

---

# Preface

The C programming interface described in this manual consists of over 500 functions, macros and external variables. This includes all functions defined in the ANSI Standard and required by the X/Open Portability Guide Issue 4, Version 2, called **XPG4 Version 2** for short. The optional "Encryption" function group of XPG4 and numerous other extensions are also supported.

The C programming interface presented here is a component of the C runtime library (BS2000) V2.5 which, in turn, is a component of the Common Runtime Environment **CRTE** V2.1 and will be released for operating system version BS2000/OSD V3.0 and higher and for OSD-SVP V2.0 and higher. The POSIX subsystem must be loaded in order to obtain the full functionality of the C library functions described in this manual.

The C library functions provide a convenient method of programming many tasks for which no higher-level language facilities are included in C itself. Typical examples of such programming tasks include:

- processing of files (open, close, seek, read, write, etc.)
- processing of individual characters or strings (search, change, copy, delete etc.)
- dynamic memory management (allocation and deallocation of storage areas, etc.)
- access to the operating system
- use of mathematical functions

All functions in the reference section "Functions and variables in alphabetical order" which are not identified as "extensions" in the title behave in conformance with the above standards (see following section). Extensions to the functionality of individual functions and provisional restrictions until branding are indicated explicitly in each description.

### Extensions

Besides the international standards mentioned above, the C library supports functions of the C runtime library (BS2000) up to version V2.4 (see also the "C Library Functions (BS2000)" manual as well as numerous other extensions which are supported on many UNIX systems, e.g. also under SINIX. The extensions in the previous C library (BS2000) are identified in the titles of the reference section by the keyword *BS2000*. The newly added extensions are identified by the keyword *extension*. This explicit identification of extensions is intended to facilitate the development of portable programs.

The functions for input/output, signal handling and the locale support extensions that are compatible with earlier versions of the C runtime library. In particular, both the data management system of BS2000/OSD (DMS) as well as the XPG4 Version 2 conformant POSIX file system can be accessed (see the manual "POSIX Basics (BS2000/OSD)").

The following are also available as additional extensions:

- 64 bit function to support NFS V3.0
- Functions to support POSIX threads in the C runtime library

### Restrictions

This version of the C runtime library is subject to the following restrictions as opposed to XPG4 Version 2:

When the environment (external variable `environ`) is reinitialized using `putenv()`, the file system defaults to DMS, so the user must explicitly set `PROGRAM-ENVIRONMENT` to `SHELL` (see the section "Scope of the supported C library" on page 18 and the C and C++ User Guides).

Specific restrictions are indicated where relevant under the actual function descriptions.

## Target group

This manual is intended for C programmers who wish to accomplish the following tasks:

- port C programs from UNIX or SINIX platforms to the POSIX subsystem
- write C programs for XPG4 Version 2 conformant environments (POSIX subsystem) under BS2000/OSD
- create C programs that can access both an XPG4 Version 2 conformant file system and DMS.

Knowledge of the C programming language and of the POSIX subsystem and BS2000/OSD operating systems are prerequisite to working with this manual.

## Summary of contents

This manual is organized into three parts: a conceptual part, a reference part, and cross-referencing aids.

The conceptual part, which follows this preface, includes the following chapters:

- a general description of the most important features of the C library and the basic characteristics of interactions between operating systems
- lists of all functions, macros and external variables described in the reference part, arranged by subject matter into themes.

The reference part contains detailed descriptions of each individual function, macro, and variable in alphabetical order. This is followed by a description of all header files, also listed alphabetically.

The cross-referencing aids include a glossary of terms and a list of references to related literature in addition to a detailed index.

### Documentation of CRTE and the C development system

The C and C++ User Guides explain in detail how the CRTE library can be accessed when compiling, linking and executing a C/C++ program.

General notes and linkage examples for the common runtime environment of C, C++ and COBOL85/COBOL2000 can be found in the "CRTE (BS2000/OSD)" User Guide.

## Organization of the POSIX documentation

The following documentation is available to familiarize the user and to facilitate working with the POSIX subsystem in BS2000/OSD:

- An overview of the strategy and objectives of POSIX in BS2000/OSD can be found in the brochure "POSIX in BS2000/OSD".
- An introduction to working with the POSIX subsystem is presented in the manual "POSIX - Basics for Users and System Administrators". Furthermore, the administration tasks that arise in conjunction with the POSIX subsystem are described. You also learn with which BS2000/OSD software products you can use the POSIX subsystem.
- A comprehensive description of the POSIX commands that can be used when working in the POSIX shell can be found in the manual "POSIX - Commands".
- The "POSIX Commands of the C and C++ Compilers" manual provides an introduction to the C-/C++ programming environment in the POSIX shell environment, describes how to compile and link C and C++ programs with the POSIX commands `cc`, `c89` and `CC` and shows you how to control the global C and C++ list generator with the POSIX command `ccxref`.
- The "POSIX V1.1A Sockets/XTI for POSIX" manual is intended for C and C++ programmers that develop communication applications with `SOCKETS` or `XTI` functions based on the POSIX interface.
- "NFS V3.0 / NFS V1.2C Network File System"

### POSIX documentation in the BS2000/OSD environment

Many software products in BS2000/OSD have been functionally extended to include the POSIX functionality.

A number of utility routines provide access to the POSIX file system. The file editor EDT V16.5, for example, can also process files of the POSIX file system.

Due to the CRTE (Common Runtime Environment) extensions based on the XPG4 Version 2 standard, it is now possible to write portable C programs using the C library functions V2.5 independently of the executing operating system.

Familiarity with the manual "POSIX - Basics for Users and System Administrators" is essential as a foundation for accessing POSIX functionality from other software products.

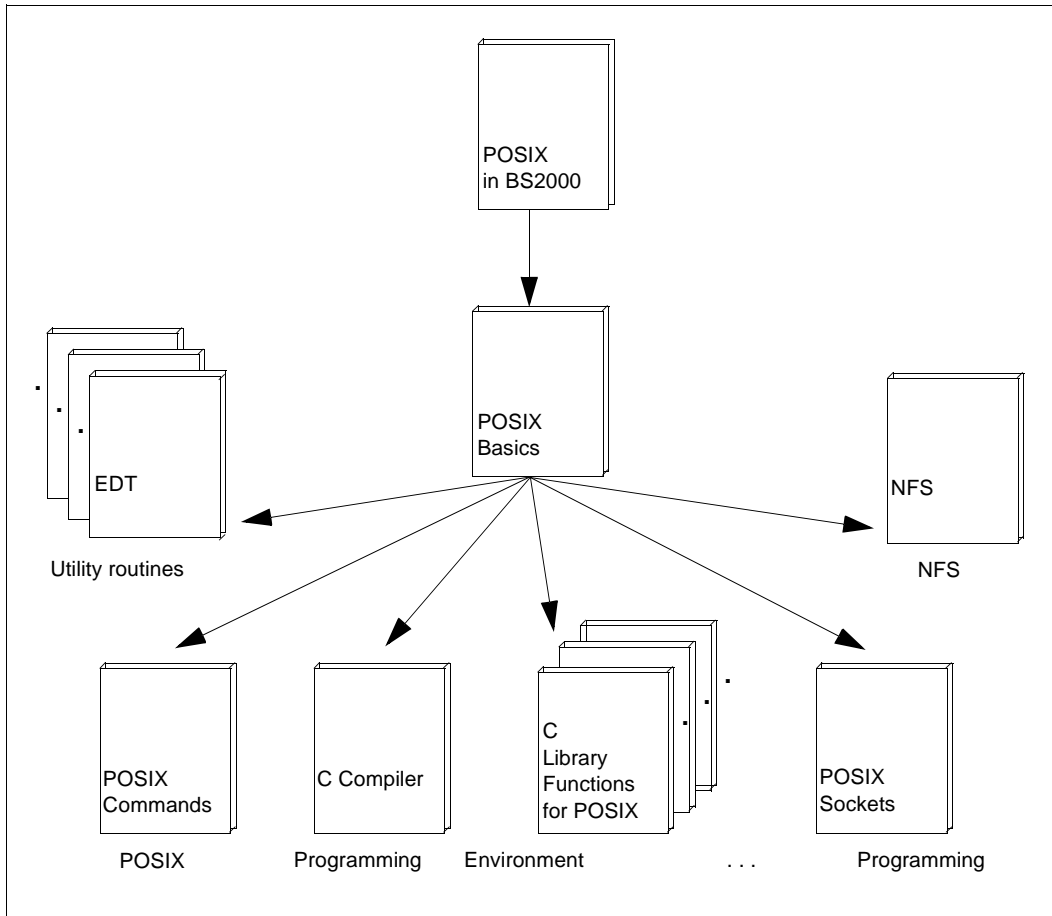


Figure 1: POSIX documentation in the BS2000/OSD environment

## Notational conventions

The following conventions are used in this manual to represent statement formats and user input:

monospace	Used to represent names that are part of the C language scope and the C library and to indicate cross-references to other terms described in the Glossary. Also used for sample inputs and outputs in examples.
UPPERCASE	Used to indicate symbolic constants (e.g. <code>HUGE_VAL</code> ), symbolic names of signals (e.g. <code>SIGABRT</code> ) and error codes (e.g. <code>EDOM</code> ) that are not implementation-dependent.
{UPPERCASE}	Indicates implementation-dependent symbolic constants that are defined in the header file <code>limits.h</code> (e.g. <code>{INT_MAX}</code> ).
<i>italics</i>	Indicate sample names for parameters in user input.
[ ]	Used to indicate optional syntax elements, i.e. elements that can, but need not be used. The square brackets themselves must not be specified.
...	Ellipses are used in syntax representations to indicate that the preceding syntactic unit may be repeated.
	Used in examples to indicate an omission of program code.
▬	This character is used to explicitly indicate a mandatory blank in order to avoid misunderstandings. In general, any white space is considered a blank character.
	This character is used as a separator for alternative specifications. One of the adjacent entries must be selected. The vertical bar itself must not be specified.
<code>Key</code>	Used to represent keys.
<code>Key1</code> + <code>Key2</code>	Indicates keys that must be pressed simultaneously.
<code>Key1</code> <code>Key2</code>	Indicates keys that must be pressed in succession.

Descriptions of the entry formats that are used in the reference section to organize the function descriptions can be found at the beginning of chapter "Functions and variables in alphabetical order (a - m)" on page 161.

## Changes since the last manual

Descriptions of the following groups of functions have been added to the manual:

- C library functions (BS2000) from versions V2.3B and V2.4A of the C runtime library (see the following sections for more details).
- 64 bit functions to support large files (> 2 GB)
- Functions to support POSIX threads

### New features of the C runtime library V2.3B

The following changes are the result of major new features in the C runtime library that were already implemented in Version 2.3B:

- Support for the new data types `wint_t`, `wctrans_t` and `mbstate_t`
- New header file `<wctype.h>` with definitions for wide characters and multibyte characters
- New header file `<iso646.h>` with the definitions of C keyword operators
- Support for all functions from Amendment 1 to the standard ISO/IEC 9899:1990

The descriptions of the functions listed in the following were added to this manual.

Functions declared in `<wchar.h>`:

<code>btowc</code>	<code>swprintf</code>	<code>wctob</code>
<code>fwide</code>	<code>vfwprintf</code>	<code>wmemchr</code>
<code>fwprintf</code>	<code>vswprintf</code>	<code>wmemcmp</code>
<code>fwscanf</code>	<code>vwprintf</code>	<code>wmemcpy</code>
<code>mbrlen</code>	<code>wrtomb</code>	<code>wmemmove</code>
<code>mbrtowc</code>	<code>wcsrtombs</code>	<code>wmemset</code>
<code>mbsinit</code>	<code>wcstoll</code>	<code>wprintf</code>
<code>mbsrtowcs</code>	<code>wcstoul</code>	<code>wscanf</code>
<code>swscanf</code>	<code>wcsstr</code>	

Functions declared in `<wctype.h>`:

`towctrans`  
`wctrans`

## New features of the C runtime library V2.4A

The following changes are the result of major new features in the C runtime library that were already implemented in Version 2.4A:

- The conversion tables based on `IO_CONVERSION` were extended to be compatible with 8 bit code.
- Two new locales were added for Euro support:  
`De.EDF04F` and  
`De.EDF04F@euro`.

These two locales differ only by the category `LC_MONETARY`.

Both locales are based on the new 8 bit code table.

The monetary decimal point (`mon_decimal_point`) is the "," character (comma) for both locales.

- An additional link option (`SYSLNK.CRTE.TIME` or `SRULNK.CRTE.TIME` library) that permits the use of the POSIX time functions without the program connecting to a POSIX subsystem that may be loaded.  
 The procedure `ICXTZ` in the `SINPRC.CRTE.023` library is available for setting the time zone for the POSIX time functions.
- The following functions were added to support the `long long integer` data type (since the rounding functions are a group of functions, the functions for the other data types were also added):

### – Rounding functions

<code>double rint</code>	<code>float round</code>
<code>float rintf</code>	<code>float roundf</code>
<code>long double rintl</code>	<code>long double roundl</code>
<code>long int lrint</code>	<code>long int lround</code>
<code>long int lrintf</code>	<code>long int lroundf</code>
<code>long int lrintl</code>	<code>long int lroundl</code>
<code>long long int llrint</code>	<code>long long int llround</code>
<code>long long int llrintf</code>	<code>long long int llroundf</code>
<code>long long int llrintl</code>	<code>long long int llroundl</code>

### – Absolute value and division

<code>long long int llabs</code>	<code>lldiv_t lldiv</code>
----------------------------------	----------------------------



- Conversion functions

<code>long long int atoll</code>	<code>long long int wcstoll</code>
<code>long long int strtoll</code>	<code>long long int wcstoull</code>
<code>long long int strtoull</code>	

- Extensions to the functions for formatting input and output by adding the `ll` format specification for values of type `long long int`.
- The output functions affected are `printf`, `fprintf`, `sprintf`, `vprintf`, `vfprintf` and `vsprintf` and the corresponding wide character functions `wprintf`, `fwprintf`, `swprintf`, `vwprintf`, `vfwprintf` and `vswprintf` as well as the input functions `scanf`, `fscanf`, `sscanf`, `vscanf`, `vfscanf` and `vsscanf` and the corresponding wide character functions `wscanf`, `fwscanf`, `swscanf`, `vwscanf`, `vfwscanf` and `vswscanf`.
- The preprocessor define `XPG_IV_SPECIAL` was replaced by `_XOPEN_SOURCE_EXTENDED`.

## Changes since the C library manual V2.2A

The following changes are the result of major new features in the C runtime library that were already implemented in Version 2.3B:

- New header files `<wchar.h>` and `<wctype.h>` with definitions for wide characters and multibyte characters.
- New header file `<iso646.h>` with the definitions of C keyword operators (see page 13).



---

# The C programming interface

This chapter describes the system requirements and components of the C programming interface and specific aspects related to its use.

## System requirements

The following table lists the software products required to support the full functionality of the C library as provided by CRTE V2.3A and described in this manual

Product	Relevant components
BS2000/OSD-BC V3.0	<ul style="list-style-type: none"><li>– Operating system</li><li>– Header files for POSIX functions</li></ul>
C/C++ V3.0	C and C++ compiler for POSIX subsystem and BS2000/OSD
CRTE V2.3A	<ul style="list-style-type: none"><li>– Header files for BS2000 functions</li><li>– Runtime modules of the C library functions</li></ul>
POSIX-BC V3.0	<ul style="list-style-type: none"><li>– POSIX file system</li><li>– Basic shell</li><li>– POSIX-HEADER V1.3</li></ul>
SDF-P	Variable structure <code>SYSPOSIX</code> for initializing the runtime environment

The commands of the **POSIX subsystem**, which includes two products, POSIX-BC and POSIX-SH, are described in the manual "POSIX Commands (BS2000/OSD)". POSIX-SH commands increase the user's level of comfort when working in the POSIX shell but are not a system requirement for compiling, linking and starting C programs.

## Components of the C library

The programming interface of the C runtime library supports more than 500 predefined functions (see also the table on page 19 ff). These functions are available either as source program fragments (macros) or in the form of precompiled program segments (modules). The function declarations, definitions of constants, data types and macros, and the function macros themselves are incorporated in "header files" (also called "include files" or simply "headers").

### Header files

The header files for the C programming interface are supplied with two separate products:

The headers for POSIX functions are supplied as POSIX HEADER components with the POSIX-BC product, and the headers for BS2000 functions are supplied with CRTE V2.3A (see table on page 11).

Headers may be included, i.e. copied into a program at compilation by means of an `#include` preprocessor directive. A detailed description on how this is accomplished can be found in the C and C++ User Guides.

Headers contain declarations or definitions for the following:

- functions or corresponding macros
- external variables
- symbolic constants and data types

The XPG4 Version 2 conformant headers are described in the chapter "Header files" on page 929.

Header files contain external "C" declarations for all functions and data. This allows C library functions to be called from C++ sources.

In the POSIX subsystem, header files are located in the standard directories `/usr/include` and `/usr/include/sys`.

In BS2000, header files are stored as PLAM library members (of type S) in the libraries `$.SYSLIB.CRTE` (for BS2000 functions) and `$.SYSLIB.POSIX-HEADER` (for POSIX functions).

The compiler will accept `include` statements in which the names of header elements contain slashes (/) for directories even if PLAM library elements are involved. Each slash in the name of a user-defined or standard header is internally converted to a period (.) for the purpose of searching PLAM libraries.

Consequently, when porting source programs from POSIX or SINIX to BS2000, for example, the slashes need not be converted to periods.

Similarly, periods need not be converted to slashes in source programs which are copied from the BS2000 environment to the POSIX subsystem. This applies only to the standard header elements, however, not to the user-defined headers.

### Header file `iso646.h`

The header file `iso646.h` contains the following 11 macros that are expanded to the symbols to the right of the macro and that therefore represent alternative ways of writing the operators:

<code>and</code>	<code>&amp;&amp;</code>	<code>compl</code>	<code>~</code>	<code>or_eq</code>	<code> =</code>
<code>and_eq</code>	<code>&amp;=</code>	<code>not</code>	<code>!</code>	<code>xor</code>	<code>^</code>
<code>bitand</code>	<code>&amp;</code>	<code>not_eq</code>	<code>!=</code>	<code>xor_eq</code>	<code>^=</code>
<code>bitor</code>	<code> </code>	<code>or</code>	<code>  </code>		

## Functions and macros

Most of the library functions are implemented as C functions, a few as macros. Some library functions are implemented both as a function and as a macro.

If a library function exists in both variants, the macro variant is generated for the call by default. A function call is generated if the name is enclosed within parentheses `()` or is undefined by means of the `#undef` statement. The selection of an appropriate variant in each case will depend on whether and which specific aspects (performance, program size, restrictions) are relevant to a particular program.

A **function** is a compiled program segment (module) which is available only once and is treated as an external subroutine at runtime. An organizational overhead is required for each function call during program execution, e.g. to manage the local, dynamic data of a function in the runtime stack, to save register contents, for return addresses, etc.

Some library functions can be generated inline under the control of the `OPTIMIZATION` compiler option. In such cases, the function code is inserted directly at the calling point, and the above-mentioned administrative activities are not required.

