t s2buf[S2LENGTH];
    wchar_t test1[S1LENGTH + S2LENGTH];

    /* Initialize the three wide-character strings */


    if (mbstowcs(s1buf, "abcmnexyz", S1LENGTH) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }


    if (mbstowcs(s2buf, " orthis", S2LENGTH) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

    if (mbstowcs(test1, "abcmnexyz orthis", S1LENGTH + S2LENGTH)

        == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

/* Concatenate s1buf with s2buf, placing the result    */
/* into * s1buf.  Then compare s1buf with the expected */
/* result in test1.                                    */

    wcscat(s1buf, s2buf);

    for (i = 0; i < S1LENGTH + S2LENGTH - 2; i++) {
        /* Check that each character is correct */
        if (test1[i] != s1buf[i]) {
            printf("Error in wcscat\n");
            exit(EXIT_FAILURE);
        }
    }

    printf("Concatenated string: <%S>\n", s1buf);
}
```

Running the example produces the following result:

```
Concatenated string: <abcmnexyz orthis>
```

wcschr

wcschr — Scans for a wide character in a specified wide-character string.

Format

```
#include <wchar.h>
wchar_t *wcschr (const wchar_t *wstr, wchar_t wc);
```

Function Variants

The `wcschr` function has variants named `_wcschr32` and `_wcschr64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

wstr

A pointer to a null-terminated wide-character string.

wc

A character of type `wchar_t`.

Description

The `wcschr` function returns the address of the first occurrence of a specified wide character in a null-terminated wide-character string. The terminating null character is considered to be part of the string.

See also `wcsrchr`.

Return Values

x

The address of the first occurrence of the specified wide character.

NULL

Indicates that the wide character does not occur in the string.

Example

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

#define BUFF_SIZE 50

main()
```

```
{
    int i;
    wchar_t s1buf[BUFF_SIZE];
    wchar_t *status;

    /* Initialize the buffer */

    if (mbstowcs(s1buf, "abcdefghijkl lkjihgfedcba", BUFF_SIZE)

        == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

 /* This program checks the wcschr function by incrementally */
 /* going through a string that ascends to the middle and    */
 /* then descends towards the end.                           */

    for (i = 0; (s1buf[i] != '\0') && (s1buf[i] != ' '); i++) {
        status = wcschr(s1buf, s1buf[i]);
        /* Check for pointer to leftmost character - test 1. */
        if (status != &s1buf[i]) {
            printf("Error in wcschr\n");
            exit(EXIT_FAILURE);
        }
    }

    printf("Program completed successfully\n");
}
```

When this example program is run, it produces the following result:

```
Program completed successfully
```

wcscmp

wcscmp — Compares two wide-character strings. It returns an integer that indicates if the strings are different, and how they differ.

Format

```
#include <wchar.h>
int wcscmp (const wchar_t *wstr_1, const wchar_t *wstr_2);
```

Arguments

wstr_1, wstr_2

Pointers to null-terminated wide-character strings.

Description

The `wcscmp` function compares the wide characters in *wstr_1* with those in *wstr_2*. If the characters differ, the function returns:

- An integer less than 0, if the codepoint of the first differing character in *wstr_1* is less than the codepoint of the corresponding character in *wstr_2*
- An integer greater than 0, if the codepoint of the first differing character in *wstr_1* is greater than the codepoint of the corresponding character in *wstr_2*

If the wide-characters strings are identical, the function returns 0.

Unlike the `wcscoll` function, the `wcscmp` function compares the string based on the binary value of each wide character.

See also `wcsncmp`.

Return Values

< 0

Indicates that *wstr_1* is less than *wstr_2*.

= 0

Indicates that *wstr_1* equals *wstr_2*.

> 0

Indicates that *wstr_1* is greater than *wstr_2*.

wcscoll

wcscoll — Compares two wide-character strings and returns an integer that indicates if the strings differ, and how they differ. The function uses the collating information in the LC_COLLATE category of the current locale to determine how the comparison is performed.

Format

```
#include <wchar.h>
int wcscoll (const wchar_t *ws1, const wchar_t *ws2);
```

Arguments

ws1, ws2

Pointers to wide-character strings.

Description

The `wcscoll` function, unlike `wcscmp`, compares two strings in a locale-dependent manner. Because no value is reserved for error indication, the application must check for one by setting `errno` to 0 before the function call and testing it after the call.

See also `wcsxfrm`.

Return Values

< 0

Indicates that *ws1* is less than *ws2*.

0

Indicates that the strings are equal.

> 0

Indicates that *ws1* is greater than *ws2*.

wcscpy

wcscpy — Copies the wide-character string *source*, including the terminating null character, into *dest*.

Format

```
#include <wchar.h>
wchar_t *wcscpy (wchar_t *dest, const wchar_t *source);
```

Function Variants

The `wcscpy` function has variants named `_wcscpy32` and `_wcscpy64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

Pointer to the null-terminated wide-character destination string.

source

Pointer to the null-terminated wide-character source string.

Description

The `wcscpy` function copies *source* into *dest*, and stops after copying *source*'s null character. If copying takes place between two overlapping strings, the behavior is undefined.

See also `wcsncpy`.

Return Value

x

The address of *source*.

wcscspn

wcscspn — Compares the characters in a wide-character string against a set of wide characters. The function returns the length of the initial substring that is comprised entirely of characters that are not in the set of wide characters.

Format

```
#include <wchar.h>
size_t wcscspn (const wchar_t *wstr1, const wchar_t *wstr2);
```

Arguments

wstr1

A pointer to a null-terminated wide-character string. If this is a null string, 0 is returned.

wstr2

A pointer to a null-terminated wide-character string that contains the set of wide characters for which the function will search.

Description

The `wcscspn` function scans the wide characters in the string pointed to by *wstr1* until it encounters a character found in *wstr2*. The function returns the length of the initial segment of *wstr1* that is formed by characters not found in *wstr2*.

Return Value

x

The length of the segment.

Example

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

/* This test sets up 2 strings, buffer and w_string, and */
/* then uses wcscspn() to calculate the maximum segment  */
/* of w_string, which consists entirely of characters    */
/* NOT from buffer.                                      */

#define BUFF_SIZE 20
#define STRING_SIZE 50

main()
{
    wchar_t buffer[BUFF_SIZE];
    wchar_t w_string[STRING_SIZE];
    size_t result;
```

```
    /* Initialize the buffer */


    if (mbstowcs(buffer, "abcdefg", BUFF_SIZE) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

    /* Initialize the string */

    if (mbstowcs(w_string, "jklmabcjklabcdehjklmno", STRING_SIZE)

        == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

/* Using wcscspn - work out the largest string in w_string */
/* which consists entirely of characters NOT from buffer   */

    result = wcscspn(w_string, buffer);
    printf("Longest segment NOT found in w_string is: %d", result);

}
```

Running the example program produces the following result:

```
Longest segment NOT found in w_string is: 4
```

wcsftime

wcsftime — Uses date and time information stored in a `tm` structure to create a wide-character output string. The format of the output string is controlled by a format string.

Format

```
#include <wchar.h>
size_t wcsftime (wchar_t *wcs, size_t maxsize, const char *format,
const struct tm *timeptr); (XPG4)
size_t wcsftime (wchar_t *wcs, size_t maxsize, const wchar_t *format,
const struct tm *timeptr); (ISO C)
```

Function Variants

Compiling with the _DECC_V4_SOURCE and _VMS_V6_SOURCE feature-test macros defined enables a local-time-based entry point to the `wcsftime` function that is equivalent to the behavior before OpenVMS Version 7.0.

Arguments

wcs

A pointer to the resultant wide-character string.

maxsize

The maximum number of wide characters to be stored in the resultant string.

format

A pointer to the string that controls the format of the output string. For the XPG4 interface, this argument is a pointer to a constant character string. For the ISO C interface, it is a pointer to a constant wide-character string.

timeptr

A pointer to the local time structure. The `tm` structure is defined in the `<time.h>` header file.

Description

The `wcsftime` function uses data in the structure pointed to by *timeptr* to create the wide-character string pointed to by *wcs*. A maximum of *maxsize* wide characters is copied to *wcs*.

The format string consists of zero or more conversion specifications and ordinary characters. All ordinary characters (including the terminating null character) are copied unchanged into the output string. A conversion specification defines how data in the `tm` structure is formatted in the output string.

A conversion specification consists of a percent (%) character followed by one or more optional characters (see Table 45), and ending with a conversion specifier (see Table 46). If any of the optional characters listed in Table 45 are specified, they must appear in the order shown in the table.

Table 45. Optional Elements of `wcsftime` Conversion Specifications

Element	Meaning
–	Optional with the field width to specify that the field is left-justified and padded with spaces. This cannot be used with the 0 element.
0	Optional with the field width to specify that the field is right-justified and padded with zeros. This cannot be used with the – element.
field width	A decimal integer that specifies the maximum field width.
.precision	A decimal integer that specifies the precision of data in a field. For the `d, H, I, j, m, M, o, S, U, w, W, y,` and `Y` conversion specifiers, the precision specifier is the minimum number of digits to appear in the field. If the conversion specification has fewer digits than that specified by the precision, leading zeros are added. For the `a, A, b, B, c, D, E, h, n, N, p, r, t, T, x, X, Z,` and `%` conversion specifiers, the precision specifier is the maximum number of wide characters to appear in the field. If the conversion specification has more characters than that specified by the precision, characters are truncated on the right. The default precision for the `d, H, I, m, M, o, S, U, w, W, y,` and `Y` conversion specifiers is 2, and the default precision for the j conversion specifier is 3.

Note that the list of optional elements of conversion specifications from Table 45 are VSI extensions to the XPG4 specification.

Table 46 lists the conversion specifiers. The `wcsftime` function uses fields in the LC_TIME category of the program's current locale to provide a value. For example, if `%B` is specified, the function accesses the *mon* field in LC_TIME to find the full month name for the month specified in the `tm` structure. The result of using invalid conversion specifiers is undefined.

Table 46. `wcsftime` Conversion Specifiers

Specifier	Replaced by
`%a`	The locale's abbreviated weekday name.
`%A`	The locale's full weekday name.
`%b`	The locale's abbreviated month name.
`%B`	The locale's full month name.
`%c`	The locale's appropriate date and time representation.
`%C`	The century number (the year divided by 100 and truncated to an integer) as a decimal number (00 – 99).
`%d`	The day of the month as a decimal number (01 – 31).
`%D`	Same as `%m/%d/%y`.
`%e`	The day of the month as a decimal number (1 – 31) in a 2-digit field with the leading space character fill.
`%Ec`	The locale's alternative date and time representation.
`%EC`	The name of the base year (period) in the locale's alternative representation.
`%Ex`	The locale's alternative date representation.
`%Ey`	The offset from the base year (`%EC`) in the locale's alternative representation.
`%EY`	The locale's full alternative year representation.
`%F`	Same as "`%Y#%m#%d`" (the ISO 8601 date format). [tm_year, tm_mon, tm_mday]
`%g`	The last 2 digits of the week-based year as a decimal number (00–99). [tm_year, tm_wday, tm_yday]
`%G`	The week-based year as a decimal number (for example, 1997). [tm_year, tm_wday, tm_yday]
`%h`	Same as `%b`.
`%H`	The hour (24-hour clock) as a decimal number (00 – 23).
`%I`	The hour (12-hour clock) as a decimal number (01 – 12).
`%j`	The day of the year as a decimal number (001 – 366).
`%k`	The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank.
`%l`	The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank.
`%m`	The month as a decimal number (01 – 12).
`%M`	The minute as a decimal number (00 – 59).
`%n`	The new-line character.
`%Od`	The day of the month using the locale's alternative numeric symbols.
`%Oe`	The date of the month using the locale's alternative numeric symbols.
`%OH`	The hour (24-hour clock) using the locale's alternative numeric symbols.

Specifier	Replaced by
%OI	The hour (12-hour clock) using the locale's alternative numeric symbols.
%Om	The month using the locale's alternative numeric symbols.
%OM	The minutes using the locale's alternative numeric symbols.
%OS	The seconds using the locale's alternative numeric symbols.
%Ou	The weekday as a number in the locale's alternative representation (Monday=1).
%OU	The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.
%OV	The week number of the year (Monday as the first day of the week) as a decimal number (01–53) using the locale's alternative numeric symbols. If the week containing January 1 has four or more days in the new year, it is considered as week 1. Otherwise, it is considered as week 53 of the previous year, and the next week is week 1.
%Ow	The weekday as a number (Sunday=0) using the locale's alternative numeric symbols.
%OW	The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%Oy	The year without the century using the locale's alternative numeric symbols.
%p	The locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%P	Same as %p but in lowercase: "am" or "pm" or a corresponding string for the current locale.
%r	The time in AM/PM notation.
%R	The time in 24-hour notation (%H:%M).
%s	The number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).
%S	The second as a decimal number (00 – 61).
%t	The tab character.
%T	The time (%H:%M:%S).
%u	The weekday as a decimal number between 1 and 7 (Monday=1).
%U	The week number of the year (the first Sunday as the first day of week 1) as a decimal number (00 – 53).
%V	The week number of the year (Monday as the first day of the week) as a decimal number (00 – 53). If the week containing January 1 has four or more days in the new year, it is considered as week 1. Otherwise, it is considered as week 53 of the previous year, and the next week is week 1.
%w	The weekday as a decimal number (0 [Sunday] – 6).
%W	The week number of the year (the first Monday as the first day of week 1) as a decimal number (00 – 53).
%x	The locale's appropriate date representation
%X	The locale's appropriate time representation
%y	The year without century as a decimal number (00– 99).
%Y	The year with century as a decimal number.
%z	The +hhmm or -hhmm numeric timezone (that is, the hour and minute offset from UTC).

Specifier	Replaced by
%Z	The time-zone name or abbreviation. If time-zone information is not available, no character is output.
%+	The date and time in date format.
%%	Literal % character.

Return Values

x

The number of wide characters placed into the array pointed to by *wcs*, not including the terminating null character.

0

Indicates an error occurred. The contents of the array are indeterminate.

Example

```
 /* Exercise the wcsftime formatting routine.             */
 /* NOTE: the format string is an "L" (or wide character) */
 /*       string indicating that this call is NOT in      */
 /*       the XPG4 format, but rather in ISO C format.    */

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <wchar.h>
#include <locale.h>
#include <errno.h>

#define NUM_OF_DATES  7
#define BUF_SIZE 256

 /* This program formats a number of different dates, once using the   */
 /* C locale and then using the fr_FR.ISO8859-1 locale.  Date and time */
 /* formatting is done using wcsftime().                               */

main()

{
  int count,
      i;
  wchar_t buffer[BUF_SIZE];
  struct tm *tm_ptr;
  time_t time_list[NUM_OF_DATES] =
  {500, 68200000, 694223999,
   694224000, 704900000, 705000000,
   705900000};

  /* Display dates using the C locale */
  printf("\nUsing the C locale:\n\n");

  setlocale(LC_ALL, "C");

  for (i = 0; i < NUM_OF_DATES; i++) {
```

```
      /* Convert to a tm structure */
      tm_ptr = localtime(&time_list[i]);

      /* Format the date and time */
      count = wcsftime(buffer, BUF_SIZE, L"Date: %A %d %B %Y%nTime: %T%n
%n",
                       tm_ptr);
      if (count == 0) {
          perror("wcsftime");
          exit(EXIT_FAILURE);
      }

      /* Print the result */
      printf("%S", buffer);
  }

  /* Display dates using the fr_FR.ISO8859-1 locale */
  printf("\nUsing the fr_FR.ISO8859-1 locale:\n\n");

  setlocale(LC_ALL, "fr_FR.ISO8859-1");

  for (i = 0; i < NUM_OF_DATES; i++) {
      /* Convert to a tm structure */
      tm_ptr = localtime(&time_list[i]);

      /* Format the date and time */
      count = wcsftime(buffer, BUF_SIZE, L"Date: %A %d %B %Y%nTime: %T%n
%n",
                       tm_ptr);
      if (count == 0) {
          perror("wcsftime");
          exit(EXIT_FAILURE);
      }

      /* Print the result */
      printf("%S", buffer);
  }
}
```

Running the example program produces the following result:

```
Using the C locale:

Date: Thursday 01 January 1970
Time: 00:08:20

Date: Tuesday 29 February 1972
Time: 08:26:40

Date: Tuesday 31 December 1991
Time: 23:59:59

Date: Wednesday 01 January 1992
Time: 00:00:00

Date: Sunday 03 May 1992
Time: 13:33:20
```

```
Date: Monday 04 May 1992
Time: 17:20:00

Date: Friday 15 May 1992
Time: 03:20:00


Using the fr_FR.ISO8859-1 locale:

Date: jeudi 01 janvier 1970
Time: 00:08:20

Date: mardi 29 février 1972
Time: 08:26:40

Date: mardi 31 décembre 1991
Time: 23:59:59

Date: mercredi 01 janvier 1992
Time: 00:00:00

Date: dimanche 03 mai 1992
Time: 13:33:20

Date: lundi 04 mai 1992
Time: 17:20:00

Date: vendredi 15 mai 1992
Time: 03:20:00
```

wcslen

wcslen — Returns the number of wide characters in a wide-character string. The returned length does not include the terminating null character.

Format

```
#include <wchar.h>
size_t wcslen (const wchar_t *wstr);
```

Argument

wstr

A pointer to a null-terminated wide-character string.

Return Value

x

The length of the wide-character string, excluding the terminating null wide character.

wcsncat

wcsncat — Concatenates a counted number of wide-characters from one string to another.

Format

```
#include <wchar.h>
wchar_t *wcsncat (wchar_t *wstr_1, const wchar_t *wstr_2, size_t maxchar);
```

Function Variants

The `wcsncat` function has variants named `_wcsncat32` and `_wcsncat64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

wstr_1, wstr_2

Pointers to null-terminated wide-character strings.

maxchar

The maximum number of wide characters from *wstr_2* that are copied to *wstr_1*. If *maxchar* is 0, no characters are copied from *wstr_2*.

Description

The `wcsncat` function appends wide characters from the wide-character string *wstr_2* to the end of *wstr_1*, up to a maximum of *maxchar* characters. A terminating null wide character is always appended to the result of the `wcsncat` function. Therefore, the maximum number of wide characters that can end up in *wstr_1* is `wcslen`(*wstr_1*) + *maxchar* + 1).

See also `wcscat`.

Return Value

x

The first argument, *wstr_1*, which is assumed to be large enough to hold the concatenated result.

Example

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

/* This program concatenates two wide-character strings using   */
/* the wcsncat function, and then manually compares the result  */
/* to the expected result                                       */

#define S1LENGTH 10
#define S2LENGTH 8
#define SIZE     3
```

```
main()
{
    int i;
    wchar_t s1buf[S1LENGTH + S2LENGTH];
    wchar_t s2buf[S2LENGTH];
    wchar_t test1[S1LENGTH + S2LENGTH];

    /* Initialize the three wide-character strings */


    if (mbstowcs(s1buf, "abcmnexyz", S1LENGTH) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }


    if (mbstowcs(s2buf, " orthis", S2LENGTH) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

    if (mbstowcs(test1, "abcmnexyz orthis", S1LENGTH + SIZE)

        == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

/* Concatenate s1buf with SIZE characters from s2buf, */
/* placing the result into s1buf. Then compare s1buf  */
/* with the expected result in test1.                 */

    wcsncat(s1buf, s2buf, SIZE);

    for (i = 0; i <= S1LENGTH + SIZE - 2; i++) {
        /* Check that each character is correct */
        if (test1[i] != s1buf[i]) {
            printf("Error in wcsncat\n");
            exit(EXIT_FAILURE);
        }

    }

    printf("Concatenated string: <%S>\n", s1buf);
}
```

Running the example produces the following result:

```
Concatenated string: <abcmnexyz or>
```

wcsncmp

wcsncmp — Compares not more than *maxchar* characters of two wide-character strings. It returns an integer that indicates if the strings are different, and how they differ.

Format

```
#include <wchar.h>
int wcsncmp (const wchar_t *wstr_1, const wchar_t *wstr_2, size_t maxchar);
```

Arguments

wstr_1, wstr_2

Pointers to null-terminated wide-character strings.

maxchar

The maximum number of characters to search in both *wstr_1* and *wstr_2*.If *maxchar* is 0, no comparison is performed and 0 is returned (the strings are considered equal).

Description

The strings are compared until a null character is encountered, the strings differ, or *maxchar* is reached. If characters differ, `wcsncmp` returns:

- An integer less than 0 if the codepoint of the first differing character in *wstr_1* is less than the codepoint of the corresponding character in *wstr_2*
- An integer greater than 0 if the codepoint of the first differing character in *wstr_1* is greater than the codepoint of the corresponding character in *wstr_2*

If no differences are found after comparing *maxchar* characters, the function returns 0.

See also `wcscmp`.

Return Values

< 0

Indicates that *wstr_1* is less than *wstr_2*.

0

Indicates that *wstr_1* equals *wstr_2*.

> 0

Indicates that *wstr_1* is greater than *wstr_2*.

wcsncpy

wcsncpy — Copies wide characters from *source* into *dest*. The function copies up to a maximum of *maxchar* characters.

Format

```
#include <wchar.h>
wchar_t *wcsncpy (wchar_t *dest, const wchar_t *source, size_t maxchar);
```

Function Variants

The `wcsncpy` function has variants named `_wcsncpy32` and `_wcsncpy64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

Pointer to the null-terminated wide-character destination string.

source

Pointer to the null-terminated wide-character source string.

maxchar

The maximum number of wide characters to copy from *source* to *dest*.