The following functions can be generated inline in the present version: `strcpy()`, `strcmp()`, `strlen()`, `strcat()`, `memcpy()`, `memcmp()`, `memset()`, `abs()`, `fabs()`, `labs()` (see also the C and C++ User Guides).

A **macro** is a source program segment that is defined by means of a `#define` statement. During compilation, the macro name in the source program is replaced by the contents of the called macro whenever the macro is called.

Using macros can improve performance during program execution, since the runtime system is not required to perform administrative activities (see "function"); however, the size of the compiled program is increased due to the macro expansions.

The following should also be taken into account when using macros:

- Macro names cannot be passed as arguments to any function that requires a pointer to a function as an argument.
- The use of increment/decrement or compound assignment operators for macro arguments may produce undesirable side effects.
- The header file containing the macro definition must always be included in the program.

## 64-bit functions to support NFS V3.0

File systems that contain files larger than 2 gigabytes (GB) can be processed with NFS V3.0. To do this, a new variant was added to numerous I/O functions that contain the suffix "64". For example, the data type `off64_t` was added to the 32-bit data type `off_t`.

The compiler environment makes all these explicit 64-bit functions and types available in addition to the 32-bit functions and types. This allows a program to use both interfaces as needed.

An application must use the 64-bit interface in order to work with large files (>2 GB).

It is only reasonable to use these functions together with POSIX since large file systems and large files can only be reached via NFS.

Since most of the names of the 64-bit functions are abbreviated to 8 characters and are therefore not unique throughout CRTE, sources that need to use the 64-bit functions must be generated as LLMs.

The new functions can only be used if the `_LARGEFILE64_SOURCE=1` define is set (prototype generation and other defines). Furthermore, the value `UNLIMITED64` must be set for the `FILESIZE` parameter in the POSIX parameter file.

## POSIX thread support in the C runtime library

CRTE V2.3A supports POSIX threads through new header files and functions. This manual contains descriptions of the new functionality resulting from the POSIX thread support.

## Wide characters and multi-byte characters

Wide characters and multi-byte characters were defined to extend the "character" concept of computer languages in which one character was allocated one byte of storage space. This allocation is insufficient for languages such as Japanese, for example, since the representation of a character in these languages requires more than one byte of storage. For this reason multi-byte characters and wide characters were added to the character concept. Multi-byte characters represent the characters in the extended character set using two, three or more bytes.

Multi-byte strings can contain "shift sequences" that change the meaning of the following multi-byte codes. Shift sequences can switch between different interpretation modes, for example: The one byte shift sequence `0200` can specify that the following two bytes are to be interpreted as Japanese characters, and the shift sequence `0201` can specify that the following two bytes are to be interpreted as characters in the ISO-Latin-1 character set.

### Programming model

Programs that work with multi-byte characters can be just as easily realized with the help of Amendment 1 functions as programs that use the traditional character concept.

When they are used, the multi-byte characters or strings that are read in from an external file are read into a `wchar_t` object or a field of type `wchar_t` internally. The multi-byte characters are converted to the corresponding wide characters during the read operation in this case.

The `wchar_t` objects can then be edited using `iswxxx` functions or `wcstod`, `wmemcmp`, etc. The resulting `wchar_t` objects are then output using output functions such as `putwchar`, `fputws`, etc.

The wide characters are converted to the corresponding multi-byte characters when output.

### Notes on wide characters

A wide character is defined as the code value of an object of type `wchar_t` (binary encoded integer value) that corresponds to an element of the extended character set.

The null character has the code value `null`.

The end-of-file criterion in wide character files is `WEOF`.

Wide character constants are written in the form `L"wide character string"`.

### Notes on this implementation

Only 1 byte characters are supported as wide characters in this version of the C runtime library. They are of type `wchar_t`, which are mapped to the `long` type internally.

Multi-byte characters correspondingly are always 1 byte long.

## Time functions

The time functions that are used without POSIX when the C library functions are used, i.e. when the POSIX link option was not used, differ in three ways from the time functions used in POSIX/UNIX environments:

- The start date for the conversion to seconds (epoch) is January 1, 1950, 00:00:00 (January 1, 1970 for POSIX).
- Time specifications are strongly locale-dependent, and when the clock switches to daylight savings time and back, the time specifications "jump". Negative numbers and differences in time can arise, especially when daylight savings time is over, that yield unexpected results later on during processing.
- The `gmtime` function is implemented like `localtime`.

A result of the use of January 1, 1950 as the start date is that the corresponding time functions do not work any more starting in the year 2018. The year 2018 is therefore a much larger problem for C applications in BS2000 systems than the Year 2000 problem. When the POSIX time functions are used, the problem arises twenty years later, and the corresponding specialists already know about the problem and are working to solve it.

**For these reasons we recommend users to convert their programs to the POSIX time functions.**

The POSIX time functions are used automatically when the POSIX link option is used, and no POSIX subsystem needs to be present to use it.

However, if the POSIX subsystem is already loaded, then linking with the POSIX link option will cause the program to connect to the POSIX subsystem.

If you use the POSIX link option, then the POSIX functions described in the "C Library Functions for POSIX" manual are also used, for example for I/O functions, and in particular file names that are not explicitly designated as BS2000 file names are interpreted as POSIX-UFS file names.

If you want your program to use only the POSIX time functions, then you need to use the TIME link option.

The libraries

- `SYSLNK.CRTE.TIME` and
- `SRULNK.CRTE.TIME`

are available for inclusion in your program.

If you do not use the TIME link option, then all existing programs and procedures will respond as before.



## Setting the time zone for POSIX time functions

The POSIX time functions evaluate the TZ variable to determine the time zone.

You can set the time zone before starting the program via the SYSPPOSIX variable. If the variable is not set when the program is started, then the C runtime library initializes the variable to the time zone in which Germany is located by setting TZ to the value MET-1DST,M3.5.0/02:00:00,M10.5.0/03:00:00.

The procedure ICXTZ in the SINPRC.CRTE.023 library is available for setting the time zone to a value other than the one valid for Germany when you want to set a different time zone for the installation.

ICXTZ,(TZ=*'time zone specification'*)

## Scope of the supported C library

The following table provides an overview of the supported C library functions.

### Key

The following characters have the following meanings in the "XPG5" column:

- x Function required by XPG5. These functions can only be executed with POSIX-BC and are portable with respect to XPG5-conformant systems.
- xx Functions that were available with the same functionality in the previous BS2000 library.
- d Functions that can process BS2000 files in addition to POSIX files.
- a Functions that have been extended to include the functionality of the previous C (BS2000) library function.
- y Function which is required by XPG5 and is portable with respect to XPG5-conformant systems. This function is also executable without POSIX-BC.

The following characters have the following meanings in the "XPG4" column:

- x Function required by XPG4 Version 2. These functions can only be executed with POSIX-BC and are portable with respect to XPG4 Version 2-conformant systems.
- xx Functions that were available with the same functionality in the previous BS2000 library.
- d Functions that can process BS2000 files in addition to POSIX files.
- a Functions that have been extended to include the functionality of the previous C (BS2000) library function.
- y Function which is required by XPG4 Version 2 and is portable with respect to XPG4 Version 2-conformant systems. This function is also executable without POSIX-BC.

x in "*Extension*" or "*BS2000*" column mean:

Extension that is only executable with POSIX-BC (*Extension*) or that was already present in the previous C library (*BS2000*).

y in "XPG4 Version 2" column means:

Functions that are required by XPG4 Version 2 and are portable with respect to XPG4 Version 2-conformant systems. The function is also executable without POSIX-BC.

x in "ANSI" column means:

Functions that are not in XPG4 Version 2 and also do not represent a BS2000 extension, but were implemented instead in accordance with the ANSI C standard (`__STDC_VERSION` 199901L).

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
a64l()	y	y			
abort()	xa	xa			
abs()	xx	xx			
access()	x	x			
acos()	xx	xx			
acosh()	x	x			
advance()	x	x			
alarm()	xa	xa			
altzone			x		
ascii_to_ebcdic()			y		
asctime()	xx	xx			
asctime_r()	y				
asin()	xx	xx			
asinh()	x	x			
assert()	xx	xx			
atan()	xx	xx			
atan2()	xx	xx			
atanh()	x	x			
atexit()	xx	xx			
atof()	xx	xx			
atoi()	xx	xx			
atol()	xx	xx			
atoll()					x
basename()	x	x			
bcmp()	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
bcopy()	x	x			
brk()	x	x			
bs2exit()				x	
bs2fstat()				x	
bs2system()			x	(system())	
bsd_signal()	x	x			
bsearch()	xx	xx			
btowc()	y				
bzero()	x	x			
cabs()				x	
calloc()	xx	xx			
catclose()	x	x			
catgets()	x	x			
catopen()	x	x			
cbirt()	x	x			
cdisco()				x	
ceil()	xx	xx			
ceilf()					x
ceilll()					x
cenaco()				x	
cfgetispeed()	x	x			
cfgetospeed()	x	x			
cfsetispeed()	x	x			
cfsetospeed()	x	x			
chdir()	x	x			
chmod()	x	x			
chown()	x	x			
chroot()	x	x			
clearerr()	xxd	xxd			
clock()	xa	xa			
close()	xxd	xxd			
closedir()	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
closelog()	x	x			
compile()	x	x			
confstr()	x	x			
cos()	x	xx			
cosh()	xx	xx			
cputime()				x	
creat()	xxd	xxd			
crypt()	x	x			
cstxlt()				x	
ctermid()	x	x			
ctime()	xa	xa			
ctime_r()	y				
cuserid()	x	x			
__DATE__		xx			
daylight	x	x			
dbm_clearerr()	x	x			
dbm_close()	x	x			
dbm_delete()	x	x			
dbm_error()	x	x			
dbm_fetch()	x	x			
dbm_firstkey()	x	x			
dbm_nextkey()	x	x			
dbm_open()	x	x			
dbm_store()	x	x			
difftime()	xx	xx			
dirname()	x	x			
div()	xx	xx			
drand48()	x	x			
dup()	x	x			
dup2()	x	x			
ebcdic_to_ascii()			y		
ecvt()	xx	xx			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
_edt()				x	
encrypt()	x	x			
endgrent()	x	x			
endpwent()	x	x			
endutxent()	x	x			
environ	x	x			
erand48()	x	x			
erf()	xx	xx			
erfc()	xx	xx			
errno	xx	xx			
execl()	x	x			
execle()	x	x			
execlp()	x	x			
execv()	x	x			
execve()	x	x			
execvp()	x	x			
_exit()	xx	xx			
exit()	xx	xx			
exp()	xx	xx			
expm1()	y	y			
fabs()	xx	xx			
fattach()	x	x			
fchdir()	x	x			
fchmod()	x	x			
fchown()	x	x			
fclose()	xxd	xxd			
fcntl()	x	x			
fcvt()	xx	xx			
fdelrec()				x	
fdetach()	x	x			
fdopen()	xxd	xxd			
feof()	xxd	xxd			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
ferror()	xxd	xxd			
fflush()	xxd	xxd			
ffs()	x	x			
fgetc()	xxd	xxd			
fgetpos()	xxd	xxd			
fgets()	xxd	xxd			
fgetwc()	yd	yd			
fgetws()	yd	yd			
__FILE__		xx			
fileno()	xxd	xxd			
flocate()				x	
flockfile()	y				
floor()	xx	xx			
floorf()					x
floorl()					x
fmod()	xx	xx			
fmtmsg()	x	x			
fopen()	xxd	xxd			
fork()	x	x			
fpathconf()	x	x			
fprintf()	xxd	xxd			
fputc()	xxd	xxd			
fputs()	xxd	xxd			
fputwc()	yd	yd			
fputws()	yd	yd			
fread()	xxd	xxd			
free()	xx	xx			
freopen()	xxd	xxd			
frexp()	xx	xx			
fscanf()	xxd	xxd			
fseek()	xxd	xxd			
fsetpos()	xxd	xxd			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
fstat()	xd	xd			
fstatvfs()	x	x			
fsync()	x	x			
ftell()	xxd	xxd			
ftello()	yd	yd			
ftime()	xa	xa			
ftok()	x	x			
ftruncate()	x	x			
ftrylockfile()	y				
ftw()	x	x			
funlockfile()	y				
fwide()	y				x
fwprintf()	y				x
fwrite()	xxd	xxd			x
fwscanf()	y				
FD_CLR()		x			
FD_ISSET()		x			
FD_SET()		x			
FD_ZERO()		x			
gamma()	xx	xx			
garbcoll()				x	
gcvt()	xx	xx			
getc()	xxd	xxd			
getc_unlocked()	yd				
getchar()	xxd	xxd			
getchar_unlocked()	yd				
getcontext()	x	x			
getcwd()	x	x			
getdate()	x	x			
getdents()			x		
getdtablesize()	x	x			
getegid()	x	x			



Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
getenv()	xx	xx			
geteuid()	x	x			
getgid()	x	x			
getgrent()	x	x			
getgrgid()	x	x			
getgrgid_r()	x				
getgrnam()	x	x			
getgrnam_r()	x				
getgroups()	x	x			
gethostid()	y	y			
getitimer()	x	x			
getlogin()	xx	xx			
getlogin_r()	y				
getmsg()	x	x			
getopt()	x	x			
getpagesize()	x	x			
getpass()	x	x			
getpgit()	x	x			
getpid()	x	x			
getpgrpname()				x	
getpgrp()	x	x			
getpmsg()	x	x			
getppid()	x	x			
getpriority()	x	x			
getpwent()	x	x			
getpwnam()	x	x			
getpwnam_r()	x				
getpwuid()	x	x			
getpwuid_r()	x				
getrlimit()	x	x			
getrusage()	x	x			
gets()	xxd	xxd			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
getsid()	x	x			
getsubopt()	x	x			
gettimeofday()	x	x			
gettsn()				x	
getuid()	x	x			
getutxent()	x	x			
getutxid()	x	x			
getutxline()	x	x			
getw()	xxd	xxd			
getwc	yd	yd			
getwd()	x	x			
getwchar()	yd	yd			
gmatch()			x		
gmtime()	xa	xa			
gmtime_r()	xa				
grantpt()	x	x			
hcreate()	x	x			
hdestroy()	x	x			
hsearch()	x	x			
hypot()	xx	xx			
iconv()	x	x			
iconv_close()	x	x			
iconv_open()	x	x			
ilogb()	y	y			
index()	xx	xx			
initstate()	x	x			
insque()	x	x			
ioctl()	x	x			
isalnum()	xx	xx			
isalpha()	xx	xx			
isascii()	xx	xx			
isastream()	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
isatty()	x	x			
iscntrl()	xx	xx			
isdigit()	xx	xx			
isebcdic()				x	
isgraph()	xx	xx			
islower()	xx	xx			
isnan()	x	x			
isprint()	xx	xx			
ispunct()	xx	xx			
isspace()	xx	xx			
isupper()	xx	xx			
iswalnum()	x	x			
iswalpha()	x	x			
iswcntrl()	x	x			
iswctype()	x	x			
iswdigit()	x	x			
iswgraph()	x	x			
iswlower()	x	x			
iswprint()	x	x			
iswpunct()	x	x			
iswspace()	x	x			
iswupper()	x	x			
iswxdigit()	x	x			
isxdigit()	xx	xx			
j0()	xx	xx			
j1()	xx	xx			
jn()	xx	xx			
jrands48()	x	x			
kill()	xa	xa			
killpg()	x	x			
_longjmp()	y	y			
l64a()	y	y			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
labs()	xx	xx			
lchown()	x	x			
lcong48()	x	x			
ldexp()	xx	xx			
ldiv()	xx	xx			
lfind()	x	x			
lgamma()	x	x			
__LINE__		xx			
link()	x	x			
llabs()					x
lldiv()					x
llrint()					x
llrintf()					x
llrintl()					x
llround()					x
llroundf()					x
llroundl()					x
loc1	x	x			
loc2	x	x			
localeconv()	xx	xx			
localtime()	xa	xa			
localtime_r()	xa				
lockf()	x	x			
locs	x	x			
log()	xx	xx			
log10()	xx	xx			
log1p()	y	y			
logb()	y	y			
longjmp()	xx	xx			
lrand48()	x	x			
lrint()					x
lrintf()					x

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
lrintl()					x
lround()					x
lroundf()					x
lroundl()					x
lsearch()	x	x			
lseek()	xxd	xxd			
lstat()	x	x			
major()			x		
makecontext()	x	x			
makedev()			x		
malloc()	xx	xx			
mblen()	xx	xx			
mbrlen()	y				x
mbrtowc()	y				x
mbsinit()	y				x
mbsrtowcs()	y				x
mbstowcs()	xx	xx			
mbtowc()	xx	xx			
memalloc()				x	
memcpy()	x	x			
memchr()	xx	xx			
memcmp()	xx	xx			
memcpy()	xx	xx			
memfree()				x	
memmove()	xx	xx			
memset()	xx	xx			
minor()			x		
mkdir()	x	x			
mkfifo()	x	x			
mknod()	x	x			
mkstemp()	x	x			
mktemp()	xa	xa			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
mktime()	xa	xa			
mmap()	x	x			
modf()	xx	xx			
mount()			x		
mprotect()	x	x			
mrnd48()	x	x			
msgctl()	x	x			
msgget()	x	x			
msgrcv()	x	x			
msgsnd()	x	x			
msync()	x	x			
munmap()	x	x			
nanosleep()	y				
nextafter()	y	y			
nftw()	x	x			
nice()	x	x			
nl_langinfo()	x	x			
nrnd48()	x	x			
offsetof()				x	
open()	xxd	xxd			
opendir()	x	x			
openlog()	x	x			
optarg	x	x			
opterr	x	x			
optint	x	x			
optopt	x	x			
pathconf()	x	x			
pause()	x	x			
pclose()	x	x			
perror()	xxd	xxd			
pipe()	x	x			
poll()	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
popen()	x	x			
pow()	xx	xx			
printf()	xxd	xxd			
ptsname()	x	x			
putc()	xxd	xxd			
putc_unlocked()	yd				
putchar()	xxd	xxd			
putchar_unlocked()	yd				
putenv()	x	x			
putmsg()	x	x			
putpmsg()	x	x			
putpwent()			x		
puts()	xxd	xxd			
pututxline()	x	x			
putw()	xxd	xxd			
putwc()	yd	yd			
putwchar()	yd	yd			
qsort()	xx	xx			
raise()	xa	xa			
rand()	xx	xx			
rand_r()	y				
random()	x	x			
re_cmp()	x	x			
re_exec()	x	x			
read()	xxd	xxd			
readdir()	x	x			
readdir_r()	x				
readlink()	x	x			
readv()	x	x			
realloc()	xx	xx			
realpath()	x	x			
regcmp()	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
regex()	x	x			
remainder()	y	y			
remove()	xxd	xxd			
remque()	x	x			
rename()	xxd	xxd			
rewind()	xxd	xxd			
rewinddir()	x	x			
rindex()	xx	xx			
rint()	y	y			
rintf()					x
rintl()					x
rmdir()	x	x			
round()					x
roundf()					x
roundl()					x
_setjmp	y	y			
sbrk()	x	x			
scalb()	y	y			
scanf()	xxd	xxd			
seed48()	x	x			
seekdir()	x	x			
select()	x	x			
semctl()	x	x			
semget()	x	x			
semop()	x	x			
setbuf()	xxd	xxd			
setcontext()	x	x			
setgid()	x	x			
setgrent()	x	x			
setitimer()	x	x			
setjmp()	xx	xx			
setkey()	x	x			



Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
setlocale()	xa	xa			
setlogmask()	x	x			
setpgid()	x	x			
setpgrp()	x	x			
setpriority()	x	x			
setpwent()	x	x			
setregid()	x	x			
setreuid()	x	x			
setrlimit()	x	x			
setsid()	x	x			
setstate()	x	x			
setuid()	x	x			
setutxent()	x	x			
setvbuf()	xxd	xxd			
shmat()	x	x			
shmctl()	x	x			
shmdt()	x	x			
shmget()	x	x			
sigaction()	x	x			
sigaddset()	x	x			
sigdelset()	x	x			
sigemptyset()	x	x			
sigfillset()	x	x			
sighold()	x	x			
sigignore()	x	x			
siginterrupt()	x	x			
sigismember()	x	x			
siglongjmp()	x	x			
signal()	xa	xa			
signalstack()	x	x			
sigpause()	x	x			
siggam	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
sigpending()	x	x			
sigprocmask()	x	x			
sigrelse()	x	x			
sigset()	x	x			
sigsetjmp()	x	x			
sigstack()	x	x			
sigsuspend()	x	x			
sin()	xx	xx			
sinh()	xx	xx			
sleep()	xa	xa			
sprintf()	xx	xx			
sqrt()	xx	xx			
srand()	xx	xx			
srand48()	x	x			
srandom()	x	x			
sscanf()	xx	xx			
stat()	xd	xd			
statvfs()	x	x			
__STDC__		xx			
__STDC_VERSION__					x
stderr	xx	xx			
stdin	xx	xx			
stdout	xx	xx			
step()	x	x			
strcasecmp()	x	x			
strcat()	xx	xx			
strchr()	xx	xx			
strcmp()	xx	xx			
strcoll()	xa	xa			
strcpy()	xx	xx			
strcspn()	xx	xx			
strdup()	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
strerror()	xx	xx			
strfill()				x	
strftime()	xa	xa			
strlen()	xx	xx			
strlower()				x	
strncasecmp()	x	x			
strncat()	xx	xx			
strncmp()	xx	xx			
strncpy()	xx	xx			
strpbrk()	xx	xx			
strrchr()	xx	xx			
strspn()	xx	xx			
strstr()	xx	xx			
strtod()	xa	xa			
strtok()	xx	xx			
strtok_r()	y				
strtol()	xx	xx			
strtoll()					x
strtoul()	xx	xx			
strtoull()					x
strupper()				x	
strxfrm()	xa	xa			
swab()	x	x			
swapcontext()	x	x			
swprintf()	y				x
swscanf()	y				x
symlink()	x	x			
sync()	x	x			
sysconf()	x	x			
sysfs()			x		
syslog()	x	x			
system()	xa	xa			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
tan()	xx	xx			
tanh()	xx	xx			
tcdrain()	x	x			
tcflow()	x	x			
tcflush()	x	x			
tcgetattr()	x	x			
tcgetpgrp()	x	x			
tcgetsid()	x	x			
tcsendbreak()	x	x			
tcsetattr()	x	x			
tcsetpgrp()	x	x			
tdelete()	x	x			
tell()				x	
telldir()	x	x			
tempnam()	x	x			
tfind()	x	x			
__TIME__		xx			
time()	xa	xa			
times()	x	x			
timezone	x	x			
tmpfile()	xxd	xxd			
tmpnam()	xxd	xxd			
toascii()	xx	xx			
toebcdic()				x	
_tolower()	x	x			
tolower()	xa	xa			
_toupper()	x	x			
toupper()	xa	xa			
towctrans()	x				x
towlower()	x	x			
towupper()	x	x			
truncate()	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
tsearch()	x	x			
ttyname()	x	x			
ttyname_r()	x				
ttyslot()	x	x			
twalk()	x	x			
tzname	x	x			
tzset()	x	x			
ualarm()	x	x			
ulimit()	x	x			
umask()	x	x			
umount()			x		
uname()	x	x			
ungetc()	xxd	xxd			
ungetwc()	x	x			
unlink()	xxd	xxd			
unlockpt()	x	x			
usleep()	x	x			
utime()	x	x			
utimes()	x	x			
va_arg()	xx	xx			
va_end()	xx	xx			
va_start()	xx	xx			
valloc()	y	y			
vfork()	x	x			
vfprintf()	xxd	xxd			
vwprintf	y				x
vprintf()	xxd	xxd			
vsprintf()	xx	xx			
vswprintf()	y				x
wprintf()	y				x
wait()	x	x			
wait3()	x	x			

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
waitid()	x	x			
waitpid()	x	x			
wcrtomb()	y				
wscat()		x			
wcschr()		x			
wcscmp()		x			
wcscoll	x	x			
wscpy()	x	x			
wcscspn()	x	x			
wcsftime	x	x			
wcslen()	x	x			
wcsncat()	x	x			
wcsncmp()	x	x			
wcsncpy()	x	x			
wcspbrk()	x	x			x
wcsrchr()	x	x			x
wcsrtombs()	x				x
wcsspn()	x	x			
wcsstr()	x				
wcstod()	x	x			x
wcstok()	x	x			
wcstol()	x	x			
wcstoll()					x
wcstombs()	xx	xx			
wcstoul()	x	x			
wcstoull()					x
wcswcs()	x	x			
wcswidth()	x	x			
wcsxfrm	x	x			
wctob()	y				x
wctomb()	xx	xx			
wctrans()	y				x

Function	XPG5	XPG4 Version 2	Extensions		ANSI
			<i>Extension</i>	<i>BS2000</i>	
wctype()	x	x			
wcwidth()	x	x			
wmemchr()	y				x
wmemcmp()	y				x
wmemcpy()	y				x
wmemmove()	y				x
wmemset()	y				x
wprintf()	y				x
write()	xxd	xxd			
writev()	x	x			
wscanf()	y				x
y0()	xx	xx			
y1()	xx	xx			
yn()	xx	xx			

## Selecting functionality

It is possible to choose between the different functionalities. In the following, a distinction is made between the range of functions that have been extended by the POSIX functionality and the range of functions available in BS2000 (without POSIX), which represents the BS2000 functionality.

The C library functions which provide the BS2000 functionality form the basis of the library. In addition, the extra functions of the C library provide the POSIX functionality. Therefore, in choosing the extended functionality, you can use all functions of the library, i.e. both the BS2000 functions and the additional XPG4 Version 2-conformant functions.

A small number of functions have different variants in BS2000 and in POSIX. These are, on the one hand, the functions for input/output and file accesses (for a list of these functions, see page 75) and, on the other, time functions, signal processing and interrupt functions, plus the `clock()` and `system()` functions.

Below is described for both types of functionality which variant of the respective functions is used.

### Range of functions extended by the POSIX functionality

When a program is compiled, linked and started in the **POSIX shell** (see also the manual "C/C++ POSIX Commands of the C and C++ Compilers", "C Compiler" or "C++ Compiler"), the available functionality of the C library is as listed in the following. This functionality is called the **POSIX functionality** in the following:

- All XPG4 Version 2-conformant functions (marked with x, y or xx in the "XPG4 Version 2" column in the table on page 19ff) are supported.
- All functions identified as extensions (marked with an x in the "Extension" and "BS2000" columns in the table on page 19ff) are supported.
- XPG4 Version 2-conformant functionality is supported for all functions marked with xa. This includes the following function groups:
  - the time functions `clock()`, `ctime()`, `ctime_r()`, `ftime()`, `gmtime()`, `localtime()`, `mktime()`, `time()`
  - the functions for process control, i.e. `abort()`, `alarm()`, `_exit()`, `kill()`, `raise()`, and `signal()`.
- In the case of functions that are marked with xd in the table, it is possible to access BS2000 or POSIX files on an individual basis. This can be controlled as described under section "Selecting the file system and the system environment" on page 42. This function group also includes the `system()` function, since it can be controlled on the source program level as in the case of the file access functions.



Programs that are run in the POSIX shell are started internally with `fork()` and an `exec` function and thus have a parent process.

The range of functions extended by POSIX functionality can also be selected when a program is compiled, linked and started on the **BS2000 command level**, provided the following is taken into account:

1. Steps to be observed at compilation:

- a) In addition to the `$.SYSLNK.CRTE` library, the `$.SYSLIB.POSIX-HEADER` library must be specified so that the correct header files are found (option `STD-INCLUDE-LIBRARY`).
- b) `_OSD_POSIX` must be defined. This can be done by choosing one of the methods given below:
  - by specifying the following before the first `#include` statement in the source code:

```
#define _OSD_POSIX
```
  - by setting the `SOURCE-PROPERTIES` option for the compilation run as follows:

```
SOURCE-PROPERTIES=PAR(DEFINE=_OSD_POSIX)
```

2. When linking the link option `$.SYSLNK.CRTE.POSIX` must be specified before `$.SYSLNK.CRTE` or `$.SYSLNK.CRTE.PARTIAL-BIND`.

Programs that are compiled, linked and started on the BS2000 command level as indicated above are executed in a task and thus have no parent process.

## BS2000 functionality

Users who wish to use only BS2000 functionality in a program must compile and link such programs with only the library `$.SYSLNK.CRTE`.

The environment variable `PROGRAM-ENVIRONMENT='SHELL'` must not be set.

If you are using only BS2000 functionality, it is best to work with the manual "C Library Functions (BS2000/OSD)".

Only a part of the library is supported when BS2000 functionality is selected. The following restrictions apply:

- All XPG4 Version 2-conformant functions that were supported by the previous (BS2000) C library (marked with `xx` in the "XPG4" column in the table on page 19ff.) are fully supported.
- All functions that are identified as an extension with *BS2000* (marked with an `x` in the "BS2000" column in the table on page 19 ff.) are also supported.
- Functions marked with `xa` are supported with BS2000 functionality only.
- Functions that are marked in the table with `xd` can only access BS2000 files.

## Selecting the file system and the system environment

In the case of I/O functions and file access functions which can process both POSIX as well as BS2000 files, and which require a pathname to be specified as an argument, the file type to be processed in each case can be specified individually in the source code. Selecting the file type automatically determines the functionality with which the corresponding function is called. This is achieved via the environment variable `PROGRAM_ENVIRONMENT` on one hand, and by conforming to a specific syntax at the source program level on the other.

### Associating the I/O streams

If, when linking the program, you specified the POSIX linkage option and POSIX is active, the standard I/O streams `stdin`, `stdout` and `stderr` are opened via POSIX.

In batch jobs, procedures or if the `PROGRAM_ENVIRONMENT` environment variable is not set to `SHELL`, the standard I/O streams are associated via POSIX with the BS2000 system files (`SYSDTA`, `SYSOUT`), otherwise with the terminal.

Without POSIX, the standard I/O streams `stdin`, `stdout` and `stderr` are directly associated with the BS2000 system files (`SYSDTA`, `SYSOUT`).

## Setting the `PROGRAM_ENVIRONMENT` variable

The `PROGRAM_ENVIRONMENT` environment variable is used in BS2000/OSD to set whether file names or commands specified in the `system()` function call which have no BS2000 or POSIX prefix, are interpreted as BS2000 or POSIX files or commands.

At the BS2000 command level, `PROGRAM_ENVIRONMENT` is not set. For how to set environment variables, see section "Environment variables" on page 71.

When the POSIX shell is started, the `PROGRAM_ENVIRONMENT` variable is automatically set to the value `SHELL`, i.e. file names and commands which do not begin with `"/BS2/"` are interpreted as POSIX file names or commands.

File names or commands which do not comply with the syntax rules of the relevant environment are acknowledged with an error message.

If the specified file or command does not exist in the chosen environment, this is also reported with a message.

### Explicit identification of file names as POSIX or BS2000

If the file name begins with a slash (`/`), it is interpreted as an absolute pathname of a POSIX file.

If the file name is specified in the format `*POSIX(name)`, it is likewise interpreted as a POSIX file name.

If the file name begins with `/BS2/`, the file name which follows the `/BS2/` is interpreted as a BS2000 file name.

### Explicit identification of commands

If the command specified in the `system()` function call begins with `/BS2/`, the command which follows the `/BS2/` is interpreted as a BS2000 command.

If the command is specified in the format `*POSIX(command)`, it is interpreted as a POSIX command.

## Syntax in the source program

If a POSIX file is to be processed, the absolute pathname of the file must be specified (see the manual "POSIX Basics (BS2000/OSD)") or the name must be qualified with `*POSIX(filename)`.

If a BS2000 file is to be processed, the file name must be qualified with `/BS2/`. As soon as a BS2000 file is accessed, the BS2000 functionality of the corresponding function applies. Deviations from the XPG4 functionality, if any, are indicated by the marker *BS2000* in the left margin of all relevant descriptions.

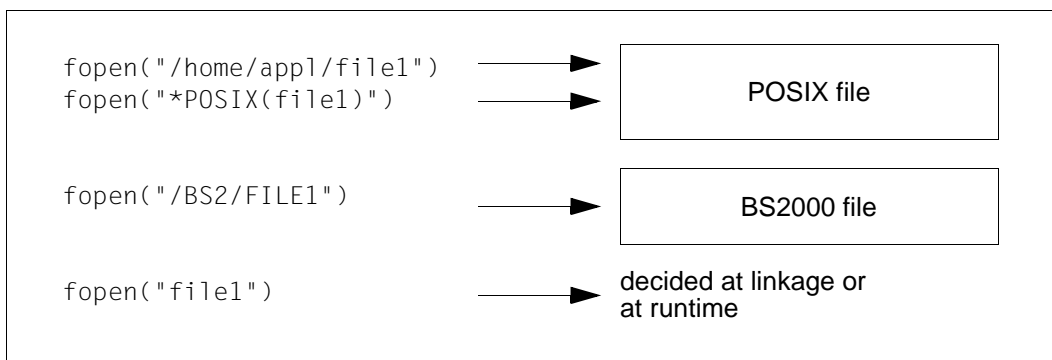


Figure 2: Control options at source code level

The `system()` function can be controlled analogously. The only difference is that a command for the desired system environment is specified instead of a file name.

## Portability

Users that want to write programs that are portable according to the XPG4 Version 2 standard must set the `_XOPEN_SOURCE` macro as well as the `_XOPEN_SOURCE_EXTENDED` macro to the value 1. The identifiers required and expressly permitted by the XPG4 Version 2 standard are made visible in this manner. These macros must be set before the first header file is included. This can be done during compilation by specifying the corresponding compiler option or in the source code using `#define` statements.

XPG4 Version 2 defined identifiers are undefined only if the `#undef` statement has been specified (see also section "Functions and macros" on page 13 ). These `#undef` statements must be called before the `#include` statements.

When the `_XOPEN_SOURCE` macro is set to 500, only the identifiers required or explicitly allowed by the XPG4 Version 2 standard are made visible. The `_XOPEN_SOURCE_EXTENDED` macro is ignored in this case. When the `_XOPEN_SOURCE` macro is not set to 500 but the `_XOPEN_SOURCE_EXTENDED` is set to 1, then only the identifiers contained in the XPG4 Version 2 standard are visible.

`_XOPEN_SOURCE_EXTENDED` can be defined for the compilation run. Therefore, to support maximum portability, it is advisable to ensure that `_XOPEN_SOURCE_EXTENDED` is set to 1 in applications by using a compiler option or by entering a `#define` statement before the first `#include` statement in the source code.

Applications which use functionality that is marked in this manual as an extension (indicated by *BS2000* or *Extension*) are not strictly XPG4 Version 2-conformant or ISO C-conformant.

To write programs that are portable according to the XPG5 standard, the `_XOPEN_SOURCE` macro must be set to 500. The `_XOPEN_SOURCE_EXTENDED` macro is ignored in this case. Not all function groups and header files contained in the XPG5 standard are realized in this implementation (for example, there is no asynchronous I/O and no real time functions). The corresponding function test macros are set to -1 in the header file `<unistd.h>`.

## Name space

All identifiers mentioned in this manual, except `environ`, are defined in at least one header file (see also chapter "Functions and variables in alphabetical order (n - y)" on page 601 ). If `_XOPEN_SOURCE` is defined, every header file may potentially define or declare identifiers that conflict with those of the application. The set of identifiers visible to an application consists of the identifiers included by means of the `#include` statement and the additional identifiers reserved by the implementation (see also the C and C++ User Guides).

## Character sets

In this version, the C runtime library supports the portable character set of XPG4 Version 2 and only EBCDIC as a coded character set.

### Portable character set

Every supported locale refers to the portable character set, which consists of 128 characters (7-bit code). The following table shows the symbolic name, the corresponding glyph, the class name of the POSIX locale, and the ASCII and EBCDIC codes for every character in the portable character set.