Description

The `wcsncpy` function copies no more than *maxchar* characters from *source* to *dest*. If *source* contains less than *maxchar* characters, null characters are added to *dest* until *maxchar* characters have been written to *dest*.

If *source* contains *maxchar* or more characters, as many characters as possible are copied to *dest*. The null terminator of *source* is not copied to *dest*.

See also `wcscpy`.

Return Value

x

The address of *dest*.

wcspbrk

wcspbrk — Searches a wide-character string for the first occurrence of one of a specified set of wide characters.

Format

```
#include <wchar.h>
wchar_t *wcspbrk (const wchar_t *wstr, const wchar_t *charset);
```

Function Variants

The `wcspbrk` function has variants named `_wcspbrk32` and `_wcspbrk64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

wstr

A pointer to a wide-character string. If this is a null string, NULL is returned.

charset

A pointer to a wide-character string containing the set of wide characters for which the function will search.

Description

The `wcspbrk` function scans the wide characters in the string, stops when it encounters a wide character found in *charset*, and returns the address of the first character in the string that appears in the character set.

Return Values

x

The address of the first wide character in the string that is in the set.

NULL

Indicates that none of the characters are in *charset*.

wcsrchr

wcsrchr — Scans for the last occurrence of a wide character in a given string.

Format

```
#include <wchar.h>
wchar_t *wcsrchr (const wchar_t *wstr, wchar_t wc);
```

Function Variants

The `wcsrchr` function has variants named `_wcsrchr32` and `_wcsrchr64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

wstr

A pointer to a null-terminated wide-character string.

wc

A character of type `wchar_t`.

Description

The `wcsrchr` function returns the address of the last occurrence of a given wide character in a null-terminated wide-character string. The terminating null character is considered to be part of the string.

See also `wcschr`.

Return Values

x

The address of the last occurrence of the specified wide character.

NULL

Indicates that the wide character does not occur in the string.

Example

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

#define BUFF_SIZE 50
#define STRING_SIZE 6

main()
{
    int i;
    wchar_t s1buf[BUFF_SIZE],
            w_string[STRING_SIZE];
    wchar_t *status;
    wchar_t *pbuf = s1buf;

    /* Initialize the buffer */

    if (mbstowcs(s1buf, "hijklabcdefg ytuhijklfedcba", BUFF_SIZE)

        == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

    /* Initialize the string to be searched for */


    if (mbstowcs(w_string, "hijkl", STRING_SIZE) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
   }

/* This program checks the wcsrchr function by searching for */
/* the last occurrence of a string in the buffer s1buf and   */
```

```
/* prints out the contents of s1buff from the location of
/* the string found.                                        */

    status = wcsrchr(s1buf, w_string[0]);
/* Check for pointer to start of rightmost character string. */
    if (status == pbuf) {
        printf("Error in wcsrchr\n");
        exit(EXIT_FAILURE);
    }

    printf("Program completed successfully\n");
    printf("String found : [%S]\n", status);

}
```

Running the example produces the following result:

```
Program completed successfully
String found : [hijklfedcba]
```

wcsrtombs

wcsrtombs — Converts a sequence of wide characters into a sequence of corresponding multibyte characters.

Format

```
#include <wchar.h>
size_t wcsrtombs (char *dst, const wchar_t **src, size_t len,
mbstate_t *ps);
```

Function Variants

The `wcsrtombs` function has variants named `_wcsrtombs32` and `_wcsrtombs64` for use with32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dst

A pointer to the destination array for converted multibyte character sequence.

src

An address of the pointer to an array containing the sequence of wide characters to be converted.

len

The maximum number of bytes that can be stored in the array pointed to by *dst*.

ps

A pointer to the `mbstate_t` object. If a NULL pointer is specified, the function uses its internal `mbstate_t` object. `mbstate_t` is an opaque datatype intended to keep the conversion state for the state-dependent codesets.

Description

The `wcsrtombs` function converts a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters, beginning in the conversion state described by the object pointed to by *ps*.

If *dst* is a not a NULL pointer, the converted characters are then stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null wide character, which is also stored.

Conversion stops earlier in two cases:

- When a code is reached that does not correspond to a valid multibyte character
- If *dst* is not a NULL pointer, when the next multibyte character would exceed the limit of *len* total bytes to be stored into the array pointed to by *dst*

Each conversion takes place as if by a call to the `wcrtomb` function.

If *dst* is not a NULL pointer, the pointer object pointed to by *src* is assigned either a NULL pointer (if the conversion stopped because it reached a terminating null wide character) or the address just beyond the last wide character converted (if any). If conversion stopped because it reached a terminating null wide character, the resulting state described is the initial conversion state.

If the `wcsrtombs` function is called as a counting function, which means that *dst* is a NULL pointer, the value of the internal `mbstate_t` object will remain unchanged.

See also `wcrtomb`.

Return Values

x

The number of bytes stored in the resulting array, not including the terminating null (if any).

-1

Indicates an encoding error – a character that does not correspond to a valid multibyte character was encountered; `errno` is set to `EILSEQ`; the conversion state is undefined.

wcsspn

wcsspn — Compares the characters in a wide-character string against a set of wide characters. The function returns the length of the first substring comprised entirely of characters in the set of wide characters.

Format

```
#include <wchar.h>
size_t wcsspn (const wchar_t *wstr1, const wchar_t *wstr2);
```

Arguments

wstr1

A pointer to a null-terminated wide-character string. If this string is a null string, 0 is returned.

wstr2

A pointer to a null-terminated wide-character string that contains the set of wide characters for which the function will search.

Description

The wcsspn function scans the wide characters in the wide-character string pointed to by *wstr1* until it encounters a character not found in *wstr2*. The function returns the length of the first segment of *wstr1* formed by characters found in *wstr2*.

Return Value

x

The length of the segment.

Example

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

/* This test sets up 2 strings, buffer and w_string. It */
/* then uses wcsspn() to calculate the maximum segment  */
/* of w_string that consists entirely of characters     */
/* from buffer.                                         */

#define BUFF_SIZE 20
#define STRING_SIZE 50

main()
{
    wchar_t buffer[BUFF_SIZE];
    wchar_t w_string[STRING_SIZE];
    size_t result;

    /* Initialize the buffer */


    if (mbstowcs(buffer, "abcdefg", BUFF_SIZE) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

    /* Initialize the string */

    if (mbstowcs(w_string, "abcedjklmabcjklabcdehjkl", STRING_SIZE)

        == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }
```

```
/* Using wcsspn - work out the largest string in w_string */
/* that consists entirely of characters from buffer       */

    result = wcsspn(w_string, buffer);
    printf("Longest segment found in w_string is: %d", result);

}
```

Running the example program produces the following result:

```
Longest segment found in w_string is: 5
```

wcsstr

wcsstr — Locates the first occurrence in the string pointed to by *s1* of the sequence of wide characters in the string pointed to by *s2*.

Format

```
#include <wchar.h>
wchar_t *wcsstr (const wchar_t *s1, const wchar_t *s2);
```

Function Variants

The `wcsstr` function has variants named `_wcsstr32` and `_wcsstr64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s1, s2

Pointers to null-terminated, wide-character strings.

Description

If *s2* points to a wide-character string of 0 length, the `wcsstr` function returns *s1*.

Return Values

x

A pointer to the located string.

NULL

Indicates an error; the string was not found.

wcstod

wcstod — Converts a given wide-character string to a double-precision number.

Format

```
#include <wchar.h>
double wcstod (const wchar_t *nptr, wchar_t **endptr);
```

Arguments

nptr

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

endptr

The address of an object where the function can store the address of the first unrecognized wide character that terminates the scan. If *endptr* is a NULL pointer, the address of the first unrecognized wide character is not retained.

Description

The `wcstod` function recognizes an optional sequence of white-space characters (as defined by `iswspace`), then an optional plus or minus sign, then a sequence of digits optionally containing a radix character, then an optional letter (e or E) followed by an optionally signed integer. The first unrecognized character ends the conversion.

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

The radix character is defined in the program's current locale (category LC_NUMERIC).

This function returns the converted value. For `wcstod`, overflows are accounted for in the following manner:

- 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.
- If the correct value causes an underflow, 0 is returned and `errno` is set to ERANGE.

If the string starts with an unrecognized wide character, **endptr* is set to *nptr* and a 0 value is returned.

Return Values

x

The converted string.

0

Indicates the conversion could not be performed. The function sets `errno` to one of:

- EINVAL – No conversion could be performed.
- ERANGE – The value would cause an underflow.
- ENOMEM – Not enough memory available for internal conversion buffer.

±HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

wcstof

wcstof — Converts a wide-character string to a `float`.

Format

```
#include <wchar.h>
float wcstof (const wchar_t * restrict nptr,
wchar_t ** restrict endptr);
```

Function Variants

The `wcstof` function has variants named `_wcstof32` and `_wcstof64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the wide-character string to be converted.

endptr

If this argument is *not* NULL, the function stores in it a pointer to the character that follows the last character used in the conversion.

Description

The `wcstof` function converts the initial portion of the wide-character string pointed to by *nptr* to `float` representation.

First, this function decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function); a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to a floating-point number, and returns the result as a value of type `float`.

The expected form of the (initial portion of the) wide string is optional leading white space, an optional plus ('+') or minus sign ('-') and then either (i) a decimal number, or (ii) a hexadecimal number, or (iii) an infinity, or (iv) a NaN (not-a-number).

Return Values

x

The converted value.

±HUGE_VALF

If the correct value is outside the range of representable values of the type, ±HUGE_VALF is returned according to the sign of the value; and `errno` is set to ERANGE.

n

If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number; whether `errno` acquires the value ERANGE is implementation-defined.

0

If no valid conversion could be performed.

wcstok

wcstok — Locates text tokens in a given wide-character string.

Format

```
#include <wchar.h>
wchar_t *wcstok (wchar_t *ws1, const wchar_t *ws2); (XPG4)
wchar_t *wcstok (wchar_t *ws1, const wchar_t *ws2, wchar_t **ptr); (ISO C)
```

Function Variants

The `wcstok` function has variants named `_wcstok32` and `_wcstok64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

ws1

A pointer to a wide-character string containing zero or more text tokens.

ws2

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

ptr

ISO C Standard only. Used only when *ws1* is NULL, *ptr* is a caller-provided `wchar_t` pointer into which `wcstok` stores information necessary for it to continue scanning the same wide-character string.

Description

A sequence of calls to `wcstok` breaks the wide-character string pointed to by *ws1* into a sequence of tokens, each of which is delimited by a wide character from the wide-character string pointed to by *ws2*.

The `wcstok` function keeps track of its position in the wide-character string between calls and, as successive calls are made, the function works through the wide-character string, identifying the text token following the one identified by the previous call.

Tokens in *ws1* are delimited by null characters that `wcstok` inserts into *ws1*. Therefore, *ws1* cannot be a `const` object.

The following sections describe differences between the XPG4 Standard and ISO C Standard interface to `wcstok`.

XPG4 Standard Behavior

The first call to the `wcstok` function searches the wide-character string for the first character that is *not* found in the separator string pointed to by *ws2*. The first call returns a pointer to the first wide character in the first token and writes a null wide character into *ws1* immediately following the returned token.

Subsequent calls to `wcstok` search for a wide character that *is* in the separator string pointed to by *ws2*. Each subsequent call (with the value of the first argument remaining NULL) returns a pointer to the next token in the string originally pointed to by *ws1*. When no tokens remain in the string, `wcstok` returns a NULL pointer.

ISO C Standard Behavior

For the first call in the sequence, *ws1* points to a wide-character string. In subsequent calls for the same string, *ws1* is NULL. When *ws1* is NULL, the value pointed to by *ptr* matches that stored by the previous call for the same wide-character string. Otherwise, the value pointed to by *ptr* is ignored.

The first call in the sequence searches the wide-character string pointed to by *ws1* for the first wide character that is *not* contained in the current separator wide-character string pointed to by *ws2*. If no such wide character is found, then there are no tokens in the wide-character string pointed to by *ws1*, and `wcstok` returns a NULL pointer.

The `wcstok` function then searches from there for a wide character that *is* contained in the current separator wide-character string. If no such wide character is found, the current token extends to the end of the wide-character string pointed to by *ws1*, and subsequent searches in the same wide-character string for a token return a NULL pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

In all cases, `wcstok` stores sufficient information in the pointer pointed to by *ptr* so that subsequent calls with a NULL pointer for *ws1* and the unmodified pointer value for *ptr* start searching just past the element overwritten by a null wide character (if any).

Return Values

x

A pointer to the first character of a token.

NULL

Indicates that no token was found.

Examples

1.
```
/* XPG4 version of wcstok call */

#include <wchar.h>
#include <string.h>
#include <stdio.h>

main()
```

```
{
    wchar_t str[] = L"...ab..cd,,ef.hi";

    printf("|%S|\n", wcstok(str, L"."));
    printf("|%S|\n", wcstok(NULL, L","));
    printf("|%S|\n", wcstok(NULL, L",."));
    printf("|%S|\n", wcstok(NULL, L",."));
}
```

2.
```
/* ISO C version of wcstok call */

#include <wchar.h>
#include <string.h>
#include <stdio.h>

main()
{
    wchar_t str[] = L"...ab..cd,,ef.hi";
    wchar_t *savptr = NULL;

    printf("|%S|\n", wcstok(str, L".", &savptr));
    printf("|%S|\n", wcstok(NULL, L",", &savptr));
    printf("|%S|\n", wcstok(NULL, L",.", &savptr));
    printf("|%S|\n", wcstok(NULL, L",.", &savptr));
}
```

Running this example produces the following results:

```
$ $ RUN WCSTOK_EXAMPLE
|ab|
|.cd|
|ef|
|hi|
$
```

wcstol

wcstol — Converts a wide-character string in a specified base to a `long` integer value.

Format

```
#include <wchar.h>
long long int wcstol (const wchar_t *nptr, wchar_t **endptr, int base);
```

Function Variants

The `wcstol` function has variants named `_wcstol32` and `_wcstol64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the wide-character string to be converted to a `long` integer.

endptr

The address of an object where the function can store a pointer to the first unrecognized character encountered in the conversion process (the character that follows the last character processed in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base

The value, 2 through 36, to use as the base for the conversion.

If *base* is 16, leading zeros after the optional sign are ignored, and 0x or 0X is ignored.

If *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.
- Any other combination of leading characters indicates decimal conversion.

Description

The `wcstol` function recognizes strings in various formats, depending on the value of the base. This function ignores any leading white-space characters (as defined by the `iswspace` function) in the given string. It recognizes an optional plus or minus sign, then a sequence of digits or letters that can represent an integer constant according to the value of the base. The first unrecognized character ends the conversion.

Return Values

x

The converted value.

0

Indicates that the string starts with an unrecognized wide character or that the value for *base* is invalid. If the string starts with an unrecognized wide character, **endptr* is set to *nptr*. The function sets `errno` to EINVAL.

LONG_MAX or LONG_MIN

Indicates that the converted value would cause a positive or negative overflow, respectively. The function sets `errno` to ERANGE.

wcstold

wcstold — Converts a wide-character string to a `long double`.

Format

```
#include <wchar.h>
long double wcstold (const wchar_t * restrict nptr,
```

```
wchar_t ** restrict endptr);
```

Function Variants

The `wcstold` function has variants named `_wcstold32` and `_wcstold64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Argumants

nptr

A pointer to the wide-character string to be converted.

endptr

If this argument is *not* NULL, the function stores in it a pointer to the character that follows the last character used in the conversion.

Description

The `wcstold` function converts the initial portion of the wide-character string pointed to by *nptr* to `long double` representation.

First, this function decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function); a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to a floating-point number, and returns the result as a value of type `long double`.

The expected form of the (initial portion of the) wide string is optional leading white space, an optional plus ('+') or minus sign ('-') and then either (i) a decimal number, or (ii) a hexadecimal number, or (iii) an infinity, or (iv) a NaN (not-a-number).

Return Values

x

The converted value.

±HUGE_VALL

If the correct value is outside the range of representable values for the type, ±HUGE_VALL is returned according to the sign of the value; and `errno` is set to ERANGE.

n

If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number; whether `errno` acquires the value ERANGE is implementation-defined.

0

If no valid conversion could be performed.

wcstoll

wcstoll — Converts a wide-character string in a specified base to a `long long` integer value.

Format

```
#include <wchar.h>
long int wcstoll (const wchar_t *nptr, wchar_t **endptr, int base);
```

Function Variants

The `wcstoll` function has variants named `_wcstoll32` and `_wcstoll64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the wide-character string to be converted to a `long long` integer.

endptr

The address of an object where the function can store a pointer to the first unrecognized character encountered in the conversion process (the character that follows the last character processed in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base

The value, 2 through 36, to use as the base for the conversion.

If *base* is 16, leading zeros after the optional sign are ignored, and 0x or 0X is ignored.

If *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.
- Any other combination of leading characters indicates decimal conversion.

Description

The `wcstoll` function recognizes strings in various formats, depending on the value of the base. This function ignores any leading white-space characters (as defined by the `iswspace` function) in the given string. It recognizes an optional plus or minus sign, then a sequence of digits or letters that can represent an integer constant according to the value of the base. The first unrecognized character ends the conversion.

Return Values

x

The converted value.

0

Indicates that the string starts with an unrecognized wide character or that the value for *base* is invalid. If the string starts with an unrecognized wide character, **endptr* is set to *nptr*. The function sets `errno` to EINVAL.

LLONG_MAX or LLONG_MIN

Indicates that the converted value would cause a positive or negative overflow, respectively. The function sets `errno` to ERANGE.

wcstombs

wcstombs — Converts a sequence of wide-character codes to a sequence of multibyte characters.

Format

```
#include <stdlib.h>
size_t wcstombs (char *s, const wchar_t *pwcs, size_t n);
```

Arguments

s

A pointer to the array containing the resulting multibyte characters.

pwcs

A pointer to the array containing the sequence of wide-character codes.

n

The maximum number of bytes to be stored in the array pointed to by *s*.

Description

The `wcstombs` function converts a sequence of codes corresponding to multibyte characters from the array pointed to by *pwcs* to a sequence of multibyte characters that are stored into the array pointed to by *s*, up to a maximum of *n* bytes. The value returned is equal to the number of characters converted or a -1 if an error occurred.

This function is affected by the LC_CTYPE category of the program's current locale.

If *s* is NULL, this function call is a counting operation and *n* is ignored.

See also `wctomb`.

Return Values

x

The number of bytes stored in *s*, not including the null terminating byte. If *s* is NULL, `wcstombs` returns the number of bytes required for the multibyte character array.

(size_t)-1

Indicates an error occurred. The function sets `errno` to EILSEQ – invalid character sequence, or a wide-character code does not correspond to a valid character.

wcstoul

wcstoul — Converts the initial portion of the wide-character string pointed to by *nptr* to an `unsigned long` integer.

Format

```
#include <wchar.h>
unsigned long int wcstoul (const wchar_t *nptr, wchar_t **endptr,
int base);
```

Function Variants

The `wcstoul` function has variants named `_wcstoul32` and `_wcstoul64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the wide-character string to be converted to an `unsigned long`.

endptr

The address of an object where the function can store the address of the first unrecognized character encountered in the conversion process (the character that follows the last character in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base

The value, 2 through 36, to use as the base for the conversion.

If *base* is 16, leading zeros after the optional sign are ignored, and 0x or 0X is ignored.

If *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 `wcstoul` function recognizes strings in various formats, depending on the value of the base. It ignores any leading white-space characters (as defined by the `iswspace` function) in the 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 wide character ends the conversion.

Return Values

x

The converted value.

0

Indicates that the string starts with an unrecognized wide character or that the value for *base* is invalid. If the string starts with an unrecognized wide character, **endptr* is set to *nptr*. The function sets `errno` to EINVAL.

ULONG_MAX

Indicates that the converted value would cause an overflow. The function sets `errno` to ERANGE.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <wchar.h>
#include <errno.h>
#include <limits.h>

/* This test calls wcstoul() to convert a string to an     */
/* unsigned long integer. wcstoul outputs the resulting    */
/* integer and any characters that could not be converted. */

#define MAX_STRING 128

main()
{

    int base = 10,
        errno;
    char *input_string = "1234.56";
    wchar_t string_array[MAX_STRING],
           *ptr;
    size_t size;
    unsigned long int val;
    printf("base = [%d]\n", base);
    printf("String to convert = %s\n", input_string);
    if ((size = mbstowcs(string_array, input_string, MAX_STRING)) ==

        (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }
    printf("wchar_t string is = [%S]\n", string_array);

    errno = 0;
    val = wcstoul(string_array, &ptr, base);
    if (errno == 0) {
printf("returned unsigned long int from wcstoul = [%u]\n", val);
      printf("wide char terminating scan(ptr) = [%S]\n\n", ptr);
    }
```

```
    if (errno == ERANGE) {
        perror("error value is :");
        printf("ULONG_MAX = [%u]\n", ULONG_MAX);
        printf("wcstoul failed, val = [%d]\n\n", val);
    }

}
```

Running the example program produces the following result:

```
base = [10]
String to convert = 1234.56
wchar_t string is = [1234.56]
returned unsigned long int from wcstoul = [1234]
wide char terminating scan(ptr) = [.56]
```

wcstoull

wcstoull — Converts the initial portion of the wide-character string pointed to by *nptr* to an `unsigned long long` integer.

Format