Symbolic name	Glyphs	Class of POSIX locale	ASCII		EBCDIC
			decimal	hex	hex
<NUL>		control	0	00	00
<SOH>		control	1	01	01
<STX>		control	2	02	02
<ETX>		control	3	03	03
<EOT>		control	4	04	37
<ENQ>		control	5	05	2D
<ACK>		control	6	06	2E
<alert>		control	7	07	2F
<backspace>		control	8	08	16
<tab>		control space blank	9	09	05
<newline>		control space	10	0A	15
<vertical-tab>		control space	11	0B	0B
<form-feed>		control space	12	0C	0C
<carriage-return>		control space	13	0D	0D
<SO>		control	14	0E	0E
<SI>		control	15	0F	0F
<DLE>		control	16	10	10
<DC1>		control	17	11	11
<DC2>		control	18	12	12
<DC3>		control	19	13	13
<DC4>		control	20	14	3C
<NAK>		control	21	15	3D

Symbolic name	Glyphs	Class of POSIX locale	ASCII		EBCDIC
			decimal	hex	hex
<SYN>		control	22	16	32
<ETB>		control	23	17	26
<CAN>		control	24	18	18
<EM>		control	25	19	19
<SUB>		control	26	1A	3F
<ESC>		control	27	1B	27
<IS4>		control	28	1C	1C
<IS3>		control	29	1D	1D
<IS2>		control	30	1E	1E
<IS1>		control	31	1F	1F
<space>		space blank	32	20	40
<exclamation-mark>	!	punct	33	21	5A
<quotation-mark>	"	punct	34	22	7F
<number-sign>	#	punct	35	23	7B
<dollar-sign>	\$	punct	36	24	5B
<percent-sign>	%	punct	37	25	6C
<ampersand>	&	punct	38	26	50
<apostrophe>	'	punct	39	27	7D
<left-parenthesis>	(	punct	40	28	4D
<right-parenthesis>	)	punct	41	29	5D
<asterisk>	*	punct	42	2A	5C
<plus-sign>	+	punct	43	2B	4E
<comma>	,	punct	44	2C	6B
<hyphen>	-	punct	45	2D	60
<period>	.	punct	46	2E	4B
<slash>	/	punct	47	2F	61
<zero>	0	digit xdigit	48	30	F0
<one>	1	digit xdigit	49	31	F1
<two>	2	digit xdigit	50	32	F2
<three>	3	digit xdigit	51	33	F3
<four>	4	digit xdigit	52	34	F4
<five>	5	digit xdigit	53	35	F5



Symbolic name	Glyphs	Class of POSIX locale	ASCII		EBCDIC
			decimal	hex	hex
<six>	6	digit xdigit	54	36	F6
<seven>	7	digit xdigit	55	37	F7
<eight>	8	digit xdigit	56	38	F8
<nine>	9	digit xdigit	57	39	F9
<colon>	:	punct	58	3A	7A
<semicolon>	;	punct	59	3B	5E
<less-than-sign>	<	punct	60	3C	4C
<equals-sign>	=	punct	61	3D	7E
<greater-than-sign>	>	punct	62	3E	6E
<question-mark>	?	punct	63	3F	6F
<commercial-at>	@	punct	64	40	7C
<A>	A	upper xdigit	65	41	C1
<B>	B	upper xdigit	66	42	C2
<C>	C	upper xdigit	67	43	C3
<D>	D	upper xdigit	68	44	C4
<E>	E	upper xdigit	69	45	C5
<F>	F	upper xdigit	70	46	C6
<G>	G	upper	71	47	C7
<H>	H	upper	72	48	C8
<I>	I	upper	73	49	C9
<J>	J	upper	74	4A	D1
<K>	K	upper	75	4B	D2
<L>	L	upper	76	4C	D3
<M>	M	upper	77	4D	D4
<N>	N	upper	78	4E	D5
<O>	O	upper	79	4F	D6
<P>	P	upper	80	50	D7
<Q>	Q	upper	81	51	D8
<R>	R	upper	82	52	D9
<S>	S	upper	83	53	E2
<T>	T	upper	84	54	E3
<U>	U	upper	85	55	E4

Symbolic name	Glyphs	Class of POSIX locale	ASCII		EBCDIC
			decimal	hex	hex
<V>	V	upper	86	56	E5
<W>	W	upper	87	57	E6
<X>	X	upper	88	58	E7
<Y>	Y	upper	89	59	E8
<Z>	Z	upper	90	5A	E9
<left-square-bracket>	[	punct	91	5B	BB
<backslash>	\	punct	92	5C	BC
<right-square-bracket>	]	punct	93	5D	BD
<circumflex>	^	punct	94	5E	6A
<underscore>	_	punct	95	5F	6D
<grave-accent>	`	punct	96	60	4A
<a>	a	lower xdigit	97	61	81
<b>	b	lower xdigit	98	62	82
<c>	c	lower xdigit	99	63	83
<d>	d	lower xdigit	100	64	84
<e>	e	lower xdigit	101	65	85
<f>	f	lower xdigit	102	66	86
<g>	g	lower	103	67	87
<h>	h	lower	104	68	88
<i>	i	lower	105	69	89
<j>	j	lower	106	6A	91
<k>	k	lower	107	6B	92
<l>	l	lower	108	6C	93
<m>	m	lower	109	6D	94
<n>	n	lower	110	6E	95
<o>	o	lower	111	6F	96
<p>	p	lower	112	70	97
<q>	q	lower	113	71	98
<r>	r	lower	114	72	99
<s>	s	lower	115	73	A2
<t>	t	lower	116	74	A3
<u>	u	lower	117	75	A4

Symbolic name	Glyphs	Class of POSIX locale	ASCII		EBCDIC
			decimal	hex	hex
<v>	v	lower	118	76	A5
<w>	w	lower	119	77	A6
<x>	x	lower	120	78	A7
<y>	y	lower	121	79	A8
<z>	z	lower	122	7A	A9
<left-curly-bracket>	{	punct	123	7B	FB
<vertical-line>		punct	124	7C	4F
<right-curly-bracket>	}	punct	125	7D	FD
<tilde>	~	punct	126	7E	FF
<DEL>	DEL	control	127	7F	07

The EBCDIC character set is an 8-bit codeset and includes a total of 256 characters. The different variants of the EBCDIC character set can be found in the manual "BS2000/OSD-BC System Applications" .

The symbolic names of the portable character set are used for the assignment of the coded character set in a codeset table.

### Wide character codes

All wide character codes in a process consist of characters with the same number of bits. **Wide characters** must not be confused with **multi-byte characters**, which may consist of a variable number of bytes.

Although the C runtime library supports functions that process multi-byte characters, the actual length of a multi-byte character in this version is only 1 byte (= 8 bits), since only EBCDIC is available for the wide character codeset.

## Character classes

The preceding table shows the assignment of characters from the portable character set to character classes as defined by the `LC_CTYPE` category in the POSIX locale. The following additional character classes which represent supersets or subsets of those classes are also defined:

Character class	Scope
alpha	upper + lower
blank	subset of space: <blank> and <tab>
cntrl	control characters
digit	decimal characters
graph	alpha + digit + punct + space
lower	lowercase letters
print	alpha + digit + punct
punct	punctuation characters
space	white-space characters
tolower	mapping of uppercase letters to lowercase
toupper	mapping of lowercase letters to uppercase
upper	uppercase letters
xdigit	set of characters for hexadecimal representation: digit + A-F + a-f

## Locale

The locale is a subset of the settings for the runtime environment. It affects the behavior of C programs with respect to country-specific conventions, norms and languages. The locale consists of one or more categories. The following categories are supported in XPG4 Version 2-conformant environments:

`LC_ALL` Determines all values of the current locale.

`LC_COLLATE` Determines the collating sequence of characters. Each character is defined in relation to another by means of a weight. This affects the behavior of the `strcoll()` and `strxfrm()`.

The name of the corresponding definition file in the POSIX subsystem is `/usr/lib/locale/locale/LC_COLLATE`.

The corresponding table in BS2000 is named `COLL/uscol`.

`LC_CTYPE` Determines character classification (i.e. the assignment of characters to character classes), case conversion (i.e. the association between uppercase and lowercase letters) and other character attributes.

The name of the corresponding definition file in the POSIX subsystem is `/usr/lib/locale/locale/LC_CTYPE`.

There are 3 tables in BS2000 for all EBCDIC characters:

The classification table `TYPE/ustyp` assigns each EBCDIC character to a particular character class. The classes are represented by the following values:

Character class	Assembler code	C code
Uppercase letter (upper)	X'01'	_U
Lowercase letter (lower)	X'02'	_L
Decimal digit (digit)	X'04'	_N
White space (space)	X'08'	_S
Punctuation character (punct)	X'10'	_P
Control character (cntrl)	X'20'	_C
Hexadecimal character (xdigit)	X'40'	_X

The C values are defined in the header file `ctype.h`.

The tables for converting uppercase letters to lowercase (`LOWER/uslow`) and lowercase letters to uppercase (`UPPER/usupp`) indicate the character obtained from the conversion of each character from `X'00'` to `X'FF'`.

These tables are used by the macros `toupper()` and `tolower()` for converting to uppercase and lowercase letters, respectively. The table needs to be filled only for characters which are classified as uppercase or lowercase letters in the classification table.

**LC\_MESSAGES** Determines the format of messages.

The name of the corresponding definition file in the POSIX subsystem is `/usr/lib/locale/locale/LC_MESSAGES`.

This category is not supported by the BS2000 functionality.

**LC\_MONETARY** Determines the formats of monetary values.

The name of the corresponding definition file in the POSIX subsystem is `/usr/lib/locale/locale/LC_MONETARY`.

**LC\_NUMERIC** Determines the representation of non-monetary numeric values for formatted I/O (`fprintf()`, `fscanf()`), the conversion of strings (`atof()`, `strtod()`), and the values returned by `localeconv()`.

The name of the corresponding definition file in the POSIX subsystem is `/usr/lib/locale/locale/LC_NUMERIC`.

**LC\_TIME** Determines the date and time representation for calls to `strfmon()`.

The name of the corresponding definition file in the POSIX subsystem is `/usr/lib/locale/locale/LC_TIME`.

These locale categories are also defined as environment variables.

The behavior of XPG4 Version 2-conformant commands (e.g. the POSIX commands) is affected by the current locale (see section "Environment variables" on page 71 and the manual "POSIX Commands (BS2000/OSD)"). The C library functions `setlocale()` and `localeconv()` may be used to change the current locale of a C program at runtime.

The following C library functions are directly affected by the current locale:

<code>atof()</code>	<code>isgraph()</code>	<code>isxdigit()</code>	<code>strxfrm()</code>
<code>isalnum()</code>	<code>islower()</code>	<code>localeconv()</code>	<code>tolower()</code>
<code>isalpha()</code>	<code>isprint()</code>	<code>setlocale()</code>	<code>toupper()</code>
<code>isascii()</code>	<code>ispunct()</code>	<code>strcoll()</code>	<code>wctomb()</code>
<code>iscntrl()</code>	<code>isspace()</code>	<code>strftime()</code>	<code>wctombs()</code>
<code>isdigit()</code>	<code>isupper()</code>	<code>strtod()</code>	

The C runtime library provides some predefined locales (see the section "Predefined locales" on page 55). However, users may also define their own locales (see section "User-specific locales" on page 70).

CRTE provides the predefined locales `De.EDF04F` and `De.EDF04F@euro` to support the Euro. These two locales differ only by the category `LC_MONETARY` that represents the German mark (DM) for the locale `De.EDF04F` and the Euro for the locale `De.EDF04F@euro`.

When the value of a locale environment variable begins with a slash (/), it is interpreted as the pathname of the locale definition.

Applications can change the current locale, i.e. set some other predefined locale by invoking `setlocale()` with the appropriate value. If the function is called with an empty string for *locale*, then the value of the environment variable that was specified using the *category* argument is evaluated:

```
setlocale(LC_ALL, "");
```

In this case all categories are determined by the corresponding environment variables. If the environment variable is unset or is set to an empty string, the environment is evaluated (see also section "Environment variables" on page 71).

## Predefined locales

The following locales are predefined in the C runtime library:

Locale	for BS2000 functionality	for XPG4 Version 2 functionality
POSIX	x	x
C	x	x
GERMANY	x	-
V1CTYPE	x	-
De.EDF04F	x	
De.EDF04F@euro	x	
V2CTYPE	x	-

The predefined locales are added to a program module at link time. A call to `setlocale()` sets an access pointer to the specified locale and thus makes it the current locale for the process.

## Locale files

The predefined locales for XPG4 Version 2 functionality are stored in the POSIX file system in the directory `/usr/lib/locale` in compliance with the following convention:

`/usr/lib/locale/locale/category`.

## POSIX or C locale

All XPG4 Version 2-conformant systems support the POSIX locale, which is also known as the C locale. The POSIX locale is the default locale for C programs at startup if `setlocale()` is not called.

The POSIX locales `C`, `De`, `De.EDF04F`, `De_DE.EDF04`, `De.EDF04@euro`, `De_DE.EDF04@EU`, `En_US.EDF04` and `POSIX` exist. The categories are defined as follows for the POSIX locales:

**LC\_COLLATE** The collation sequence for the characters specified in the Table on page 47 corresponds to the order given in the table. This affects only the functions `strcoll()` and `strxfrm()`.

**LC\_CTYPE** The classification corresponds to the EBCDIC definition of the individual characters (EBCDIC.DF.03-IRV, international version).

**LC\_NUMERIC** The components defined in `localeconv()` have the following values:

localeconv component	Value of the POSIX locale
<code>decimal_point</code>	"<period>"
<code>thousands_sep</code>	""
<code>grouping</code>	""

**LC\_MESSAGES** The constants defined in `langinfo.h` have the following values:

langinfo constant	Value
<code>YESEXPR</code>	"^[yY]"
<code>NOEXPR</code>	"^[nN]"
<code>YESSTR</code> Will no longer be supported by the X/ Open-Standard in future.	"yes"
<code>NOSTR</code> Will no longer be supported by the X/ Open-Standard in future.	"no"

**LC\_MONETARY** The components defined in `localeconv()` have the following values:

localeconv component	Value
<code>int_curr_symbol</code>	""
<code>currency_symbol</code>	""
<code>mon_decimal_point</code>	""
<code>mon_thousands_sep</code>	""
<code>mon_grouping</code>	""



localeconv component	Value
positive_sign	""
negative_sign	""
int_frac_digits	{CHAR_MAX}
frac_digits	{CHAR_MAX}
p_cs_precedes	{CHAR_MAX}
n_cs_precedes	{CHAR_MAX}
p_sep_by_space	{CHAR_MAX}
n_sep_by_space	{CHAR_MAX}
p_sign_pos	{CHAR_MAX}
n_sign_pos	{CHAR_MAX}

LC\_TIME

The constants defined in `langinfo.h` have the following values:

langinfo constant	Value
D_T_FMT	"%a %b %e %H:%M:%S %Y"
D_FMT	"%m/%d/%y"
T_FMT	"%H:%M:%S"
AM_STR	"AM"
PM_STR	"PM"
T_FMT_AMP	"%I:%M:%S %p"
DAY_1	"Sunday"
DAY_2	"Monday"
DAY_3	"Tuesday"
DAY_4	"Wednesday"
DAY_5	"Thursday"
DAY_6	"Friday"
DAY_7	"Saturday"
ABDAY_1	"Sun"
ABDAY_2	"Mon"
ABDAY_3	"Tue"
ABDAY_4	"Wed"
ABDAY_5	"Thu"
ABDAY_6	"Fri"
ABDAY_7	"Sat"
MON_1	"January"

langinfo constant	Value
MON_2	"February"
MON_3	"March"
MON_4	"April"
MON_5	"May"
MON_6	"June"
MON_7	"July"
MON_8	"August"
MON_9	"September"
MON_10	"October"
MON_11	"November"
MON_12	"December"
ABMON_1	"Jan"
ABMON_2	"Feb"
ABMON_3	"Mar"
ABMON_4	"Apr"
ABMON_5	"May"
ABMON_6	"Jun"
ABMON_7	"Jul"
ABMON_8	"Aug"
ABMON_9	"Sep"
ABMON_10	"Oct"
ABMON_11	"Nov"
ABMON_12	"Dec"

## V1CTYPE

This locale is identified as "V1CTYPE" or LC\_C\_V1CTYPE. It matches for the most part the "C" locale. Only the following differences arise in the classification of characters (category LC\_CTYPE):

In the "V1CTYPE" locale, the characters X'8B', X'8C' and X'8D' are in the character class lower; X'AB', X'AC' and X'AD' are in the character class upper and X'CO' and X'DO' are in the character class punct. In the "C" locale, all of these characters belong to the character class cntrl (i.e. control characters).

**V2CTYPE**

This locale is identified as "V2CTYPE" or LC\_C\_V2CTYPE. It matches for the most part the "C" locale. However, there is the following difference in the collation sequence of characters (category LC\_COLLATE): the collating order corresponds to that of the EBCDIC character set.

**GERMANY**

A country-specific locale is available for German-speaking regions. This locale is identified as "GERMANY" or LC\_C\_GERMANY. The following values, which deviate from those of the POSIX locale, apply:

**LC\_CTYPE**      The characters ä (X'FB'), ö (X'4F'), ü (X'FD'), and ß (X'FF') belong to the character class `lower`.

The characters Ä (X'BB'), Ö (X'BC') and Ü (X'BD') belong to the character class `upper`.

When lowercase characters are converted to uppercase (`toupper()`, `strupper()`), the character ß (X'FF') remains unchanged.

**LC\_MONETARY**   International currency symbol (`int_curr_symbol`): "EUR"

Local currency symbol (`currency_symbol`): "?"

Radix character (`mon_decimal_point`): ",", "

**LC\_TIME**        German is used for the days of the week and the months of the year.

The format for the date corresponds to the usual conventions for German-speaking countries:

*weekday name, day of month. name of month year*

**Example:**

Donnerstag, 25. Juli 1991

**De.EDF04F and De.EDF04F@euro**

These two locales support the processing of files and text that contain the Euro symbol.

The underlying conversion tables were extended to be compatible with 8 bit code in both locales. The conversion tables are based on the ISO 8859-15 ASCII code and the EDF04F EBCDIC code.

The two locales differ only by the category `LC_MONETARY`.

`LC_CTYPE`

The base class that each character belongs to can be determined from the following table:

Symbolic names	Glyphs	Class(es)	ASCII	EBCDIC
<NUL>		control	00	00
<SOH>		control	01	01
<STX>		control	02	02
<ETX>		control	03	03
<EOT>		control	04	37
<ENQ>		control	05	2D
<ACK>		control	06	2E
<alert>		control	07	2F
<backspace>		control	08	16
<tab>		control space blank	09	05
<newline>		control space	0A	15
<vertical-tab>		control space	0B	0B
<form-feed>		control space	0C	0C
<carriage-return>		control space	0D	0D
<SO>		control	0E	0E
<SI>		control	0F	0F
<DLE>		control	10	10
<DC1>		control	11	11
<DC2>		control	12	12
<DC3>		control	13	13
<DC4>		control	14	3C
<NAK>		control	15	3D
<SYN>		control	16	32
<ETB>		control	17	26
<CAN>		control	18	18

Symbolic names	Glyphs	Class(es)	ASCII	EBCDIC
<EM>		control	19	19
<SUB>		control	1A	3F
<ESC>		control	1B	27
<IS4>		control	1C	1C
<IS3>		control	1D	1D
<IS2>		control	1E	1E
<IS1>		control	1F	1F
<space>		space blank	20	40
<exclamation-mark>	!	punct	21	5A
<quotation-mark>	"	punct	22	7F
<number-sign>	#	punct	23	7B
<dollar-sign>	\$	punct	24	5B
<percent-sign>	%	punct	25	6C
<ampersand>	&	punct	26	50
<apostrophe>	'	punct	27	7D
<left-parenthesis>	(	punct	28	4D
<right-parenthesis>	)	punct	29	5D
<asterisk>	*	punct	2A	5C
<plus-sign>	+	punct	2B	4E
<comma>	,	punct	2C	6B
<hyphen>	-	punct	2D	60
<period>	.	punct	2E	4B
<slash>	/	punct	2F	61
<zero>	0	digit xdigit	30	F0
<one>	1	digit xdigit	31	F1
<two>	2	digit xdigit	32	F2
<three>	3	digit xdigit	33	F3
<four>	4	digit xdigit	34	F4
<five>	5	digit xdigit	35	F5
<six>	6	digit xdigit	36	F6
<seven>	7	digit xdigit	37	F7
<eight>	8	digit xdigit	38	F8
<nine>	9	digit xdigit	39	F9

Symbolic names	Glyphs	Class(es)	ASCII	EBCDIC
colon	:	punct	3A	7A
<semicolon>	;	punct	3B	5E
<less-than-sign>	<	punct	3C	4C
<equals-sign>	=	punct	3D	7E
<greater-than-sign>	>	punct	3E	6E
<question-mark>	?	punct	3F	6F
<commercial-at>	@	punct	40	7C
<A>	A	upper xdigit	41	C1
<B>	B	upper xdigit	42	C2
<C>	C	upper xdigit	43	C3
<D>	D	upper xdigit	44	C4
<E>	E	upper xdigit	45	C5
<F>	F	upper xdigit	46	C6
<G>	G	upper	47	C7
<H>	H	upper	48	C8
<I>	I	upper	49	C9
<J>	J	upper	4A	D1
<K>	K	upper	4B	D2
<L>	L	upper	4C	D3
<M>	M	upper	4D	D4
<N>	N	upper	4E	D5
<O>	O	upper	4F	D6
<P>	P	upper	50	D7
<Q>	Q	upper	51	D8
<R>	R	upper	52	D9
<S>	S	upper	53	E2
<T>	T	upper	54	E3
<U>	U	upper	55	E4
<V>	V	upper	56	E5
<W>	W	upper	57	E6
<X>	X	upper	58	E7
<Y>	Y	upper	59	E8
<Z>	Z	upper	5A	E9