```
#include <wchar.h>
unsigned long long int wcstoull (const wchar_t *nptr, wchar_t **endptr,
int base);
```

Function Variants

The `wcstoull` function has variants named `_wcstoull32` and `_wcstoull64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

nptr

A pointer to the wide-character string to be converted to an `unsigned long long`.

endptr

The address of an object where the function can store the address of the first unrecognized character encountered in the conversion process (the character that follows the last character in the string being converted). If *endptr* is a NULL pointer, the address of the first unrecognized character is not retained.

base

The value, 2 through 36, to use as the base for the conversion.

If *base* is 16, leading zeros after the optional sign are ignored, and 0x or 0X is ignored.

If *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 `wcstoull` function recognizes strings in various formats, depending on the value of the base. It ignores any leading white-space characters (as defined by the `iswspace` function) in the 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 wide character ends the conversion.

Return Values

x

The converted value.

0

Indicates that the string starts with an unrecognized wide character or that the value for *base* is invalid. If the string starts with an unrecognized wide character, **endptr* is set to *nptr*. The function sets `errno` to EINVAL.

ULLONG_MAX

Indicates that the converted value would cause an overflow. The function sets `errno` to ERANGE.

wcswcs

wcswcs — Locates the first occurrence in the string pointed to by *wstr1* of the sequence of wide characters in the string pointed to by *wstr2*.

Format

```
#include <wchar.h>
wchar_t *wcswcs (const wchar_t *wstr1, const wchar_t *wstr2);
```

Function Variants

The `wcswcs` function has variants named `_wcswcs32` and `_wcswcs64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

wstr1, wstr2

Pointers to null-terminated wide-character strings.

Return Values

Pointer

A pointer to the located wide-character string.

NULL

Indicates that the wide-character string was not found.

Example

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

/* This test uses wcswcs() to find the occurrence of each */
/* subwide-character string, string1 and string2, within  */
/* the main wide-character string, lookin.                */

#define BUF_SIZE 50

main()
{
  static char lookin[] = "that this is a test was at the end";

  char string1[] = "this",
       string2[] = "the end";

  wchar_t buffer[BUF_SIZE],
          input_buffer[BUF_SIZE];

  /* Convert lookin to wide-character format.         */
  /* Buffer and print it out.                         */


  if (mbstowcs(buffer, lookin, BUF_SIZE) == (size_t)-1) {

      perror("mbstowcs");
      exit(EXIT_FAILURE);
  }

  printf("Buffer to look in: %S\n", buffer);

  /* Convert string1 to wide-character format and use */
  /* wcswcs() to locate it within buffer              */


  if (mbstowcs(input_buffer, string1, BUF_SIZE) == (size_t)-1) {

      perror("mbstowcs");
      exit(EXIT_FAILURE);
  }

  printf("this: %S\n", wcswcs(buffer, input_buffer));

  /* Convert string2 to wide-character format and use */
  /* wcswcs() to locate it within buffer              */


  if (mbstowcs(input_buffer, string2, BUF_SIZE) == (size_t)-1) {

      perror("mbstowcs");
      exit(EXIT_FAILURE);
```

```
  }
  printf("the end: %S\n", wcswcs(buffer, input_buffer));

  exit(1);
}
```

Running this example produces the following results:

```
Buffer to look in: that this is a test was at the end
this: this is a test was at the end
the end: the end
```

wcswidth

wcswidth — Determines the number of printing positions on a display device that are required for a wide-character string.

Format

```
#include <wchar.h>
int wcswidth (const wchar_t *pwcs, size_t n);
```

Arguments

pwcs

A pointer to a wide-character string.

n

The maximum number of characters in the string.

Description

The `wcswidth` function returns the number of printing positions required to display the first *n* characters of the string pointed to by *pwcs*. If there are less than *n* wide characters in the string, the function returns the number of positions required for the whole string.

Return Values

x

The number of printing positions required.

0

If *pwcs* is a null character.

-1

Indicates that one (or more) of the wide characters in the string pointed to by *pwcs* is not a printable character.

wcsxfrm

wcsxfrm — Changes a wide-character string such that the changed string can be passed to the `wcscmp` function and produce the same result as passing the unchanged string to the `wcscoll` function.

Format

```
#include <wchar.h>
size_t wcsxfrm (wchar_t *ws1, const wchar_t *ws2, size_t maxchar);
```

Arguments

ws1, ws2

Pointers to wide-character strings.

maxchar

The maximum number of wide characters, including the null wide-character terminator, allowed to be stored in *s1*.

Description

The `wcsxfrm` function transforms the string pointed to by *ws2* and stores the resulting string in the array pointed to by *ws1*. No more than *maxchar* wide characters, including the null wide terminator, are placed into the array pointed to by *ws1*.

If the value of *maxchar* is less than the required size to store the transformed string (including the terminating null), the contents of the array pointed to by *ws1* is indeterminate. In such a case, the function returns the size of the transformed string.

If *maxchar* is 0, then, *ws1* is allowed to be a NULL pointer, and the function returns the required size of the *ws1* array before making the transformation.

The wide-character string comparison functions, `wcscoll` and `wcscmp`, can produce different results given the same two wide-character strings to compare. This is because `wcscmp` does a straightforward comparison of the code point values of the characters in the strings, whereas `wcscoll` uses the locale information to do the comparison. Depending on the locale, the `wcscoll` comparison can be a multipass operation, which is slower than `wcscmp`.

The `wcsxfrm` function transforms wide-character strings in such a way that if you pass two transformed strings to the `wcscmp` function, the result is the same as passing the two original strings to the `wcscoll` function. The `wcsxfrm` function is useful in applications that need to do a large number of comparisons on the same wide-character strings using `wcscoll`. In this case, it may be more efficient (depending on the locale) to transform the strings once using `wcsxfrm` and then use the `wcscmp` function to do comparisons.

Return Values

x

Length of the resulting string pointed to by *ws1*, not including the terminating null character.

(size_t)-1

Indicates that an error occurred. The function sets `errno` to EINVAL – The string pointed to by *ws2* contains characters outside the domain of the collating sequence.

Example

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

 /* This program verifies that two transformed strings,    */
 /* when passed through wcsxfrm and then compared, provide */
 /* the same result as if passed through wcscoll without   */
 /* any transformation.                                    */

#define  BUFF_SIZE  20

main()
{
    wchar_t w_string1[BUFF_SIZE];
    wchar_t w_string2[BUFF_SIZE];
    wchar_t w_string3[BUFF_SIZE];
    wchar_t w_string4[BUFF_SIZE];
    int errno;
    int coll_result;
    int wcscmp_result;
    size_t wcsxfrm_result1;
    size_t wcsxfrm_result2;

    /* setlocale to French locale */

    if (setlocale(LC_ALL, "fr_FR.ISO8859-1") == NULL) {
        perror("setlocale");
        exit(EXIT_FAILURE);
    }

    /* Convert each of the strings into wide-character format. */


    if (mbstowcs(w_string1, "<a`>bcd", BUFF_SIZE) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }


    if (mbstowcs(w_string2, "abcz", BUFF_SIZE) == (size_t)-1) {

        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }

    /* Collate string 1 and string 2 and store the result. */

    errno = 0;
```

```
    coll_result = wcscoll(w_string1, w_string2);
    if (errno) {
        perror("wcscoll");
        exit(EXIT_FAILURE);
    }
    else {

       /*  Transform the strings (using wcsxfrm) into  */
       /*  w_string3 and w_string4.                    */

       wcsxfrm_result1 = wcsxfrm(w_string3, w_string1, BUFF_SIZE);

       if (wcsxfrm_result1 == ((size_t) - 1))
           perror("wcsxfrm");
       else if (wcsxfrm_result1 > BUFF_SIZE) {
           perror("\n** String is too long **\n");
           exit(EXIT_FAILURE);
       }
       else {
          wcsxfrm_result2 = wcsxfrm(w_string4, w_string2, BUFF_SIZE);
          if (wcsxfrm_result2 == ((size_t) - 1)) {
              perror("wcsxfrm");
              exit(EXIT_FAILURE);
           }
          else if (wcsxfrm_result2 > BUFF_SIZE) {
              perror("\n** String is too long **\n");
              exit(EXIT_FAILURE);
           }

        /* Compare the two transformed strings and verify that  */
        /* the result is the same as the result from wcscoll on */
        /* the original strings.                                */

          else {
              wcscmp_result = wcscmp(w_string3, w_string4);
              if (wcscmp_result == 0 && (coll_result == 0)) {
                  printf("\nReturn value from wcscoll() and return value"
                                       " from wcscmp() are both zero.");
                  printf("\nThe program was successful\n\n");
              }
              else if ((wcscmp_result < 0) && (coll_result < 0)) {
                  printf("\nReturn value from wcscoll() and return value"
                                     " from wcscmp() are less than zero.");
                  printf("\nThe program was successful\n\n");
              }
              else if ((wcscmp_result > 0) && (coll_result > 0)) {
                  printf("\nReturn value from wcscoll() and return value"
                                    " from wcscmp() are greater than zero.");
                  printf("\nThe program was successful\n\n");
              }
              else {
                  printf("** Error **\n");
                  printf("\nReturn values are not of the same type");
              }
           }
       }
    }
}
```

Running the example program produces the following result:

```
Return value from wcscoll() and return value
       from wcscmp() are less than zero.
The program was successful
```

wctob

wctob — Determines if a wide character corresponds to a single-byte multibyte character and returns its multibyte character representation.

Format

```
#include <stdio.h>
#include <wchar.h>
int wctob (wint_t c);
```

Argument

c

The wide character to be converted to a single-byte multibyte character.

Description

The `wctob` function determines whether the specified wide character corresponds to a single-byte multibyte character when in the initial shift state and, if so, returns its multibyte character representation.

Return Values

x

The single-byte representation of the wide character specified.

EOF

Indicates an error. The wide character specified does not correspond to a single-byte multibyte character.

wctomb

wctomb — Converts a wide character to its multibyte character representation.

Format

```
#include <stdlib.h>
int wctomb (char *s, wchar_t wchar);
```

Arguments

s

A pointer to the resulting multibyte character.

wchar

The code for the wide character.

Description

The `wctomb` function converts the wide character specified by *wchar* to its multibyte character representation. If *s* is NULL, then 0 is returned. Otherwise, the number of bytes comprising the multibyte character is returned. At most, MB_CUR_MAX bytes are stored in the array object pointed to by *s*.

This function is affected by the LC_CTYPE category of the program's current locale.

Return Values

x

The number of bytes comprising the multibyte character corresponding to *wchar*.

0

If *s* is NULL.

-1

If *wchar* is not a valid character.

wctrans

wctrans — Returns the description of a mapping, corresponding to specified property, that can later be used in a call to `towctrans`.

Format

```
#include <wctype.h>
wctrans_t wctrans (const char *property);
```

Argument

property

The name of the mapping. The following property names are defined for all locales:

- "toupper"
- "tolower"

Additional property names may also be defined in the LC_CTYPE category of the current locale.

Description

The `wctrans` function constructs a value with type `wctrans_t` that describes a mapping between wide characters identified by the *property* argument.

See also `towctrans`.

Return Values

nonzero

According to the LC_CTYPE category of the current program locale, the string specified as a property argument is the name of an existing character mapping. The value returned can be used in a call to the `towctrans` function.

0

Indicates an error. The property argument does not identify a character mapping in the current program's locale.

wctype

wctype — Used for defining a character class. The value returned by this function is used in calls to the `iswctype` function.

Format