Symbolic names	Glyphs	Class(es)	ASCII	EBCDIC
<left-square-bracket>	[	punct	5B	BB
<backslash>	\	punct	5C	BC
<right-square-bracket>	]	punct	5D	BD
<circumflex>	^	punct	5E	6A
<underscore>	_	punct	5F	6D
<grave-accent>	`	punct	60	4A
<a>	a	lower xdigit	61	81
<b>	b	lower xdigit	62	82
<c>	c	lower xdigit	63	83
<d>	d	lower xdigit	64	84
<e>	e	lower xdigit	65	85
<f>	f	lower xdigit	66	86
<g>	g	lower	67	87
<h>	h	lower	68	88
<i>	i	lower	69	89
<j>	j	lower	6A	91
<k>	k	lower	6B	92
<l>	l	lower	6C	93
<m>	m	lower	6D	94
<n>	n	lower	6E	95
<o>	o	lower	6F	96
<p>	p	lower	70	97
<q>	q	lower	71	98
<r>	r	lower	72	99
<s>	s	lower	73	A2
<t>	t	lower	74	A3
<u>	u	lower	75	A4
<v>	v	lower	76	A5
<w>	w	lower	77	A6
<x>	x	lower	78	A7
<y>	y	lower	79	A8
<z>	z	lower	7A	A9
<left-curly-bracket>	{	punct	7B	FB

Symbolic names	Glyphs	Class(es)	ASCII	EBCDIC
<vertical-line>		punct	7C	4F
<right-curly-bracket>	}	punct	7D	FD
<tilde>	~	punct	7E	FF
<DEL>	DEL	control	7F	07
<sc00>			80	20
<sc01>			81	21
<sc02>			82	22
<sc03>			83	23
<sc04>			84	24
<sc05>		control	85	25
<sc06>			86	06
<sc07>			87	17
<sc08>			88	28
<sc09>			89	29
<sc0a>			8A	2A
<sc0b>			8B	2B
<sc0c>			8C	2C
<sc0d>			8D	09
<sc0e>			8E	0A
<sc0f>			8F	1B
<sc10>			90	30
<sc11>			91	31
<sc12>			92	1A
<sc13>			93	33
<sc14>			94	34
<sc15>			95	35
<sc16>			96	36
<sc17>			97	08
<sc18>			98	38
<sc19>			99	39
<sc1a>			9A	3A
<sc1b>			9B	3B
<sc1c>			9C	04



Symbolic names	Glyphs	Class(es)	ASCII	EBCDIC
<sc1d>			9D	14
<sc1e>			9E	3E
<sc1f>			9F	5F
<nbsp>	NBSP		A0	41
<revexcl>	¡	punct	A1	AA
<cent>	¢	punct	A2	B0
<pound>	£	punct	A3	B1
<euro>	€	punct	A4	9F
<yen>	¥	punct	A5	B2
<CARON-S>	Š	upper	A6	D0
<section>	§	punct	A7	B5
<caron-s>	š	lower	A8	79
<copyright>	©	punct	A9	B4
<fem-ord>	ª	punct	AA	9A
<ang_q_l>	«	punct	AB	8A
<not>	¬	punct	AC	BA
<shy>	SHY	punct	AD	CA
<register>	®	punct	AE	AF
<macron>	ˉ	punct	AF	A1
<degree>	°	punct	B0	90
<plu-min>	±	punct	B1	8F
<sup-two>	²	punct	B2	EA
<sup-three>	³	punct	B3	FA
<CARON-Z>	Ž	upper	B4	BE
<micro>	μ	punct	B5	A0
<pilcrow>	¶	punct	B6	B6
<mid-dot>	·	punct	B7	B3
<caron-z>	ž	lower	B8	9D
<sup-one>	¹	punct	B9	DA
<mas-ord>	º	punct	BA	9B
<ang-q-r>	»	punct	BB	8B
<OE>	Œ	upper	BC	B7
<oe>	œ	lower	BD	B8

Symbolic names	Glyphs	Class(es)	ASCII	EBCDIC
<DIA-Y>	ÿ	upper	BE	B9
<revquest>	¿	punct	BF	AB
<GRAVE-A>	À	upper	C0	64
<ACUTE-A>	Á	upper	C1	65
<CIRC-A>	Â	upper	C2	62
<TILDE-A>	Ã	upper	C3	66
<DIA-A>	Ä	upper	C4	63
<RING-A>	Å	upper	C5	67
<AE>	Æ	upper	C6	9E
<CEDIL-C>	Ç	upper	C7	68
<GRAVE-E>	È	upper	C8	74
<ACUTE-E>	É	upper	C9	71
<CIRC-E>	Ê	upper	CA	72
<DIA-E>	Ë	upper	CB	73
<GRAVE-I>	Ì	upper	CC	78
<ACUTE-I>	Í	upper	CD	75
<CIRC-I>	Î	upper	CE	76
<DIA-I>	Ï	upper	CF	77
<ETH>	Ð	upper	D0	AC
<TILDE_N>	Ñ	upper	D1	69
<GRAVE-O>	Ò	upper	D2	ED
<ACUTE-O>	Ó	upper	D3	EE
<CIRC-O>	Ô	upper	D4	EB
<TILDE_O>	Õ	upper	D5	EF
<DIA-O>	Ö	upper	D6	EC
<multiply>	×	punct	D7	BF
<SLASH-O>	Ø	upper	D8	80
<GRAVE-U>	Ù	upper	D9	E0
<ACUTE-U>	Ú	upper	DA	FE
<CIRC-U>	Û	upper	DB	DD
<DIA-U>	Ü	upper	DC	FC
<ACUTE-Y>	Ý	upper	DD	AD
<THORN>	Þ	upper	DE	8E

Symbolic names	Glyphs	Class(es)	ASCII	EBCDIC
<sharp-s>	ß	lower	DF	59
<grave-a>	à	lower	E0	44
<acute-a>	á	lower	E1	45
<circ-a>	â	lower	E2	42
<tilde-a>	ã	lower	E3	46
<dia-a>	ä	lower	E4	43
<ring-a>	å	lower	E5	47
<ae>	æ	lower	E6	9C
<cedil-c>	ç	lower	E7	48
<grave-e>	è	lower	E8	54
<acute-e>	é	lower	E9	51
<circ-e>	ê	lower	EA	52
<dia-e>	ë	lower	EB	53
<grave-i>	ì	lower	EC	58
<acute-i>	í	lower	ED	55
<circ-i>	î	lower	EE	56
<dia-i>	ï	lower	EF	57
<eth>	ð	lower	F0	8C
<tilde-n>	ñ	lower	F1	49
<grave-o>	ò	lower	F2	CD
<acute-o>	ó	lower	F3	CE
<circ-o>	ô	lower	F4	CB
<tilde-o>	õ	lower	F5	CF
<dia-o>	ö	lower	F6	CC
<divide>	÷	punct	F7	E1
<slash-o>	ø	lower	F8	70
<grave-u>	ù	lower	F9	C0
<acute-u>	ú	lower	FA	DE
<circ-u>	û	lower	FB	DB
<dia-u>	ü	lower	FC	DC
<acute-y>	ý	lower	FD	8D
<thorn>	þ	lower	FE	AE
<dia-y>	ÿ	lower	FF	DF

The other classes are defined as follows:

<code>alpha</code>	The character belongs to the <code>upper</code> or <code>lower</code> class.
<code>alnum</code>	The character belongs to the <code>alpha</code> or <code>digit</code> class.
<code>print</code>	The character belongs to the <code>alnum</code> or <code>punct</code> class or the character is the <code>&lt;space&gt;</code> character.
<code>graph</code>	The character belongs to the <code>alnum</code> or <code>punct</code> class.

The `tolower` and `toupper` mappings behave as usual: `<XYZ>` becomes `<xyz>` and `<xyz>` becomes `<XYZ>`.

#### LC\_COLLATE

Only the characters in the 7-bit code as well as the German umlaut characters (ä, ö, etc.) are taken into account for the sort order. This is the same as under UNIX. The umlauts are considered to be equal to their base vowel; The umlauts follow their corresponding base vowels in their secondary weighting.

The 'ß' character has the ASCII value X'DF' (EBCDIC: X'59').

Otherwise the order is the same as the order in the ASCII character set.

#### LC\_NUMERIC

```
decimal_point:  ","
thousand_sep:  "."
grouping:      0;0
```

#### LC\_TIME

German is used for the days of the week and the months of the year.

The abbreviations for the days of the week are: So, Mo, Di, Mi, Do, Fr, Sa.

The abbreviations for the months of the year are: Jan, Feb, Mär, Apr, Mai, Jun, Jul, Aug, Sep, Okt, Nov, Dez.

```
am_pm: "AM", "PM"
```

```
Time and date representation (%c) d_t_fmt: "%a %d.%h.%Y, %T, %Z"
```

```
Date representation (%x) d_fmt: "%d.%m.%y"
```

```
Time representation (%X) t_fmt: "%T %Z"
```

```
12 hour clock time representation (%X) t_fmt_ampm: "%T: %M:%S %p"
```

```
time_fmt: "%H.%M:%S"
```

```
day_fmt: "%&d.%m"
```

```
full_day: "%a %e.%b"
```

```
ar_date: "%b %d %H:%M %Y"
```

```
last_date: "%a %e.%b %H:%M"
```

```
ls_date: "%h %e %H:%M"
```

```
ls_date2: "%h %e %Y"
```

```
ps_date: "%d.%b"
```

```
su_date: "%d.%m %H:%M"
```

```
tar_date: "%e.%b %H:%M %Y"
```

```
diff_date: "%a %e.%b.%Y, %T"
```

## LC\_MESSAGES

```
yesstring      "ja"
nostr          "nein"
quitstr       "abbrechen"
noexpr        "^[\nN]"
yesexpr       "^[\jJ]"
quitexpr      "^[\aA]"
```

## LC\_MONETARY

Element	De.EDF04F	De.EDF04F@euro
int_curr_symbol	"DEM"	"EUR"
currency_symbol	"DM"	"?"
mon_decimal_point	","	","
mon_thousands_sep	."	."
mon_grouping	3;3	3;3
positive_sign	""	""
negativ_sign	"_"	"_"
int_frac_digits	2	2
frac_digits	2	2
p_cs_precedes	0	0
p_sep_by_space	1	1
n_cs_precedes	0	0
n_sep_by_space	1	1
p_sign_posn	1	1
n_sign_posn	1	1

## User-specific locales

Users can also define their own locales.

The CRTE library `$.SYSLNK.CRTE` provides two source program elements (type S) with the names `USLOCC` and `USLOCA` for this purpose. `USLOCC` is a C source program; `USLOCA` is an Assembler source program. The two source programs are equally effective at generating user-specific locales.

The source programs define the data for the individual locale categories and are preset with the data of the C locale (see section "POSIX or C locale" on page 56 ). This data can be changed to the desired values.

The following changes must also be made in the source programs:

- An address table with the name `USERLOC` is defined in the source programs. This name must be changed to one selected by the user. The name must be a valid entry name.
- This can be done in the C source program by simply changing the name `USERLOC` with a `#define` statement. In the Assembler source program, the name `USERLOC` must be modified in the definition line of the table and in the `ENTRY` statement.
- The name modified by the user can then be used in a call to `setlocale()` as the *locale* argument to identify the user-specific locale.

The modified source programs can be compiled or assembled with the C/C++ compiler or with the Assembler (also `ASSGEN`). If the module is stored in a PLAM library other than `$.SYSLNK.CRTE`, this library must be assigned with the following `ADD-FILE-LINK` command before the C program is started:

```
/ADD-FILE-LINK LINK-NAME=IC@LOCAL,FILE-NAME=library
```

## Environment variables

The environment variables (also called shell variables) described in this section affect the operation of commands, functions and applications. There are other environment variables that are of interest only to specific commands (refer to the keyword "shell variables" in the manual "POSIX Commands (BS2000/OSD)"). When a process begins execution, an array of strings called the **environment** is made available by the `exec` functions (see `exec`). The following external variable points to this vector:

```
extern char **environ;
```

In accordance with the XPG4 Version 2 standard, these strings have the form "*name=value*", e.g. "PATH=/sbin:/usr/sbin".

Applications may also define their own environment variables, provided the naming conventions are complied with (see manual "POSIX Commands (BS2000/OSD)").

### Supplying the environment variables with default values from BS2000/OSD

You can supply environment variables with default values from within BS2000 by defining an SDF-P variable with the name `SYSPOSIX` as a structure (see the manual "SDF-P. Programming in the Command Language"). When the value of the variable `SYSPOSIX.name` is *value*, the string "*name=value*" is written to the global data area of the program; however, only variables of the type 'string' are taken into account.

The SDF-P variable structure can be declared via the `Scope` parameter as a procedure or a task. Task variables are always found; procedural variables may potentially overwrite the task variables.

Only uppercase letters may be used for variable names at the BS2000 command level. Hyphens in the names of SDF-P variables are converted to underscores, e.g. `SYSPOSIX.LC-NAME` would be converted to the string "`LC_NAME=...`".

### Environment variables for internationalization

An internationalized program makes no fixed assumptions about its runtime environment. It determines its specific runtime environment from environment variables.

For example, the environment for displaying outputs is determined from the environment variables `LANG` and `LC_xxx`, while the functions for processing message catalogs interpret the `NLSPATH` environment variable.

The following environment variables are supported for internationalization:

**LANG** Determines the locale category for native language, local customs and coded character set in the absence of `LC_ALL` and other environment variables. `LANG` can be used by applications to determine the language and format for error messages, collating sequences, date formats, and so forth. The value of this environment variable has the form:

```
LANG=language[_territory[_codeset]]
```

For example, a user from Austria who speaks German and is using a terminal with the ISO 8859/1 character set would set the `LANG` variable to the following value:

```
LANG=De_A.88591
```

This enables a user to find the appropriate message catalogs, assuming that they exist.

Specific language operations are initialized at runtime by calling the `setlocale()` function. Normally, the user's language requirements, as specified by the setting of `LANG`, are bound to a program's locale in a subsequent invocation of `setlocale()` as follows:

```
setlocale (LC_ALL, "");
```

**LC\_ALL** On X/Open systems, this form of a `setlocale()` call is defined to initialize the program locale from the associated environment variables. `LC_ALL` addresses the program's entire locale, and `LANG` provides the necessary defaults if any of the category-specific variables are not set or are set to the empty string.

**LC\_COLLATE** This category specifies the collation sequence to be used. The related information is stored in a database that is created by the `colltbl()` command. This environment variable affects `strcoll()` and `strxfrm()`.

**LC\_CTYPE** This category determines character classification, case conversion, and the size of multi-byte characters. The related information is stored in a database that is created by the `chrtbl()` command. The default definition for C corresponds to the 7-bit codeset. This environment variable is used by `ctype()`, `mbchar()` and several commands, e.g. `cat`, `ed`, `ls` and `vi`.

**LC\_MESSAGES** This category determines the language of the message catalog used. For example, an application may have one message catalog with French messages and another containing messages in German.

**LC\_MONETARY** Specifies the currency symbols and separators for a specific environment. This environment variable is used by `localeconv()`.



- `LC_NUMERIC` This category defines the separators for decimal places and thousands. The environment variable is used by `localeconv()`, `printf()` and `strtod()`.
- `LC_TIME` This category specifies the date and time formats.
- `NLSPATH` The environment variable `NLSPATH` returns the location of message catalogs in the form of a search path as well as the naming conventions associated with the message catalogs. For example:

```
NLSPATH=/nlslib/%L/%N.cat:/nlslib/%N/%L
```

The metacharacter `%` indicates a substitution field, where `%L` is replaced by the current setting of the environment variable `LANG` (see below) and `%N` is replaced by the value of the *name* parameter passed to `catopen()`. In the example above, `catopen()` looks first in `/nlslib/$LANG/name.cat` and then in `/nlslib/name/$LANG` for the specified message catalog.

`NLSPATH` is usually set system-wide (e.g. in `/etc/profile`) and therefore makes the location and naming conventions for message catalogs transparent to programs as well as the user.

The complete set of metacharacters include the following symbols:

Meta-character	Meaning
<code>%N</code>	Value of the name parameter passed to <code>catopen()</code>
<code>%L</code>	Value of <code>LANG</code>
<code>%l</code>	Value of the language element from <code>LANG</code>
<code>%t</code>	Value of the territory element from <code>LANG</code>
<code>%C</code>	Value of the codeset element from <code>LANG</code>
<code>%%</code>	A single <code>%</code> character

The behavior of the language information function `nls_langinfo()` is likewise affected by the values set for these environment variables (see also `langinfo.h`).

`LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC` and `LC_TIME` are defined to accept an additional field *@modifier*, which allows the user to select a specific instance of localization data within a single category (for example, for selecting the dictionary as opposed to the character ordering of data). The syntax for these environment variables is thus defined as:

```
[language[_territory[.codeset]][@modifier]]
```

For example, if a user wants to interact with the system in French, but needs to sort German text files, `LANG` and `LC_COLLATE` could be defined as:

```
LANG=Fr_FR
LC_COLLATE=De_DE
```

This could be extended to select dictionary collation (for example) by use of the *@modifier* field:

```
LC_COLLATE=De_DE@dict
```

These values are linked to a program's locale at runtime by calling `setlocale()`.

## File processing

When the C library functions are used with POSIX functionality, it is basically possible to access both POSIX or UFS files and BS2000 files. The compiler environment also provides explicit 64-bit functions and types in addition to the 32-bit functions and types. The 64-bit interface needs to be used to be able to process files > 2 GB via NFS. File systems that contain files larger than 2 GB can be processed with NFS V3.0. See also the section "Scope of the supported C library" on page 18.

In the following section, the file processing functions are classified into different groups, depending on whether they can process both POSIX and BS2000 files or only POSIX files. Functions that process only POSIX files set `errno` explicitly if a BS2000 file was specified instead of a POSIX file.

The initial classification into function groups is followed by a description of POSIX file processing and some special features of BS2000 file processing, which are discussed further below.