```
#include <wctype.h> (ISO C)
#include <wchar.h> (XPG4)
wctype_t wctype (const char *char_class);
```

Argument

char_class

A pointer to a valid character class name.

Description

The `wctype` function converts a valid character class defined for the current locale to an object of type `wctype_t`. The following character class names are defined for all locales:

```
alnum        cntrl        lower        space
alpha        digit        print        upper
blank        graph        punct        xdigit
```

Additional character class names may also be defined in the LC_CTYPE category of the current locale.

See also `iswctype`.

Return Values

x

An object of type `wctype_t` that can be used in calls to the `iswctype` function.

0

If the character class name is not valid for the current locale.

Example

```
#include <locale.h>
#include <wchar.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>


 /* This test will set up a number of character class using wctype() */
 /* and then verify whether calls to iswctype() using these classes  */
 /* produce the same results as calls to the is**** routines.        */

main()
{

    wchar_t w_char;
    wctype_t ret_val;

    char *character = "A";

    /* Convert character to wide character format - w_char */

    if (mbtowc(&w_char, character, 1) == -1) {
        perror("mbtowc");
        exit(EXIT_FAILURE);
    }

    /* Check if results from iswalnum() matches check on */
    /* alnum character class                             */

    if ((iswalnum((wint_t) w_char)) &&
        (iswctype((wint_t) w_char, wctype("alnum"))))
        printf("[%C] is a member of the character class alnum\n", w_char);
 else
    printf("[%C] is not a member of the character class alnum\n", w_char);

    /* Check if results from iswalpha() matches check on */
    /* alpha character class                             */

    if ((iswalpha((wint_t) w_char)) &&
        (iswctype((wint_t) w_char, wctype("alpha"))))
      printf("[%C] is a member of the character class alpha\n", w_char);
    else
     printf("[%C] is not a member of the character class alpha\n", w_char);

    /* Check if results from iswcntrl() matches check on */
    /* cntrl character class                             */

    if ((iswcntrl((wint_t) w_char)) &&
        (iswctype((wint_t) w_char, wctype("cntrl"))))
        printf("[%C] is a member of the character class cntrl\n", w_char);
    else
     printf("[%C] is not a member of the character class cntrl\n", w_char);

    /* Check if results from iswdigit() matches check on */
```

```
/* digit character class                              */

if ((iswdigit((wint_t) w_char)) &&
    (iswctype((wint_t) w_char, wctype("digit"))))
    printf("[%C] is a member of the character class digit\n", w_char);
else
 printf("[%C] is not a member of the character class digit\n", w_char);

/* Check if results from iswgraph() matches check on */
/* graph character class                              */

if ((iswgraph((wint_t) w_char)) &&
    (iswctype((wint_t) w_char, wctype("graph"))))
    printf("[%C] is a member of the character class graph\n", w_char);
else
 printf("[%C] is not a member of the character class graph\n", w_char);

/* Check if results from iswlower() matches check on */
/* lower character class                              */

if ((iswlower((wint_t) w_char)) &&
    (iswctype((wint_t) w_char, wctype("lower"))))
    printf("[%C] is a member of the character class lower\n", w_char);
else
 printf("[%C] is not a member of the character class lower\n", w_char);

/* Check if results from iswprint() matches check on */
/* print character class                              */

if ((iswprint((wint_t) w_char)) &&
    (iswctype((wint_t) w_char, wctype("print"))))
    printf("[%C] is a member of the character class print\n", w_char);
else
 printf("[%C] is not a member of the character class print\n", w_char);

/* Check if results from iswpunct() matches check on */
/* punct character class                              */

if ((iswpunct((wint_t) w_char)) &&
    (iswctype((wint_t) w_char, wctype("punct"))))
    printf("[%C] is a member of the character class punct\n", w_char);
else
 printf("[%C] is not a member of the character class punct\n", w_char);

/* Check if results from iswspace() matches check on */
/* space character class                              */

if ((iswspace((wint_t) w_char)) &&
    (iswctype((wint_t) w_char, wctype("space"))))
    printf("[%C] is a member of the character class space\n", w_char);
else
printf("[%C] is not a member of the character class space\n", w_char);

/* Check if results from iswupper() matches check on */
/* upper character class                              */

if ((iswupper((wint_t) w_char)) &&
    (iswctype((wint_t) w_char, wctype("upper"))))
```

```
        printf("[%C] is a member of the character class upper\n", w_char);
    else
    printf("[%C] is not a member of the character class upper\n", w_char);

    /* Check if results from iswxdigit() matches check on */
    /* xdigit character class                             */

    if ((iswxdigit((wint_t) w_char)) &&
        (iswctype((wint_t) w_char, wctype("xdigit"))))
        printf("[%C] is a member of the character class xdigit\n", w_char);
    else
     printf("[%C] is not a member of the character class xdigit\n",
 w_char);

}
```

Running this example produces the following result:

```
[A] is a member of the character class alnum
[A] is a member of the character class alpha
[A] is not a member of the character class cntrl
[A] is not a member of the character class digit
[A] is a member of the character class graph
[A] is not a member of the character class lower
[A] is a member of the character class print
[A] is not a member of the character class punct
[A] is not a member of the character class space
[A] is a member of the character class upper
[A] is a member of the character class xdigit
```

wcwidth

wcwidth — Determines the number of printing positions on a display device required for the specified wide character.

Format

```
#include <wchar.h>
int wcwidth (wchar_t wc);
```

Argument

wc

A wide character.

Description

The `wcwidth` function determines the number of column positions needed for the specified wide character *wc*. The value of *wc* must be a valid wide character in the current locale.

Return Values

x

The number of printing positions required for *wc*.

0

If *wc* is a null character.

-1

Indicates that *wc* does not represent a valid printing wide character.

wmemchr

wmemchr — Locates the first occurrence of a specified wide character in an array of wide characters.

Format

```
#include <wchar.h>
wchar_t wmemchr (const wchar_t *s, wchar_t c, size_t n);
```

Function Variants

The `wmemchr` function has variants named `_wmemchr32` and `_wmemchr64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s

A pointer to an array of wide characters to be searched.

c

The wide character value to search for.

n

The maximum number of wide characters in the array to be searched.

Description

The `wmemchr` function locates the first occurrence of the specified wide character in the initial *n* wide characters of the array pointed to by *s*.

Return Values

x

A pointer to the first occurrence of the wide character in the array.

NULL

The specified wide character does not occur in the array.

wmemcmp

wmemcmp — Compares two arrays of wide characters.

Format

```
#include <wchar.h>
int wmemcmp (const wchar_t *s1, const wchar_t *s2, size_t n);
```

Arguments

s1, s2

Pointers to wide-character arrays.

n

The maximum number of wide characters to be compared.

Description

The `wmemcmp` function compares the first *n* wide characters of the array pointed to by *s1* with the first *n* wide characters of the array pointed to by *s2*. The wide characters are compared not according to locale-dependent collation rules, but as integral objects of type `wchar_t`.

Return Values

0

Arrays are equal.

Positive value

The first array is greater than the second.

Negative value

The first array is less than the second.

wmemcpy

wmemcpy — Copies a specified number of wide characters from one wide-character array to another.

Format

```
#include <wchar.h>
wchar_t wmemcpy (wchar_t *dest, const wchar_t *source, size_t n);
```

Function Variants

The `wmemcpy` function has variants named `_wmemcpy32` and `_wmemcpy64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

A pointer to the destination array.

source

A pointer to the source array.

n

The number of wide characters to be copied.

Description

The `wmemcpy` function copies *n* wide characters from the array pointed to by *source* to the array pointed to by *dest*.

Return Value

x

The value of *dest*.

wmemmove

wmemmove — Copies a specified number of wide characters from one wide-character array to another.

Format

```
#include <wchar.h>
wchar_t wmemmove (wchar_t *dest, const wchar_t *source, size_t n);
```

Function Variants

The `wmemmove` function has variants named `_wmemmove32` and `_wmemmove64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

dest

A pointer to the destination array.

source

A pointer to the source array.

n

The number of wide characters to be moved.

Description

The `wmemmove` function copies *n* wide characters from the location pointed to by *source* to the location pointed to by *dest*.

The `wmemmove` and `wmemcpy` routines perform the same function, except that `wmemmove` ensures that the original contents of the source array are copied to the destination array even if the two arrays overlap. Where such overlap is possible, programs that require portability should use `wmemmove`, not `wmemcopy`.

Return Value

x

The value of *dest*.

wmemset

wmemset — Sets a specified value to a specified number of wide characters in an array of wide characters.

Format

```
#include <wchar.h>
wchar_t wmemset (wchar_t *s, wchar_t c, size_t n);
```

Function Variants

The `wmemset` function has variants named `_wmemset32` and `_wmemset64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

s

A pointer to the array of wide characters.

c

The value to be placed in the first *n* wide characters of the array.

n

The number of wide characters to be set to the specified value *c*.

Description

The `wmemset` function copies the value of *c* into each of the first *n* wide characters of the array pointed to by *s*.

Return Value

x

The value of *s*.

wprintf

wprintf — Performs formatted output from the standard output (`stdout`). See Chapter 2 for information on format specifiers.

Format

```
#include <wchar.h>
int wprintf (const wchar_t *format, ...);
```

Arguments

format

A pointer to a wide-character string containing the format specifications. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

. . .

Optional expressions whose resultant types correspond to conversion specifications given in the format specification.

If no conversion specifications are given, the output sources can be omitted. Otherwise, the function calls must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources.

Conversion specifications are matched to output sources in left-to-right order. Excess output pointers, if any, are ignored.

Description

The `wprintf` function is equivalent to the `fwprintf` function with the `stdout` argument interposed before the `wprintf` arguments.

Return Values

n

The number of wide characters written.

Negative value

Indicates an error. The function sets `errno` to one of the following:

- EILSEQ – Invalid character detected.
- EINVAL – Insufficient arguments.

- ENOMEM – Not enough memory available for conversion.
- ERANGE – Floating-point calculations overflow.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This might indicate that conversion to a numeric value failed because of overflow.

The function can also set `errno` to the following as a result of errors returned from the I/O subsystem:

- EBADF – The file descriptor is not valid.
- EIO – I/O error.
- ENOSPC – No free space on the device containing the file.
- ENXIO – Device does not exist.
- EPIPE – Broken pipe.
- ESPIPE – Illegal seek in a file opened for append.
- EVMSERR – Nontranslatable OpenVMS error. `vaxc$errno` contains the OpenVMS error code. This indicates that an I/O error occurred for which there is no equivalent C error code.

wrapok

wrapok — In the UNIX system environment, allows the wrapping of a word from the right border of the window to the beginning of the next line. This routine is provided only for UNIX software compatibility and serves no function in the OpenVMS environment.

Format

```
#include <curses.h>
wrapok (WINDOW *win, bool boolf);
```

Arguments

win

A pointer to the window.

boolf

A Boolean TRUE or FALSE value. If *boolf* is FALSE, scrolling is not allowed. This is the default setting. The `bool` type is defined in the `<curses.h>` header file as follows:

```
#define bool int
```

write

write — Writes a specified number of bytes from a buffer to a file.

Format

```
#include <unistd.h>
ssize_t write (int file_desc, void *buffer, size_t nbytes); (ISO POSIX-1)
int write (int file_desc, void *buffer, int nbytes); (Compatibility)
```

Arguments

file_desc

A file descriptor that refers to a file currently opened for writing or updating.

buffer

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

nbytes

The maximum number of bytes involved in the write operation.

Description

If the `write` is to an RMS record file and the buffer contains embedded new-line characters, more than one record may be written to the file. Even if there are no embedded new-line characters, if *nbytes* is greater than the maximum record size for the file, more than one record will be written to the file. The `write` function always generates at least one record.

If the `write` is to a mailbox and the third argument, *nbytes*, specifies a length of 0, an end-of-file message is written to the mailbox. This occurs for mailboxes created by the application using SYS$CREMBX, but not for mailboxes created to implement POSIX pipes. For more information, see Chapter 5.

Return Values

x

The number of bytes written.

-1

Indicates errors, including undefined file descriptors, illegal buffer addresses, and physical I/O errors.

writev

writev — Writes to a file.

Format

```
#include <uio.h>
ssize_t writev (int file_desc, const struct iovec *iov,
int iovcnt);
ssize_t __writev64 (int file_desc, const struct __iovec64 *iov,
int iovcnt);
```

Function Variants

The `writev` function has variants named `_writev32` and `__writev64` for use with 32-bit and 64-bit pointer sizes, respectively. See Section 1.9 for more information on using pointer-size-specific functions.

Arguments

file_desc

A file descriptor that refers to a file currently opened for writing or updating.

iov

Array of `iovec` structures from which the output data is gathered.

iovcnt

The number of buffers specified by the members of the *iov* array.

Description

The `writev` function is equivalent to `write` but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt -1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to {IOV_MAX}, defined in `<limits.h>`.

Each `iovec` entry specifies the base address and length of an area in memory from which data should be written. The `writev` function writes a complete area before proceeding to the next.

If *filedes* refers to a regular file and all of the `iov_len` members in the array pointed to by *iov* are 0, `writev` returns 0 and has no other effect.

For other file types, the behavior is unspecified.

If the sum of the `iov_len` values is greater than SSIZE_MAX, the operation fails and no data is transferred.

Upon successful completion, `writev` returns the number of bytes actually written. Otherwise, it returns a value of -1, the file pointer remains unchanged, and `errno` is set to indicate an error.

Return Values

x

The number of bytes written.

-1

Indicates an error. The file times do not change, and the function sets `errno` to one of the following values:

- EBADF– The *file_desc* argument is not a valid file descriptor open for writing.
- EINTR– The write operation was terminated due to the receipt of a signal, and no data was transferred.

- EINVAL– The sum of the `iov_len` values in the *iov* array would overflow an `ssize_t`, or the *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.
- EIO – A physical I/O error has occurred.
- ENOSPC– There was no free space remaining on the device containing the file.
- EPIPE– An attempt is made to write to a pipe or FIFO that is not open for reading by any process, or that only has one end open. A SIGPIPE signal will also be sent to the thread.

wscanf

wscanf — Reads input from the standard input (`stdin`) under control of the wide-character format string.

Format

```
#include <wchar.h>
int wscanf (const wchar_t *format, ...);
```

Arguments

format

A pointer to a wide-character string containing the format specifications. For more information about format and conversion specifications and their corresponding arguments, see Chapter 2.

. . .

Optional expressions whose results correspond to conversion specifications given in the format specification.

If no conversion specifications are given, you can omit the input pointers. Otherwise, the function calls must have exactly as many input pointers as there are conversion specifications, and the conversion specifications must match the types of the input pointers.

Conversion specifications are matched to input sources in left-to-right order. Excess input pointers, if any, are ignored.

Description

The `wscanf` function is equivalent to the `fwscanf` function with the `stdin` arguments interposed before the `wscanf` arguments.

Return Values

n

The number of input items assigned. The number can be less than provided for, even zero, in the event of an early matching failure.

EOF

Indicates an error. An input failure occurred before any conversion.

y0, y1, yn

y0, y1, yn — Compute Bessel functions of the second kind.

Format