**Functions for POSIX and BS2000 files**

The C library functions listed in the following table can process both POSIX as well as BS2000 files.

btowc	fstatvfs()	open()	vprintf()
creat()	ftell()	perror()	vsprintf
clearerr()	ftruncate()	printf()	vwprintf
close()	fwide	putc()	wcrtomb
creat()	fwprintf	putchar()	wcscat
fclose()	fwscanf	puts()	wcschr
fcntl()	fwrite()	putw()	wcscmp
fdopen()	getc()	putwc()	wscoll
feof()	getchar()	putwchar()	wscpy
ferror()	getdents()	read()	wscspn
fflush()	getrlimit()	readdir()	wcsftime
fgetc()	gets()	remove()	wcscat
fgetwc	getw()	rename()	wcslen
fgetws	getwc()	rewind()	wcncat
fgetpos()	getwchar()	scanf()	wcncmp
fgets()	iswalnum	setbuf()	wcncpy
fgetwc()	iswalpha	setrlimit()	wcspbrk
fgetws()	iswcntrl	setvbuf()	wcsrchr
fileno()	iswctype	swscanf	wcsrtombs
fopen()	iswdigit	swprintf	wcsspn
freopen()	iswgraph	stat()	wcsstr
fprintf()	iswlower	statvfs()	wctob
fputc()	iswprint	tmpfile()	wcstod
fputs()	iswpunct	tmpnam()	wcstok
fputwc()	iswspace	towctrans	wcstol
fputws()	iswupper	tolower	wcstoul
fread()	iswxdigit	toupper	wcsxfrm
freopen()	lockf()	truncate()	wctrans
fscanf()	lseek()	ungetc()	wctype
fseek()	lstat()	ungetwc()	write()
fsetpos()	mktemp()	unlink()	
fstat()	mmap()	vfprintf()	

### Functions that reject BS2000 files

The following functions process only POSIX files (see also the manual "POSIX Basics (BS2000/OSD)"). All of these functions - except `sync()` - set `errno` to `EINVAL` if an attempt is made to access BS2000 files.

<code>access()</code>	<code>fpathconf()</code>	<code>symlink()</code>	<code>tcgetpgrp()</code>
<code>chmod()</code>	<code>fsync()</code>	<code>sync()</code>	<code>tcsendbreak()</code>
<code>chown()</code>	<code>isatty()</code>	<code>sysfs() *</code>	<code>tcsetattr()</code>
<code>dup()</code>	<code>link()</code>	<code>tempnam() *</code>	<code>tcsetpgrp()</code>
<code>dup2()</code>	<code>mknod()</code>	<code>tcdrain()</code>	<code>utime()</code>
<code>fchmod()</code>	<code>mkfifo()</code>	<code>tcflow()</code>	
<code>fchown()</code>	<code>pathconf()</code>	<code>tcflush()</code>	
<code>fcntl()</code>	<code>readlink()</code>	<code>tcgetattr()</code>	

\*) `sync()` has no effect on BS2000 files.  
`tempnam()` sets `errno` to `EINVAL` if `PROGRAM-ENVIRONMENT` is not set.

When standard I/O streams are used, these functions are subject to restrictions if the streams were associated with the BS2000 SYSFILE management files by the POSIX subsystem (see section "Streams" on page 77).

### Functions that access only POSIX files

The functions in the following list will always access POSIX files, regardless of which functionality was selected (POSIX or BS2000), since they do not exist as BS2000 functions.

<code>chdir()</code>	<code>getpass()</code>	<code>popen()</code>	<code>telldir()</code>
<code>chroot()</code>	<code>mkdir()</code>	<code>readdir()</code>	<code>ttyname()</code>
<code>closedir()</code>	<code>opendir()</code>	<code>rewinddir()</code>	<code>umount()</code>
<code>ftw()</code>	<code>pclose()</code>	<code>rmdir()</code>	<code>umask()</code>
<code>getcwd()</code>	<code>pipe()</code>	<code>seekdir()</code>	

## Streams

A stream is associated with an external file (which may be a physical device) by **opening** a file. This is also the case when a new file is **created**. Creating an existing file causes its former contents to be discarded if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a **file position indicator** associated with the stream is positioned at the start (byte number 0) of the file, unless the file is opened with append mode, in which case the file position indicator may be initially positioned at the beginning or end of the file. The file position indicator facilitates subsequent reads, writes and positioning requests on the file. All input takes place as if bytes were read by successive calls to `fgetc`; all output takes place as if bytes were written by successive calls to `fputc`.

### Buffering streams

When a stream is **unbuffered**, bytes are passed through to the system immediately. Otherwise, bytes may be accumulated and transmitted as a block. When a stream is **fully buffered**, bytes are transmitted as a block when the buffer is filled. When a stream is **line buffered**, bytes are transmitted as a block when a newline byte is encountered. Furthermore, bytes transmitted as a block when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line-buffered stream that requires the transmission of bytes. Support for these characteristics can be initiated and affected via `setbuf` and `setvbuf`.

### Disassociating a file from a stream

A file may be disassociated from a controlling stream by **closing** the file. Output streams are **flushed** (i.e. the unwritten buffer contents are transmitted) before the stream is disassociated from the file. The value of a pointer to a `FILE` object is indeterminate after the associated file is closed (including the standard streams).

A file may be subsequently reopened by the same or another program, and its contents may be reclaimed or modified (if the file can be repositioned at its start). If the `main` function returns to its original caller, or if the `exit` function is called, all output streams are flushed and all open files are closed before program termination. Other methods of program termination, such as calling `abort`, may not close all files properly.

The address of the `FILE` object used to control a stream may be significant; a copy of a `FILE` object may not necessarily serve in place of the original.

## Standard I/O streams

At program startup, three streams are predefined and need not be opened explicitly:

- **standard input**, for reading conventional input
- **standard output**, for writing conventional output
- **standard error**, for writing diagnostic output

When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device. Otherwise, the streams are line buffered.

Depending on which functionality is selected (see section "Selecting the file system and the system environment" on page 42 ), standard I/O streams may be associated with POSIX or BS2000 files.

The following association is created when accessing the DMS:

<code>stdin</code>	<code>SYSDTA</code>
<code>stdout, stderr</code>	<code>SYSOUT</code>

In this case, behavior is compatible with the earlier versions of the C runtime system (see also section "BS2000 system files" on page 82).

Functions that only use POSIX functionality cannot be applied on `stdin`, `stdout` or `stderr` in this case.

When the POSIX file system is accessed, the standard I/O streams are associated with `/dev/tty` (see also section "Associating the I/O streams" on page 42)

In batch mode, the association is always with `SYSFILE`, since no terminal is present. In child processes, I/O streams that are associated with `SYSFILE` can no longer be accessed, even if the association was made via POSIX.

If the association of standard I/O streams is controlled by selecting POSIX functionality with environment variables, the association can be affected by changing the variables with `putenv()`: when a program is initiated with one of the `exec` functions, the environment variables are reevaluated at C runtime initialization, and the corresponding associations are made for the program started with the `exec` function.

## Interaction of file descriptors and streams

An open file description may be accessed through a file descriptor that is created by `open()` or `pipe()` or through a stream created by the `fopen()` or `popen()` functions. A file descriptor or a stream is called a **handle** on (or a link to) the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by explicit user action, without affecting the underlying open file description. Some of the functions to create them include `fcntl()`, `dup()`, `fdopen()`, `fileno()` and `fork()`. The handles can be destroyed by at least `fclose()`, `close()` and the `exec` functions.

A file descriptor that is never used in an operation that could affect the file offset (for example, `read()`, `write()` or `lseek()`) is not considered a handle, but could become one (as a consequence of `fdopen()`, `dup()` or `fork()` for example). This exception does not include the file descriptor underlying a stream, whether created with `fopen()` or `fdopen()`, so long as it is not used directly by the application to affect the file offset. The `read()` and `write()` functions implicitly affect the file offset; `lseek()` affects it explicitly.

The result of function calls involving only one handle (the **active handle**) are described in the reference section. If two or more handles are used, however, and one of them is a stream, their actions must be coordinated as described in the section “Actions” on page 80.

A handle which is a stream is considered to be closed when either an `fclose()` or `freopen()` is executed on it (the result of `freopen()` is a new stream, which cannot be a handle on the same open file description as its previous value), or when the process owning that stream terminates with `exit()` or `abort()`. A file descriptor is closed by `close()`, `_exit()` or one of the `exec` functions when `FD_CLOEXEC` is set on that file descriptor.

For a handle to become the active handle, the actions below must be performed between the last use of the handle (the current active handle) and the first use of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle must be suspended until it again becomes the active file handle. If a stream function calls an underlying function that affects the file offset, the calling stream function will be considered to affect the file offset. The underlying functions involved are described below.

The handles need not be in the same process for these rules to apply.

## Actions

If a handle is still open after the actions required below are taken, the application can close it.

- No action is required for the **first handle** if one of the following conditions apply:
  - The handle is a file descriptor or an unbuffered stream.
  - The only further action to be performed on any handle is to close it.
  - The handle is a stream which is line buffered, and the last action has the same effect on the associated file as `fputs()`.
  - The handle is a stream open for reading and `feof()` is `TRUE`.
- If none of the conditions listed above apply, either an `fflush()` must occur or the stream must be closed in the following cases:
  - If it is a stream which is open for writing or appending, but not also open for reading.
  - If the stream is open with a mode that allows reading, and the underlying open file description refers to a device that is capable of seeking.
- In all other cases, the result is undefined.

The following applies to the **second** handle:

If any previous active handle has been used by a function that explicitly changed the file offset, except as required above for the first handle, the application must perform an `lseek()` or `fseek()` (as appropriate to the type of handle) to an appropriate location.

If the active handle ceases to be accessible before the requirements on the first handle have been met, the state of the open file description becomes undefined. This could occur during the execution of functions such as a `fork()` or `_exit()`.

The `exec` functions ensure that all streams which are open at the time they are called are made inaccessible, independent of which streams or file descriptors are available to the new process image.

If the above rules are followed, regardless of the sequence of handles used, the C runtime library will ensure that an application, even one consisting of several processes, will always yield correct results, i.e. that no data will be lost or duplicated when writing, that all data will be written in order (except when the order is changed as requested by seeks), and that all data will be found when reading sequentially. It does not matter which order the handles are used. If the rules above are not followed, the result is undefined.

See also the manual "POSIX Basics (BS2000/OSD)".



## Support for file systems in ASCII

File systems located on machines that normally use the ASCII character set instead of EBCDIC can also be mounted in the POSIX file system. To facilitate this interaction, an automatic conversion is performed for text files in the C library.

The following conditions must be satisfied for the automatic conversion to occur:

- The file has been opened with `fopen()`, `fdopen()` or `freopen()` and is thus associated with a stream.
- Mode "b" for binary must not be specified.
- `fstat()` does not return the BS2000 file system bit.
- The environment variable `IO_CONVERSION` has the value "YES".

The functions `ascii_to_ebcdic()` and `ebcdic_to_ascii()` are provided for cases in which no automatic conversion occurs.

## BS2000 file processing

Besides POSIX files, the following types of files can be processed with the I/O functions of the Common Runtime Environment CRTE:

- the BS2000 system files SYSDTA, SYSOUT and SYSLST
- cataloged disk files with access methods SAM, ISAM and PAM
- temporary PAM files (INCORE).

In C-BS2000, a distinction is made between binary files and text files on one hand, and between stream-oriented and record-oriented I/O on the other.

The following table shows the possible combinations in which the various file types can be processed:

	Text file Stream I/O	Binary file Stream I/O	Binary file Record I/O
System files	X		
INCORE		X	
SAM	X	X	X
ISAM	X		X
PAM		X	X

A maximum of 256 files (including `stdin`, `stdout` and `stderr`) may be open at one time.

## BS2000 system files

The system files in BS2000 correspond to streams. The functionality of these files is therefore relevant for any function that is called with BS2000 functionality.

### SYSDTA

A C program can use SYSDTA as follows:

- An open function (`fopen()`, `freopen()`, `open()`) is used to open a file with the name "`(SYSDTA)`" or "`(SYSTEM)`" for reading. The file pointer returned by the open function then serves as an argument for a subsequent input function.

#### *Example*

```
FILE *fp;
fp = fopen("(SYSDTA)", "r");
fgetc(fp);
```

- For input functions, the file pointer `stdin` or the file descriptor `0` is specified as the file argument.

#### *Examples*

```
fgetc(stdin);
read(0, buf, n);
```

- Input functions that read from `stdin` by default (e.g. `scanf()`, `getchar()`, `gets()`) are used.

If the input is to be obtained from a cataloged file instead of the terminal, this can be done by two methods:

1. If a parameter line was requested with `PARAMETER-PROMPTING=YES` (specified in the `RUNTIME-OPTIONS` compiler option), this parameter line can be used to redirect the standard input (file pointer `stdin` or file descriptor `0`) to a catalog file (see also the C and C++ User Guides).

The redirection **does not** affect files that were opened with the name "`(SYSDTA)`" or "`(SYSTEM)`". Input from any file with either of these names will still be expected from the terminal.

2. By using the command `ASSIGN-SYSDTA filename` before program startup.

This causes input data to be expected from the assigned file for all input functions. The following must be observed when using the `ASSIGN-SYSDTA` command:

- After the program is executed, the internal record pointer will be positioned after the last record that was read or at the end of the file. If the file is to be read again from the beginning in a subsequent program run, a new `ASSIGN-SYSDTA` command must be issued before the program is started.

- If `PARAMETER-PROMPTING=YES` was selected (in the `RUNTIME-OPTIONS` option), the first record of the assigned file is interpreted as a parameter line for the `main` function.

### Note

If no other end criterion for reading was declared in the C program, the EOF condition for inputs at the terminal can be forced by pressing the K2 key and entering the EOF and `RESUME-PROGRAM` commands.

## SYSOUT

A C program can use `SYSOUT` as follows:

- An open function (`fopen()`, `freopen()`, `open()`) is used to open a file with the name "`(SYSOUT)`" or "`(SYSTEM)`" for writing. The file pointer returned by the open function then serves as an argument for a subsequent output function.

### Example

```
FILE *fp;
fp = fopen("(SYSTEM)", "w");
fputc(fp);
```

- For output functions, the file pointer `stdout` or the file descriptor 1 is specified as the file argument.

### Examples

```
fputc(stdout);
write(1, buf, n);
```

- The file pointer `stderr` or the file descriptor 2 may also be specified as the file argument for output functions.
- Output functions that write to `stdout/stderr` by default (e.g. `printf()`, `puts()`, `putchar()` or `perror()`) are used.

If a parameter line was requested with `PARAMETER-PROMPTING=YES` (specified in the `RUNTIME-OPTIONS` compiler option), this parameter line can be used to redirect the standard output (file pointer `stdout` or file descriptor 1) and the standard error output (file pointer `stderr` or file descriptor 2) to a catalog file (see also `C and C++ User Guides`).

The redirection **does not** affect files that were opened with the name "`(SYSOUT)`" or "`(SYSTEM)`".

## SYSLST

A C program can use SYSLST as follows:

- An open function (`fopen()`, `freopen()`, `open()`) is used to open a file with the name "(SYSLST)" for writing. The file pointer returned by the open function then serves as an argument for a subsequent output function.

### Example

```
FILE *fp;
fp = fopen("(SYSLST)", "w");
fprintf(fp, "\t TEXT \n");
```

- If a parameter line was requested with `PARAMETER-PROMPTING=YES` (specified in the `RUNTIME-OPTIONS` compiler option), this parameter line can be used to redirect the standard output or standard error to SYSLST (see also the C and C++ User Guides).

The redirection **does not** affect files that were opened with the name "(SYSOUT)".

By default, SYSLST files are printed out automatically at the end of a task (LOGOFF).

If the data is to be output to a catalog file instead of being automatically printed, SYSLST must be redirected before the program is executed. This can be done with the command `ASSIGN-SYSLST filename`.

## White-space characters

The control characters for white space and the backspace control character '`\b`' (see table below) are interpreted by all output functions which write to text files and which receive such control characters, either as character constants (starting with `\`) or as numerical EBCDIC values, as arguments. The decimal and hexadecimal values of the control characters can be found in the C and C++ User Guides (EBCDIC table).

Key to the following table:

X The control character is converted to its appropriate effect.

blank The control character is written to the file as a text character (EBCDIC value).

Output medium	<code>\ n</code>	<code>\ t</code>	<code>\ f</code>	<code>\ v</code>	<code>\ r</code>	<code>\ b</code>
SAM/ISAM	X	X				
SYSOUT/SYSTEM	X	X	X			
SYSLST	X	X	X	X	X	X

**Tab character (\t)**

The tab character is converted to the appropriate number of spaces. Tab stops are set 8 columns apart (1, 9, 17, ...). Spaces are also substituted for the tab character on input.

In the case of SAM and ISAM files, the tab character is expanded to spaces by default only in the KERNIGHAN-RITCHIE compilation mode, not in the ANSI mode (see also `fopen()` and `freopen()`).

**Line feed (\n)**

The newline character is converted to a change of line (change of record). Subsequent read functions will then return a newline character for a change of record.

**Page feed (\f)**

SYSLST: A page feed is executed, and subsequent data is output on a new page.

SYSOUT, SYSTEM for writing: The message

please acknowledge is output on the terminal.

**Vertical tab (\v)**

An appropriate number of blank lines is output to reach the next line tab position. These tab positions are 8 lines apart (1, 9, 17, ...).

**Carriage return (\r)**

The cursor is returned to the start of the current line without a line feed, i.e. subsequent data is written to the same line. This enables characters to be underlined, for example.

**Backspace (\b)**

The next character is written to the position of the previous character. This allows a letter to be provided with an accent, for example. Strictly speaking, `\b` is not a white-space character (see `isspace()`) but a control character (see `iscntrl()`).

The use of `\r` and `\b` is only meaningful for printers with overwrite capabilities.

**Cataloged disk files (SAM, ISAM, PAM)**

C programs process cataloged disk files by means of the SAM, ISAM and PAM access methods.

When an existing file is opened, the access method and other file attributes are taken from the catalog entry.

When a new file is created, default values of the C runtime library are assigned in accordance with the type of C file (binary file, text file, stream-oriented or record-oriented I/O). These values can be changed with an `ADD-FILE-LINK` command before the program is called. To do this, a link name ("`link=linkname`") must be specified with the open functions (`open()`, `creat()`, `fopen()`, `freopen()`), and this link name must be associated with the name of the cataloged file in the `ADD-FILE-LINK` command.

Not all possible file attributes can be combined. Combinations that are not required for performance reasons are not supported by the I/O functions of the C runtime library.

The following sections provide information on

- the default values and possible modifications of the file attributes
- the K and NK block formats
- stream-oriented and record-oriented processing of disk files.

### Default values and possible modifications for file attributes

The I/O functions of the C runtime library can process disk files with the file attributes listed in the following tables. The default attributes inserted by the runtime system when the user does not specify any options in the `ADD-FILE-LINK` command or in the open functions are underlined.

#### Notes on the following tables

- The maximum number of data bytes in the tables indicates the number of characters that can be stored by the C program in a record or block (fixed record length) or the maximum number of characters that can be stored (variable record length).
- The size of the logical block (`BLKSIZE`) varies according to the type and format of the volume:
  - K and NK2 disks: a standard block (2048 bytes) or the integral multiple of a standard block (max. of 16 standard blocks);
  - NK4 disks: a minimum of two standard blocks (4096 bytes) or an integral multiple thereof (2, 4, 6, 8 standard blocks).
- For more information on the block format (`BLKCTRL`) and the maximum number of data bytes, see also section "K and NK block formats" on page 89. It explains, in particular, how overflow blocks can be avoided with NK-ISAM files. Overflow blocks occur if the full length of a transfer unit is utilized when writing records (`RECSIZE = BLKSIZE`).
- In C, the 4-byte record length field in files with variable record length (`RECFORM=V`) is not counted as part of the record data. The maximum number of data bytes is therefore reduced by 4 bytes.

- For files with RECFORM=U, the register in which the length of a record is passed is defined by RECSIZE (RECORD-SIZE parameter in the ADD-FILE-LINK command). This register is predefined (R4) and must not be changed.