```
#include <math.h>
double y0 (double x);
float y0f (float x);
long double y0l (long double x);
double y1 (double x);
float y1f (float x);
long double y1l (long double x);
double yn (int n, double x);
float ynf (int n, float x);
long double ynl (int n, long double x);
```

Arguments

x

A positive, real value.

n

An integer.

Description

The `y0` functions return the value of the Bessel function of the second kind of order 0.

The `y1` functions return the value of the Bessel function of the second kind of order 1.

The `yn` functions return the value of the Bessel function of the second kind of order *n*.

Return Values

x

The relevant Bessel value of *x* of the second kind.

-HUGE_VAL

The *x* argument is 0.0; `errno` is set to ERANGE.

NaN

The *x* argument is negative or NaN; `errno` is set to EDOM.

0

Underflow occurred; `errno` is set to ERANGE.

HUGE_VAL

Overflow occurred; `errno` is set to ERANGE.

Appendix A. Version-Dependency Tables

New functions are added to the VSI C Run-Time Library with each version of VSI C. These functions are implemented and shipped with the OpenVMS operating system, while the documentation and header files containing their prototypes are shipped with versions of the VSI C compiler.

You might have a newer version of VSI C that has header files and documentation for functions that are not supported on your older OpenVMS system. For example, if your target operating system platform is OpenVMS Version 7.2, you cannot use C RTL functions introduced on OpenVMS Version 7.3, even though they are documented in this manual.

This appendix contains several tables that list what C RTL functions are supported on recent OpenVMS versions. This is helpful for determining the functions to avoid using on your target OpenVMS platforms.

A.1. Functions Available on all OpenVMS VAX, Alpha, and Integrity servers Versions

Table A.1 lists functions available on all OpenVMS VAX and OpenVMS Alpha versions.

Table A.1. Functions Available on All OpenVMS Systems

abort	abs	access	acos
alarm	asctime	asin	assert
atan2	atan	atexit	atof
atoi	atoll (Alpha)	atol	atoq (Alpha)
box	brk	bsearch	cabs
calloc	ceil	cfree	chdir
chmod	chown	clearerr	clock
close	cosh	cos	creat
ctermid	ctime	cuserid	decc$crtl_init
decc$fix_time	decc$from_vms	decc$match_wild	decc$record_read
decc$record_write	decc$set_reentrancy	decc$to_vms	decc$translate_vms
delete	delwin	difftime	div
dup2	dup	ecvt	endwin
execle	execlp	execl	execve
execvp	execv	exit	_exit
exp	fabs	fclose	fcvt
fdopen	feof	ferror	fflush
fgetc	fgetname	fgetpos	fgets
fileno	floor	fmod	fopen
fprintf	fputc	fputs	fread
free	freopen	frexp	fscanf

fseek	fsetpos	fstat	fsync
ftell	ftime	fwait	fwrite
gcvt	getchar	getcwd	getc
getegid	getenv	geteuid	getgid
getname	getpid	getppid	gets
getuid	getw	gmtime	gsignal
hypot	initscr	isalnum	isalpha
isapipe	isascii	isatty	iscntrl
isdigit	isgraph	islower	isprint
ispunct	isspace	isupper	isxdigit
kill	labs	ldexp	ldiv
llabs (Alpha)	lldiv (Alpha)	localeconv	localtime
log10	log	longjmp	longname
lseek	lwait	malloc	mblen
mbstowcs	mbtowc	memchr	memcmp
memcpy	memmove	memset	mkdir
mktemp	mktime	modf	mvwin
mv[w]addstr	newwin	nice	open
overlay	overwrite	pause	perror
pipe	pow	printf	putchar
putc	puts	putw	qabs (Alpha)
qdiv (Alpha)	qsort	raise	rand
read	realloc	remove	rename
rewind	sbrk	scanf	scroll
setbuf	setgid	setjmp	setlocale
setuid	setvbuf	sigblock	signal
sigpause	sigstack (VAX)	sigvec	sinh
sin	sleep	sprintf	sqrt
srand	sscanf	ssignal	stat
strcat	strchr	strcmp	strcoll
strcpy	strcspn	strerror	strftime
strlen	strncat	strncmp	strncpy
strpbrk	strrchr	strspn	strstr
strtod	strtok	strtoll (Alpha)	strtol
strtoq (Alpha)	strtoull (Alpha)	strtoul	strtouq (Alpha)
strxfrm	subwin	system	tanh
tan	times	time	tmpfile
tmpnam	toascii	tolower	_tolower
touchwin	toupper	_toupper	ttyname

umask	ungetc	vaxc$calloc_opt	vaxc$cfree_opt
vaxc$crtl_init	vaxc$establish	vaxc$free_opt	vaxc$malloc_opt
vaxc$realloc_opt	va_arg	va_count	va_end
va_start	va_start_1	vfork	vfprintf
vprintf	vsprintf	wait	wcstombs
wctomb	write	[w]addch	[w]addstr
[w]clear	[w]clrattr	[w]clrtobot	[w]clrtoeol
[w]delch	[w]deleteln	[w]erase	[w]getch
[w]getstr	[w]inch	[w]insch	[w]insertln
[w]insstr	[w]move	[w]printw	[w]refresh
[w]scanw	[w]setattr	[w]standend	[w]standout

A.2. Functions Available on OpenVMS Version 6.2 and Higher

Table A.2 lists additional functions available on OpenVMS VAX and OpenVMS Alpha Version 6.2 and higher.

Table A.2. Functions Added in OpenVMS Version 6.2

catclose	catgets	catopen	fgetwc
fgetws	fputwc	fputws	getopt
getwc	getwchar	iconv	iconv_close
iconv_open	iswalnum	iswalpha	iswcntrl
iswctype	iswdigit	iswgraph	iswlower
iswprint	iswpunct	iswspace	iswupper
iswxdigit	nl_langinfo	putwc	putwchar
strnlen	strptime	towlower	towupper
ungetwc	wcscat	wcschr	wcscmp
wcscoll	wcscpy	wcscspn	wcsftime
wcslen	wcsncat	wcsncmp	wcsncpy
wcspbrk	wcsrchr	wcsspn	wcstol
wcstoul	wcswcs	wcswidth	wcsxfrm
wcstod	wctype	wcwidth	wcstok

A.3. Functions Available on OpenVMS Version 7.0 and Higher

Table A.3 lists additional functions available on OpenVMS VAX and OpenVMS Alpha Version 7.0 and higher.

Table A.3. Functions Added in OpenVMS Version 7.0

basename	bcmp	bcopy	btowc
bzero	closedir	confstr	dirname
drand48	erand48	ffs	fpathconf
ftruncate	ftw	fwide	fwprintf
fwscanf	getclock	getdtablesize	getitimer
getlogin	getpagesize	getpwnam	getpwuid
gettimeofday	index	initstate	jrand48
lcong48	lrand48	mbrlen	mbrtowc
mbsinit	mbsrtowcs	memccpy	mkstemp
mmap	mprotect	mrand48	msync
munmap	nrand48	opendir	pathconf
pclose	popen	putenv	random
readdir	rewinddir	rindex	rmdir
seed48	seekdir	setenv	setitimer
setstate	sigaction	sigaddset	sigdelset
sigemptyset	sigfillset	sigismember	siglongjmp
sigpending	sigprocmask	sigsetjmp	sigsuspend
srand48	srandom	strcasecmp	strdup
strfmon	strncasecmp	strsep	swab
swprintf	swscanf	sysconf	telldir
tempnam	towctrans	truncate	tzset
ualarm	uname	unlink	unsetenv
usleep	vfwprintf	vswprintf	vwprintf
wait3	wait4	waitpid	wcrtomb
wcsrtombs	wcsstr	wctob	wctrans
wmemchr	wmemcmp	wmemcpy	wmemmove
wmemset	wprintf	wscanf	

A.4. Functions Available on OpenVMS Alpha Version 7.0 and Higher

Table A.4 lists additional functions available on OpenVMS Alpha Version 7.0 and higher.

Table A.4. Functions Added in OpenVMS Alpha Version 7.0

_basename32	_basename64	_bsearch32	_bsearch64
_calloc32	_calloc64	_catgets32	_catgets64
_ctermid32	_ctermid64	_cuserid32	_cuserid64
_dirname32	_dirname64	_fgetname32	_fgetname64
_fgets32	_fgets64	_fgetws32	_fgetws64

_gcvt32	_gcvt64	_getcwd32	_getcwd64
_getname32	_getname64	_gets32	_gets64
_index32	_index64	_longname32	_longname64
_malloc32	_malloc64	_mbsrtowcs32	_mbsrtowcs64
_memccpy32	_memccpy64	_memchr32	_memchr64
_memcpy32	_memcpy64	_memmove32	_memmove64
_memset32	_memset64	_mktemp32	_mktemp64
_mmap32	_mmap64	_qsort32	_qsort64
_realloc32	_realloc64	_rindex32	_rindex64
_strcat32	_strcat64	_strchr32	_strchr64
_strcpy32	_strcpy64	_strdup32	_strdup64
_strncat32	_strncat64	_strncpy32	_strncpy64
_strpbrk32	_strpbrk64	_strptime32	_strptime64
_strrchr32	_strrchr64	_strsep32	_strsep64
_strstr32	_strstr64	_strtod32	_strtod64
_strtok32	_strtok64	_strtol32	_strtol64
_strtoll32	_strtoll64	_strtoq32	_strtoq64
_strtoul32	_strtoul64	_strtoull32	_strtoull64
_strtouq32	_strtouq64	_tmpnam32	_tmpnam64
_wcscat32	_wcscat64	_wcschr32	_wcschr64
_wcscpy32	_wcscpy64	_wcsncat32	_wcsncat64
_wcsncpy32	_wcsncpy64	_wcspbrk32	_wcspbrk64
_wcsrchr32	_wcsrchr64	_wcsrtombs32	_wcsrtombs64
_wcsstr32	_wcsstr64	_wcstok32	_wcstok64
_wcstol32	_wcstol64	_wcstoul32	_wcstoul64
_wcswcs32	_wcswcs64	_wmemchr32	_wmemchr64
_wmemcpy32	_wmemcpy64	_wmemmove32	_wmemmove64
_wmemset32	_wmemset64		

A.5. Functions Available on OpenVMS Version 7.2 and Higher

Table A.5 lists additional functions available on OpenVMS VAX and OpenVMS Alpha Version 7.2 and higher.

Table A.5. Functions Added in OpenVMS Version 7.2

asctime_r	dlerror
ctime_r	dlopen
decc$set_child_standard_streams	dlsym
decc$validate_wchar	fcntl

decc$write_eof_to_mbx	gmtime_r
dlclose	localtime_r

A.6. Functions Available on OpenVMS Version 7.3 and Higher

Table A.6 lists additional functions available on OpenVMS VAX and OpenVMS Alpha Version 7.3 and higher.

Table A.6. Functions Added in OpenVMS Version 7.3

fchown	
link	
utime	
utimes	
writev	

A.7. Functions Available on OpenVMS Version 7.3-1 and Higher

Table A.7 lists additional functions available on OpenVMS Alpha Version 7.3-1 and higher.

Table A.7. Functions Added in OpenVMS Version 7.3-1

access	ftello
chmod	ftw
chown	readdir_r
decc$feature_get_index	stat
decc$feature_get_name	vfscanf
decc$feature_get_value	vfwscanf
decc$feature_set_value	vscanf
fseeko	vwscanf
fstat	vsscanf
	vswscanf

A.8. Functions Available on OpenVMS Version 7.3-2 and Higher

Table A.8 lists additional functions available on OpenVMS Alpha Version 7.3-2 and higher.

Table A.8. Functions Added in OpenVMS Version 7.3-2

a64l	clock_getres	clock_gettime	clock_settime
endgrent	getgrent	getgrgid	getgrgid_r

getgrnam	getgrnam_r	getpgid	getpgrp
_getpwnam64	getpwnam_r	_getpwnam_r64	_getpwent64
getpwuid	_getpwuid64	getpwuid_r	_getpwuid_r64
getsid	l64a	nanosleep	poll
pread	pwrite	rand_r	readv
_readv64	seteuid	setgrent	setpgid
setpgrp	setregid	setreuid	setsid
sighold	sigignore	sigrelse	sigtimedwait
sigwait	sigwaitinfo	snprintf	ttyname_r
vsnprintf	__writev64	decc$set_child_default_dir	

A.9. Functions Available on OpenVMS Version 8.2 and Higher

Table A.9 lists additional functions available on OpenVMS Alpha and Integrity servers Version 8.2 and higher.

Table A.9. Functions Added in OpenVMS Version 8.2

clearerr_unlocked	feof_unlocked
ferror_unlocked	fgetc_unlocked
fputc_unlocked	flockfile
ftrylockfile	funlockfile
getc_unlocked	getchar_unlocked
putc_unlocked	putchar_unlocked
statvfs	fstatvfs
_glob32	_glob64
_globfree32	_globfree64
socketpair	

A.10. Functions Available on OpenVMS Version 8.3 and Higher

Table A.10 lists additional functions available on OpenVMS Alpha and Integrity servers Version 8.3 and higher.

Table A.10. Functions Added in OpenVMS Version 8.3

crypt	readlink
encrypt	realpath
setkey	symlink
lchown	unlink
lstat	fchmod

A.11. Functions Available on OpenVMS Version 8.4 and Higher

Table A.11 lists additional functions available on OpenVMS Alpha and Integrity servers Version 8.4 and higher.

Table A.11. Functions Added in OpenVMS Version 8.4

ftok	sem_init
semctl	sem_open
semget	sem_post
semop	sem_timedwait
sem_close	sem_trywait
sem_destroy	sem_unlink
sem_getvalue	sem_wait

Appendix B. Prototypes Duplicated to Nonstandard Headers

The various standards dictate which header file must define each of the standard functions. This is the included header file documented with each function prototype in the Reference Section of this manual.

However, many of the functions defined by the standards already existed on several operating systems and were defined in different header files. This is especially true on OpenVMS systems with the header files `<processes.h>`, `<unixio.h>`, and `<unixlib.h>`.

So, to provide upward compatibility for these functions, their prototypes are duplicated in both the expected header file as well as the header file defined by the standards.

Table B.1 lists these functions.

Table B.1. Duplicated Prototypes

Function	Duplicated in	Standard says
access	<unixio.h>	<unistd.h>
alarm	<signal.h>	<unistd.h>
bcmp	<string.h>	<strings.h>
bcopy	<string.h>	<strings.h>
bzero	<string.h>	<strings.h>
chdir	<unixio.h>	<unistd.h>
chmod	<unixio.h>	<stat.h>
chown	<unixio.h>	<unistd.h>
close	<unixio.h>	<unistd.h>
creat	<unixio.h>	<fcntl.h>
ctermid	<stdio.h>	<unistd.h>
cuserid	<stdio.h>	<unistd.h>
dirname	<string.h>	<libgen.h>
dup	<unixio.h>	<unistd.h>
dup2	<unixio.h>	<unistd.h>
ecvt	<unixlib.h>	<stdlib.h>
execl	<processes.h>	<unistd.h>
execle	<processes.h>	<unistd.h>
execlp	<processes.h>	<unistd.h>
execv	<processes.h>	<unistd.h>
execve	<processes.h>	<unistd.h>
execvp	<processes.h>	<unistd.h>
_exit	<stdlib.h>	<unistd.h>
fcvt	<unixlib.h>	<stdlib.h>
ffs	<string.h>	<strings.h>

Function	Duplicated in	Standard says
fsync	<stdio.h>	<unistd.h>
ftime	<time.h>	<timeb.h>
gcvt	<unixlib.h>	<stdlib.h>
getcwd	<unixlib.h>	<unistd.h>
getegid	<unixlib.h>	<unistd.h>
getenv	<unixlib.h>	<stdlib.h>
geteuid	<unixlib.h>	<unistd.h>
getgid	<unixlib.h>	<unistd.h>
getopt	<stdio.h>	<unistd.h>
getpid	<unixlib.h>	<unistd.h>
getppid	<unixlib.h>	<unistd.h>
getuid	<unixlib.h>	<unistd.h>
index	<string.h>	<strings.h>
isatty	<unixio.h>	<unistd.h>
lseek	<unixio.h>	<unistd.h>
mkdir	<unixlib.h>	<stat.h>
mktemp	<unixio.h>	<stdlib.h>
nice	<stdlib.h>	<unistd.h>
open	<unixio.h>	<fcntl.h>
pause	<signal.h>	<unistd.h>
pipe	<processes.h>	<unistd.h>
read	<unixio.h>	<unistd.h>
rindex	<string.h>	<strings.h>
sbrk	<stdlib.h>	<unistd.h>
setgid	<unixlib.h>	<unistd.h>
setuid	<unixlib.h>	<unistd.h>
sleep	<signal.h>	<unistd.h>
strcasecmp	<string.h>	<strings.h>
strncasecmp	<string.h>	<strings.h>
system	<processes.h>	<stdlib.h>
times	<time.h>	<times.h>
umask	<stdlib.h>	<stat.h>
vfork	<processes.h>	<unistd.h>
wait	<processes.h>	<wait.h>
write	<unixio.h>	<unistd.h>