FCB-TYPE	REC-FORM	BLKCTRL	BLKSIZE (STD, <i>n</i> )	RECSIZE ( <i>r</i> byte)	Maximum number of data bytes
SAM <sup>1)</sup>	V	PAMKEY	$1 \leq n \leq 16$	$4 \leq r \leq n * 2048 - 4$	RECSIZE - 4
		DATA(2K)	$1 \leq n \leq 16$	$4 \leq r \leq n * 2048 - 16$	RECSIZE - 4
		DATA(4K)	$2 \leq n \leq 16$		
	U	PAMKEY	$1 \leq n \leq 16$		BLKSIZE
		DATA(2K)	$1 \leq n \leq 16$		BLKSIZE - 16
		DATA(4K)	$2 \leq n \leq 16$		
ISAM <sup>2)</sup>	V	PAMKEY	$1 \leq n \leq 16$	$12 \leq r \leq n * 2048$	RECSIZE - 12
		DATA(2K)	$1 \leq n \leq 16$	$12 \leq r \leq n * 2048$	RECSIZE - 12
		DATA(4K)	$2 \leq n \leq 16$		

- 1) SAM files are only created in the KR mode (see also the SOURCE-PROPERTIES option in the C and C++ User Guides) by default. In ANSI mode, ISAM files are created by default.
- 2) The default value for the key position is 5, and the default key length is 8. These values cannot be modified. The user cannot access the keys; they are generated and managed by the C runtime library: when a new ISAM file is created, the first record is assigned the key "00010000", and the key is then incremented in steps of 100 for each further record.

FCB-TYPE	REC-FORM	BLKCTRL	BLKSIZE (STD, <i>n</i> )	RECSIZE ( <i>r</i> byte)	Maximum number of data bytes	
SAM	E	PAMKEY	$1 \leq n \leq 16$	$1 \leq r \leq n * 2048$	RECSIZE	
		DATA(2K)	$1 \leq n \leq 16$	$1 \leq r \leq n * 2048 - 16$	RECSIZE	
		DATA(4K)	$2 \leq n \leq 16$			
	V	PAMKEY	$1 \leq n \leq 16$		$4 \leq r \leq n * 2048 - 4$	RECSIZE - 4
		DATA(2K)	$1 \leq n \leq 16$		$4 \leq r \leq n * 2048 - 16$	RECSIZE - 4
		DATA(4K)	$2 \leq n \leq 16$			
	U	PAMKEY	$1 \leq n \leq 16$			BLKSIZE
		DATA(2K)	$1 \leq n \leq 16$			BLKSIZE - 16
		DATA(4K)	$2 \leq n \leq 16$			

FCB-TYPE	REC-FORM	BLKCTRL	BLKSIZE (STD, <i>n</i> )	RECSIZE ( <i>r</i> byte)	Maximum number of data bytes
PAM		PAMKEY	$1 \leq n \leq 16$		BLKSIZE
		DATA(2K)	$1 \leq n \leq 16$		BLKSIZE - 12
		DATA(4K)	$2 \leq n \leq 16$		
		NO(2K)	$1 \leq n \leq 16$		BLKSIZE
		NO(4K)	$2 \leq n \leq 16$		
<u>SAM</u>	<u>V</u>	PAMKEY	$1 \leq n \leq 16$	$4 \leq r \leq n * 2048 - 4$	RECSIZE - 4
		DATA(2K)	$1 \leq n \leq 16$	$4 \leq r \leq n * 2048 - 16$	RECSIZE - 4
		DATA(4K)	$2 \leq n \leq 16$		
	F	PAMKEY	$1 \leq n \leq 16$	$1 \leq r \leq n * 2048$	RECSIZE
		DATA(2K)	$1 \leq n \leq 16$	$1 \leq r \leq n * 2048 - 16$	RECSIZE
		DATA(4K)	$2 \leq n \leq 16$		
	U	PAMKEY	$1 \leq n \leq 16$		BLKSIZE
		DATA(2K)	$1 \leq n \leq 16$		BLKSIZE - 16
		DATA(4K)	$2 \leq n \leq 16$		
PAM		PAMKEY	$1 \leq n \leq 16$		BLKSIZE
		DATA(2K)	$1 \leq n \leq 16$		BLKSIZE - 12
		DATA(4K)	$2 \leq n \leq 16$		
		NO(2K)	$1 \leq n \leq 16$		BLKSIZE
		NO(4K)	$2 \leq n \leq 16$		
ISAM <sup>1)</sup>	<u>V</u>	PAMKEY	$1 \leq n \leq 16$	$5 \leq r \leq n * 2048$	RECSIZE - 4
		DATA(2K)	$1 \leq n \leq 16$	$5 \leq r \leq n * 2048$	RECSIZE - 4
		DATA(4K)	$2 \leq n \leq 16$		
	F	PAMKEY	$1 \leq n \leq 16$	$1 \leq r \leq n * 2048 - 4$	RECSIZE
		DATA(2K)	$1 \leq n \leq 16$	$1 \leq r \leq n * 2048 - 4$	RECSIZE
		DATA(4K)	$2 \leq n \leq 16$		

- 1) The default attributes for the key position (for record format V = 5 and for F = 1) and key length (8) can be modified to a maximum of 32767 and 255, respectively.

Multiple keys can also be defined (DUP-KEY=Y). The default value is DUP-KEY=N.

In contrast to stream-oriented I/O, the ISAM keys are a part to the record data that is written or read by the C program.



## K and NK block formats

BS2000 supports volumes with different formats:

- **Key** volumes for storing files in which the block control information is maintained in a separate field ("Pamkey") per 2 Kbyte data block. These files have the block format PAMKEY.
- **Non-Key** volumes for files without separate Pamkey fields. The block control information is either omitted (block format NO) or stored in the respective data blocks (block format DATA).

Starting with BS2000/OSD V1.0, NK volumes are distinguished by the minimum size of the transfer unit. NK2 volumes have the old transfer unit (2 Kbytes). NK4 volumes have a transfer unit of 4 Kbytes.

The block format is controlled by the `BLOCK-CONTROL-INFO` operand in the `ADD-FILE-LINK` command:

```
BLOCK-CONTROL-INFO = BY-PROGRAM / BY-CATALOG / WITHIN-DATA-BLOCK / NO /
PAMKEY
```

As of BS2000 V11.0, there are two further operand values for NK-ISAM files:

```
WITHIN-DATA-2K-BLOCK / WITHIN-DATA-4K-BLOCK
```

The `BLOCK-CONTROL-INFO` operand, various file and volume structures, and conversions from the K file format to NK file format are explained in detail in the manual „Introductory Guide to DMS“.

If the `ADD-FILE-LINK` command is not used when creating a new file or `BLOCK-CONTROL-INFO=BY-PROGRAM` is specified, the default values of the C runtime library are used. These values depend on the disk type, on the class 2 option that may be specified by the system administrator, and on the access method:

File organization	CLASS2-OPTION BLKCTRL=NONKEY			
	not specified		specified	
	K disk	NK disk	K disk	NK disk
SAM	PAMKEY	DATA	DATA	DATA
ISAM	PAMKEY	DATA	DATA	DATA
PAM	PAMKEY	NO	NO	NO

## K and NK-ISAM files

ISAM files in K format that use of the maximum record length become longer in NK format than the usable area of the data block. They can be processed in NK format because the DMS creates extensions to the data blocks known as overflow blocks.

The creation of overflow blocks presents the following problems:

- The overflow blocks increase space requirements on the disk and hence the number of I/O operations during file processing.
- The ISAM key must not be in an overflow block under any circumstances.

Overflow blocks can be avoided by ensuring that the longest record in the file is no longer than the area of a logical block that may be used for NK-ISAM files.

### Usable area for records (NK-ISAM files)

The following table can be used to calculate the amount of space available per logical block for records in ISAM files..

File format	RECORD-FORMAT	Maximum usable area
K-ISAM	VARIABLE	BUF-LEN
	FIXED	$BUF-LEN - (s*4)$ where $s$ = number of records per logical block
NK-ISAM	VARIABLE	$BUF-LEN - (n*16) - 12 - (s*2)$ (rounded down to the next number divisible by 4) where $n$ = blocking factor $s$ = number of records per logical block
	FIXED	$BUF-LEN - (n*16) - 12 - (s*2) - (s*4)$ (rounded down to the next number divisible by 4) where $n$ = blocking factor $s$ = number of records per logical block

### Explanation of the formulas

For NK-ISAM files, each PAM page of a logical block contains 16 bytes of administrative information. The logical block also contains a further 12 bytes of administrative information and a record pointer with a length of 2 bytes for each record.

For RECORD-FORMAT=FIXED there is a 4-byte record length field for each record, but this is not included when calculating the record length. Consequently, 4 bytes must be deducted per record in such cases.

**Example: Maximum record length of an NK-ISAM file (fixed record length)**

File definition:

```
/ADD-FILE-LINK . . . ,RECORD-FORMAT=FIXED,BUFFER-LENGTH=STD(SIZE=2),
BLOCK-CONTROL-INFO=WITHIN-DATA-BLOCK
```

maximum record length (according to the formula):

$4096 - (2*16) - 12 - 1*2 - 1*4 = 4046$ , rounded down to the next number divisible by 4: 4044 (bytes).

**Support for the DIV access method**

The new access method DIV (DATA IN VIRTUAL) was introduced with BS2000/OSD V1.0. This access method is specially suitable for processing the unstructured streams that are frequently encountered in C programs (possibly ported from UNIX).

DIV can be used to process NK-PAM files which are located on public disks and contain no administrative information (BLOCK-CONTROL-INFO=NO).

Repeated access to data that has already been read into a "window" by a preceding access operation can lead to a substantial improvement in performance.

Detailed background information on the DIV access method can be found in the manual "DMS Assembler Interface".

Starting with BS2000/OSD V1.0, the C runtime library always uses the DIV access method to perform stream-oriented I/O on NK-PAM files without administrative information. The DIV access method cannot be used with NK-PAM files that were opened for record-oriented I/O.

**Notes on stream-oriented I/O****Binary files (SAM)**

Fixed record length (F) is the default. When a file is closed, the last record is padded with binary zeros (if necessary). If the same file is opened again and data is written at the end of the file, a new record is always started. In other words, the new data is written after the binary zeros.

If a variable record length is used (V or U), new data can be written on a byte-specific basis. The variable record length does, however, result in a loss of performance during seek operations (with `fseek()` and `ftell()` for example).

### Binary files (PAM)

In order to permit byte-specific updating of PAM files (after a close and reopen), the C runtime system writes administrative data at the end of the file. This data is maintained in a consistent state at the time the file is opened and closed. Consequently, it is not possible for different tasks to process a PAM file concurrently if the file is extended by one of the tasks involved.

The C runtime system does not set any locks. If data is modified by several users, inconsistent states might result.

### Text files (SAM, ISAM)

When SAM or ISAM files are processed in update mode, the original record length must not be changed when modifying existing records. This means that a newline character (`\n`) must not be changed to another character, or vice versa.

### Notes on record-oriented I/O

Record-oriented I/O is possible for SAM, ISAM and PAM files.

When the `fopen()` and `freopen()` functions are called, the file must always be opened in binary mode and with the option `type=record`.

I/O functions that read or write characters or strings (up to `\n`) cannot be used for record-oriented input/output.

### Available I/O functions

The following functions are available for processing files with stream I/O:

<code>fopen()</code> , <code>freopen()</code>	Open
<code>fclose()</code>	Close
<code>fread()</code>	Read
<code>fwrite()</code>	Write
<code>fsetpos()</code> , <code>fgetpos()</code>	Set file position to determined location
<code>fseek()</code>	Set file position to start/end of file
<code>rewind()</code>	Set file position to start of file
<code>flocate()</code>	Explicitly positioning in an ISAM file
<code>fdelrec()</code>	Delete a record in an ISAM file

In addition, the following functions for file processing and error handling can be used unchanged:

`feof()`, `ferror()`, `clearerr()`, `unlink()`, `remove()`, `rename()`

All I/O functions not listed here are unavailable for record-oriented input/output and will be rejected with an error return value.

It should be noted, however, that no checks are performed for the two macros `getc()` and `putc()` for performance reasons. If these macros are used on files with record-oriented I/O, the behavior is undefined.

### Processing a file with record-oriented and stream-oriented I/O

Files created for record-oriented I/O can be opened for stream-oriented I/O and vice versa. However, stream-oriented I/O does not support all the file attributes that are possible for record-oriented I/O.

### FCBTYPE of a new file to be created

The FCBTYPE of a new file to be created can be defined as follows:

- Specification in an `ADD-FILE-LINK` command and use of the `LINK` name in the `fopen()` or `freopen()` function
- Specification of the `forg` parameter in the `fopen()` or `freopen()` function:
  - `forg=seq`: a SAM file is created.
  - `forg=key`: an ISAM file is created.

The following restrictions apply to the FCBTYPE of a file and the entries for `fopen()` and `freopen()`:

- If `type=record` is specified, the FCBTYPE of the file must be SAM, PAM or ISAM.
- If `forg=seq` is specified, the FCBTYPE of the file must be SAM or PAM.
- If `forg=key` is specified, the FCBTYPE of the file must be ISAM.
- The append mode "a" is not allowed for ISAM files. The position is determined by the key in the record.

### Multiple keys for ISAM files

By default, multiple keys are not permitted for ISAM files. They may, however, be used if `DUP-KEY=Y` is specified in an `ADD-FILE-LINK` command.

## Temporary PAM files in virtual memory (INCORE files)

If the file name "(INCORE)" is specified with the functions `fopen()`, `freopen()`, or `open()`, a temporary PAM file is created in virtual memory. This file "exists" only for the duration of a program run.

INCORE files must be opened for writing before they can be accessed for reading (see also `fopen()`, `freopen()`, `open()`).

INCORE files are processed as binary files.

## General terminal interface

This section describes a general terminal interface that is provided to control serial communications ports. These are locally connected asynchronous lines.

On BS2000 block terminals, support for this interface is subject to certain restrictions.

## Opening a terminal device file

When a terminal device file is opened, it normally causes the process to wait until a connection is established. In practice, application programs seldom open these files; they are opened by special programs and then become the standard input, standard output, and standard error of applications.

As described in `open()`, opening a terminal device file with the `O_NONBLOCK` flag clear causes the process to block until the terminal device is ready and available. If the `CLOCAL` mode is not set, this implies waiting until a connection is established. If `CLOCAL` mode is set in the terminal, or the `O_NONBLOCK` flag is specified when calling `open()`, the `open()` function returns a file descriptor without waiting for a connection to be established.

## Process groups

A terminal may have a foreground process group associated with it. This foreground process group plays a special role in handling signal-generating input characters, as described in section "Special characters" on page 99.

A terminal's foreground process group may be set or examined by a process, assuming the permission requirements in this section are met; see `tcgetpgrp()` and `tcsetpgrp()`. The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the current process group. For further details, see section "Terminal access control" on page 95.

## The controlling terminal

A terminal may belong to a process as its controlling terminal. Every process of a session that has a controlling terminal has the same controlling terminal. A terminal may be the controlling terminal for at most one session. The first open terminal device file is reserved as the controlling terminal for a session by the session leader. If a session leader has no controlling terminal and opens a terminal device file (without the `O_NOCTTY` bit set) that is not already associated with a session (see `open()`), the terminal can become the controlling terminal of the session leader. If a process which is not a session leader opens a terminal file, or if the `O_NOCTTY` option is used when calling `open()`, then that terminal does not become the controlling terminal of the process. When a controlling terminal becomes associated with a session, its foreground process group is set to the process group of the session leader.

The controlling terminal is inherited by a child process by means of a `fork` call. A process relinquishes its controlling terminal when it creates a new session with the `setsid()` function, or when all file descriptors associated with the controlling terminal have been closed..

When a controlling process terminates, the controlling terminal is disassociated from the current session, allowing it to be acquired by a new session leader. Subsequent access to the terminal by other processes in the earlier session may be denied, with attempts to access the terminal treated as if a modem disconnect had been sensed.

## Terminal access control

If a process is in the foreground process group of its controlling terminal, it will be allowed to read from this terminal, as described in the section "Input processing and reading data" on page 96 . For those implementations that support job control, any attempt by a process in a background process group to read from its controlling terminal will cause its process group to be sent a `SIGTTIN` signal unless one of the following special cases applies:

- The reading process is ignoring or blocking the `SIGTTIN` signal.
- The process group of the reading process is orphaned.

In the above cases, the `read()` function returns `-1`, with `errno` set to `EIO`, and no signal is sent. The default action of the `SIGTTIN` signal is to stop the process to which it is sent (see also `signal.h`).

If a process is in the foreground process group of its controlling terminal, then write operations are allowed as described in the section "Writing data and output processing" on page 99. Attempts by a process in a background process group to write to its controlling terminal will cause the process group to be sent a SIGTTOU signal unless one of the following special cases apply:

- If TOSTOP is not set, or if TOSTOP is set and the process is ignoring or blocking the SIGTTOU signal, then the process is allowed to write to the terminal and the SIGTTOU signal is not sent.
- If TOSTOP is set and the process group of the writing process is orphaned, and the writing process is not blocking SIGTTOU, the write() function returns -1, with errno set to EIO, and no signal is sent.

Certain function calls that set terminal parameters are treated in the same way as the write() function, except that TOSTOP is ignored; that is, the effect is identical to that of an attempt to write to the terminal when TOSTOP is set (see also section "Local modes" on page 107, tcdrain(), tcflow(), tcflush(), tcsendbreak() and tcsetattr()).

## Input processing and reading data

A terminal associated with a special file may operate in full-duplex mode, so that input characters may be entered at any time, even while output is occurring. In the POSIX subsystem, full-duplex mode for terminals is simulated by TIAM.

Each special file of a terminal is associated with an **input queue** in which incoming data is stored by the system before being read by a process. The input is lost if the input queues of the system are full or if any input line exceeds the maximum number of bytes permitted for input (as defined by {MAX\_INPUT}; see `limits.h`). {MAX\_INPUT} must be greater than or equal to {\_POSIX\_MAX\_CANON}. This value can be queried with `pathconf()`.

Two general types of input processing are available, depending on whether the special file associated with the terminal device is in **canonical mode** or **non-canonical mode**. These modes are described in the next two sections on "Canonical mode input processing" and "Non-canonical mode input processing", respectively. Additionally, input characters are processed according to the settings of the `c_iflag` (see section "Input modes" on page 102) and `c_lflag` (see section "Local modes" on page 107) components. Such processing can include local echoing, which in general means that input characters are immediately transmitted back to the terminal when they are received from the terminal. This is particularly useful for terminals that operate in full-duplex mode.



If the `O_NONBLOCK` flag is clear, then `read` requests block until data is available or a signal has been received. If the `O_NONBLOCK` flag is set, then the `read` request is completed, without blocking, in one of three ways:

- If there is enough data available to satisfy the entire request, the `read()` function completes successfully and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, the `read()` function completes successfully, having read as much data as possible, and returns the number of bytes actually read.
- If there is no data available, the `read()` function returns `-1`, with `errno` set to `EAGAIN`.

When data is available depends on whether the input processing mode is canonical or non-canonical. The following sections, "Canonical mode input processing" and "Non-canonical mode input processing", describe each of these input processing modes.

## Canonical mode input processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a newline character (LF), an end-of-file character (EOF), or an end-of-line character (EOL). For more information on EOF and EOL, see the section "Special characters" on page 99. This means that a reading program will be suspended until an entire line has been typed. Also, no matter how many bytes are requested in the `read()` call, the input will comprise at most one line. It is not, however, necessary to read a whole line at once; any number of bytes, even one, may be requested in a `read()` without losing information.

`{MAX_CANON}`, the maximum number of bytes in a line (see `limits.h`), must be greater than or equal to `{_POSIX_MAX_CANON}`. If this limit is exceeded, the behavior of the system is undefined. If `{MAX_CANON}` is not defined, there is no such limit (see also `pathconf()`). Both constants have no effect for BS2000 block terminals, since I/O is controlled there by TIAM.

ERASE and KILL processing occur when either of the two special characters, the ERASE and KILL characters (see section "Special characters" on page 99), is read. The processing of this character affects the input buffer that has not been delimited yet by a newline character (LF), an end-of-file character or an end-of-line character. This undelimited data constitutes the current line. The ERASE character deletes the last character entered in the current line, provided such a character follows the start of the line. The KILL character kills (deletes) the entire current line, if there is one, and may optionally output a new newline character. The ERASE and KILL characters have no effect if there is no data in the current line. The ERASE and KILL characters themselves are not placed in the input queue. Both characters take effect immediately after the corresponding key is pressed, independent of any backspace or tab characters that may have been entered. It is also possible to enter them directly as constants by preceding them with the escape character `\`. The escape character itself is not read in this case. The deleted characters can be changed.

## Non-canonical mode input processing

This type of input processing is only supported by character-oriented terminals, not by block terminals.

In non-canonical mode input processing, input bytes are not assembled into lines, and erase and kill processing does not occur. The values of the `MIN` and `TIME` elements of the `c_cc` array are used to determine how the process is to receive the bytes. The `O_NONBLOCK` flag (see also `open()` or `fcntl()`) has precedence over the specifications in the `c_cc` array. Consequently, if `O_NONBLOCK` is set, `read()` will return immediately, regardless of the `MIN` and `TIME` values. Furthermore, if no data is present, `read()` can return either 0 or -1 and set `errno` to `EAGAIN` in the latter case.

`MIN` represents the minimum number of bytes (maximum 255) that should be received (i.e. returned to the user) when the `read()` function successfully returns. `TIME` is a timer of 0.1 second granularity that is used to time-out bursty and short term data transmissions. If `MIN` is greater than `{MAX_INPUT}`, the response to the request is not defined. The four possible combinations for `MIN` and `TIME` and their interactions are described below:

### Case 1: `MIN > 0, TIME > 0`

In this case `TIME` serves as an inter-byte timer and is activated after the first byte is received. Since it is an inter-byte timer, `TIME` is reset for each byte and started as soon as one byte is received. If `MIN` bytes are received before the inter-byte timer expires, the read operation is satisfied. If the timer expires before `MIN` bytes are received, the characters received to that point are returned to the user. Note that if `TIME` expires, at least one byte is returned, since the timer is not enabled unless a byte is received. In this case (`MIN > 0, TIME > 0`) the read blocks until either the `MIN` and `TIME` mechanisms are activated by the receipt of the first byte, or a signal is received.

### Case 2: `MIN > 0, TIME = 0`

Since the value of `TIME` is zero, the timer plays no role, and only `MIN` is significant. In this case, the read operation blocks until `MIN` bytes are received or a signal arrives. A program that uses this case to read records from the terminal may block for any length of time in the read operation (even indefinitely).

### Case 3: `MIN = 0, TIME > 0`

Since `MIN = 0`, `TIME` no longer represents an inter-byte timer in this case but serves as a read timer (for the entire read operation) that is activated as soon as the call to `read()` is processed (default action). In this case, a read operation is satisfied as soon as a single byte is received or the read timer `TIME` expires. If no byte is received within `TIME * 0.1` seconds after `read()` is called, the `read()` function returns a value of zero, having read no data.

### Case 4: MIN = 0, TIME = 0

In this case, either the number of bytes to be read or the number of bytes currently available (if there are not enough bytes) is returned without waiting for more bytes to be input. If no input characters are available, the `read()` function returns a value of zero, having read no data.

## Writing data and output processing

When a process writes one or more bytes to a special file associated with a terminal, these bytes are processed according to the settings in `c_oflag` (see section "Output modes" on page 104). The system may provide a buffering mechanism, with the result that when a call to `write()` completes, all of the bytes written will have been scheduled for transmission to the device, but the transmission will not necessarily have completed. See `write()` for the effects of `O_NONBLOCK` on `write()`.

## Special characters

The special characters described below are assigned at task initialization to program keys by a precursor task. They are associated with certain special functions on input and/or output. Cases where the association between a character and function cannot be changed are indicated by enclosing the relevant character in parentheses:

INTR	Special character on input, which is recognized if the <code>ISIG</code> flag is set. It generates a <code>SIGINT</code> signal (interrupt) which is sent to all processes in the foreground process group associated with the terminal. If the <code>ISIG</code> flag is set, the <code>INTR</code> character is discarded when processed. Under normal circumstances, this results in the termination of all these processes; however, arrangements may be made to ignore the signal or initiate a jump to a previously defined address location (see <code>sigaction()</code> and <code>signal()</code> ).
QUIT	Special character on input, which is recognized if the <code>ISIG</code> flag is set. It generates a <code>SIGQUIT</code> signal (quit) which is sent to all processes in the foreground process group associated with the terminal. If <code>ISIG</code> is set, the <code>QUIT</code> character is discarded when processed. Its treatment is almost identical to the interrupt signal <code>SIGINT</code> , except that, if a receiving process has not made other arrangements, the process will not only be terminated, but a core dump will be generated (see <code>sigaction()</code> ).
ERASE	Special character on input, which is recognized if the <code>ICANON</code> flag is set. It erases the preceding character, but not beyond the start of a line, i.e. an <code>NL</code> , <code>EOF</code> or <code>EOL</code> character (see also the section "Canonical mode input processing" on page 97). If <code>ICANON</code> is set, the <code>ERASE</code> character is discarded when processed.

KILL	<p>Special character on input, which is recognized if the <code>ICANON</code> flag is set. It deletes the entire line as of the last <code>NL</code>, <code>EOF</code> or <code>EOL</code> character. If <code>ICANON</code> is set, the <code>KILL</code> character is discarded when processed.</p> <p>This character is not supported on BS2000 block terminals.</p>
EOF	<p>Special character on input, which is recognized if the <code>ICANON</code> flag is set. When <code>EOF</code> is received, all the bytes waiting to be read are immediately passed to the program without waiting for an <code>NL</code> character, and the <code>EOF</code> is discarded. If no bytes are present, i.e. the <code>EOF</code> is at the beginning of a line, <code>read()</code> returns a value of 0. A return value of 0 for a read operation is the default end-of-file identifier. If <code>ICANON</code> is set, the <code>EOF</code> character is discarded when processed.</p>
NL	<p>Special character on input, which is recognized if the <code>ICANON</code> flag is set. <code>NL</code> is the standard line delimiter <code>\n</code> and cannot be changed.</p>
EOL	<p>Special character on input, which is recognized if the <code>ICANON</code> flag is set. <code>EOL</code> is an additional line delimiter and serves the same function as <code>NL</code>. It is normally not used.</p>
SUSP	<p>If job control is supported (section "Special control characters" on page 109) on an X/Open-compatible system, the <code>SUSP</code> special character is recognized on input. If the <code>ISIG</code> flag is set, receipt of the <code>SUSP</code> character causes a <code>SIGTSTP</code> signal to be sent to all processes in the foreground process group associated with the terminal. After it is processed, the <code>SUSP</code> character is discarded as well. This character has no effect in the POSIX subsystem, since job control is not supported in its current implementation.</p>
STOP	<p>Special character on both input and output, which is recognized if either the <code>IXON</code> (for input) or <code>IXOFF</code> (for output) flag is set. <code>STOP</code> can be used to suspend output temporarily. It is useful with CRT terminals to prevent output from disappearing before it can be read. If <code>IXON</code> is set, the <code>STOP</code> character is discarded when processed. So long as the output is suspended, additional <code>STOP</code> characters are ignored and not read. The <code>STOP</code> character can be neither changed nor escaped.</p> <p>This character is not supported on BS2000 block terminals.</p>
START	<p>Special character on both input and output, which is recognized if either the <code>IXON</code> (for input) or <code>IXOFF</code> (for output) flag is set. The <code>START</code> character is used to resume output that has been suspended by a <code>STOP</code> character. So long as output continues, subsequent <code>START</code> characters are ignored and not read. If <code>IXON</code> is set, the <code>START</code> character is discarded when processed. The <code>START</code> character can be neither changed nor escaped.</p> <p>This character is not supported on BS2000 block terminals.</p>

**CR** Special character on input, which is recognized if the `ICANON` flag is set. This character corresponds to the character `\r`. If `ICANON` and `ICRNL` are set, and `IGNCR` is not, this character is translated into an `NL`, and has the same effect as an `NL` character. The character `CR` cannot be changed.

The values for `INTR`, `QUIT`, `ERASE`, `KILL`, `EOF`, `EOL` and `SUSP` (job control only) can be changed by the user.

If two or more special characters have the same value, the function performed when that character is received is undefined.

The `ERASE`, `KILL` and `EOF` characters can be escaped by a preceding `\` (escape character), in which case the function associated with it is not executed.

The user may overwrite key assignments at any time. In such cases, the default XPG4 Version 2-conformant key assignments on the BS2000 command level can be restored with `/RESTORE-CONTROL-KEYS`.

## Modem disconnect

If the carrier signal is lost, i.e. a modem disconnect is detected by the terminal interface for a controlling terminal, and if `CLOCAL` is not set in `c_cflag` (see section "Control modes" on page 106), the hangup signal `SIGHUP` is sent to the controlling process associated with the terminal. Unless other arrangements have been made, this causes the controlling process to terminate (see `exit()`). All subsequent read operations from this terminal device return with an end-of-file indication. Thus, processes that read a terminal file and test for end-of-file can terminate appropriately after a disconnect. Any subsequent `write()` to the terminal device returns `-1`, with `errno` set to `EIO`, until the file is closed.

## Closing a terminal device file

When the last process closes a terminal device file, any pending output is sent to the device, and any input that is still to be read is discarded. If `HUPCL` is set in the control modes and the communications port supports a disconnect function, the terminal interface performs a disconnect.

## Settable parameters

### The termios structure

Programs that need to control I/O flags for terminals can do this by means of the `termios` structure defined in the header `termios.h`. The members of this structure include:

Member type	Array size	Member name	Definition
<code>tcflag_t</code>		<code>c_iflag</code>	input modes
<code>tcflag_t</code>		<code>c_oflag</code>	output modes
<code>tcflag_t</code>		<code>c_cflag</code>	control modes
<code>tcflag_t</code>		<code>c_lflag</code>	local modes
<code>cc_t</code>	NCCS	<code>c_cc[]</code>	control characters

The types `tcflag_t` and `cc_t` are defined in the header `termios.h` as unsigned integral types.

### Input modes

The `c_iflag` field describes the basic terminal input control:

Mask name	Definition
BRKINT	Send SIGINT signal on break
ICRNL	Map CR to NL on input
IGNBRK	Ignore break
IGNCR	Ignore CR
IGNPAR	Ignore characters with parity errors
INLCR	Map NL to CR on input
INPCK	Enable input parity check
ISTRIP	Strip 8th bit of input character
IXOFF	Enable START/STOP input control
IXON	Enable START/STOP output control
PARMRK	Mark parity errors
IUCLC Will no longer be supported by the X/Open-Standard in future.	Map uppercase to lowercase on input
IXANY	Enable any input character to restart output

In the context of asynchronous serial transmission via a serial interface, a `break` is defined as a sequence of zero-valued bits that continues for more than the time required to send one byte. The entire sequence of zero-valued bits is interpreted as a single `break`, even if the sequence comprises more than one byte. In contexts other than asynchronous serial data transmission, the meaning of a `break` condition is not defined.

If `IGNBRK` is set, any `break` that occurs on input will be ignored, i.e. not placed in the input queue and therefore not read by any process. On the other hand, if `BRKINT` is set, the `break` condition generates a single interrupt signal `SIGINT` and flushes both the input and output queues. If neither `IGNBRK` nor `BRKINT` is set, a `break` condition is read as a single `\0` character, or if `PARMRK` is set, as `\377`, `\0`, `\0`.

If `IGNPAR` is set, any byte with a character or parity error (other than `break`) is ignored.

If `PARMRK` is set, and `IGNPAR` is not set, any byte with a framing or parity error (other than `break`) is passed to the application as the three-character sequence `\377`, `\0` and `X`, where `\377` and `\0` constitute a two-byte flag preceding each sequence and `X` corresponds to the character received in error. To avoid ambiguity in this case, if `ISTRIP` is not set, a valid character of `\377` is passed to the application as `\377`, `\377`. If neither `PARMRK` nor `IGNPAR` is set, a framing or parity error (other than `break`) is passed to the application as a single

If `INPCK` is set, input parity checking is enabled. If `INPCK` is not set, input parity checking is disabled, allowing generation of the output parity bit without heeding any input parity errors that may have occurred.

## Note

The enabling or disabling of input parity checking is independent of whether parity detection is enabled or disabled (see section "Control modes" on page 106). If parity detection is enabled but input parity checking is disabled, the hardware to which the terminal is connected will recognize the parity bit, but the terminal special file will not check whether or not this bit is correctly set.

If `INLCR` is set, a received `NL` character (newline) is translated into a `CR` character (carriage return). If `IGNCR` is set, a received `CR` character is ignored (not read). However, if `IGNCR` is not set and `ICRNL` is set, a received `CR` character is converted into an `NL` character.

If `IUCLC` is set, a received uppercase letter is mapped to the corresponding lowercase letter (Will no longer be supported by the X/Open-Standard in future.).

If `IXON` is set, `START/STOP` output control is enabled. A received `STOP` character suspends output, and a received `START` character restarts output. The control characters for `START` and `STOP` are not read during a read operation, however, they perform flow control functions when `IXON` is set. When `IXON` is not set, the `START` and `STOP` characters are read. If `IXANY` is set, the suspended output is resumed as soon as any character is entered.

If `IXOFF` is set, `START/STOP` input flow control is enabled. The system transmits `STOP` characters in order to cause the terminal device to stop transmitting data as needed to prevent an overflow in the input queue (no more than `{MAX_INPUT}` bytes are permitted). It transmits `START` characters to cause the terminal device to resume transmitting data, as soon as the device can continue doing so without any risk of overflowing the input queue.

The initial value for all input modes after an `open()` is that no flag is set.

## Output modes

The `c_oflag` field specifies how the terminal interface handles output. It is constructed from the bitwise inclusive OR of zero or more of the following masks, which are bitwise distinct. The mask names in the table below are defined in `termios.h`:

Mask name	Definition
<code>OPOST</code>	Post-process output
<code>OLCUC</code>	Map lowercase letters to uppercase on output. Will no longer be supported by the X/Open-Standard in future.
<code>ONLCR</code>	Map <code>NL</code> to <code>CR-NL</code> on output
<code>OCRNL</code>	Map <code>CR</code> to <code>NL</code> on output
<code>ONOCR</code>	No <code>CR</code> output at column 0
<code>ONLRET</code>	<code>NL</code> performs <code>CR</code> function
<code>OFILL</code>	Use fill characters for delay
<code>OFDEL</code>	The fill character is <code>DEL</code> (otherwise <code>NUL</code> )
<code>NLDLY</code>	Select newline ( <code>NL</code> ) delays
<code>NL0</code>	<code>NL</code> character type 0
<code>NL1</code>	<code>NL</code> character type 1
<code>CRDLY</code>	Select carriage return ( <code>CR</code> ) delays:
<code>CR0</code>	<code>CR</code> delay type 0
<code>CR1</code>	<code>CR</code> delay type 1
<code>CR2</code>	<code>CR</code> delay type 2
<code>CR3</code>	<code>CR</code> delay type 3
<code>TABDLY</code>	Select delays for horizontal tabs:
<code>TAB0</code>	Horizontal-tab delay type 0
<code>TAB1</code>	Horizontal-tab delay type 1
<code>TAB2</code>	Horizontal-tab delay type 2
<code>TAB3</code>	Expand tabs to spaces
<code>BSDLY</code>	Select delays for backspace:
<code>BS0</code>	Backspace-delay type 0
<code>BS1</code>	Backspace-delay type 1



Mask name	Definition
VTDLY	Select delays for vertical tabs:
VT0	Vertical-tab delay type 0
VT1	Vertical-tab delay type 1
FFDLY	Select delays for form-feed:
FF0	Form-feed delay type 0
FF1	Form-feed delay type 1

If `OPOST` is set, output data is post-processed on the basis of the remaining bits of `c_oflag` so that lines of text are modified to appear appropriately on the terminal device; otherwise, characters are transmitted without change.

If `OLCUC` is set, a lowercase letter is mapped to the corresponding uppercase letter before being transmitted. This function is often used in conjunction with `IUCLC` for input modes. Will no longer be supported by the X/Open-Standard in future.

When `ONLCR` is set, the `NL` character (newline) is transmitted as the character pair `CR-NL` (carriage return - newline). If `OCRNL` is set, the `CR` character is transmitted as an `NL` character. When `ONOCR` is set, a `CR` character in column 0 (first position in the line) is not transmitted. If `ONLRET` is set, it is assumed that the `NL` character performs the carriage return function; the column pointer is set to 0, and the applicable carriage return delay is used. When `ONLRET` is not set, it is assumed that the `NL` character just performs the line-feed function; the column pointer will remain unchanged in this case. The column pointer is also set to 0 if the `CR` character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay. If `OFILL` is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If `OFDEL` is set, the fill character is `DEL`, otherwise `NUL`.

If a form-feed or vertical-tab delay is specified, the delay will last for about 2 seconds. A newline delay lasts for about 0.10 seconds. If `ONLRET` is specified, carriage-return delays are used instead of newline delays. Two fill characters are transmitted if `OFILL` is set.

Carriage-return delay type 1 depends on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If `OFILL` is set, two fill characters are transmitted for type 1, four for type 2.

Horizontal-tab delay type 1 depends on the current column position. Type 2 lasts for about 0.10 seconds; type 3 specifies that tabs are to be expanded into spaces. If `OFILL` is set, two fill characters are transmitted for any delay.

The backspace delay lasts for about 0.05 seconds. One fill character is transmitted if `OFILL` is set. The actual delays will depend on the line speed and the load on the system.

The initial value for all output modes (value of `c_oflag`) after a call to `open()` is that no flag is set.

## Control modes

The control modes described below have no significance for BS2000 computers. The `c_cflag` field describes the hardware control of the terminal. The following elements are supported for character-oriented terminals:

Mask name	Definition
CLOCAL	Ignore modem status
CREAD	Enable receiver
CSIZE	Character size (number of bits per byte):
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits
CSTOPB	Send 2 stop bits (else 1)
HUPCL	Hang up on last <code>close()</code>
PARENB	Enable parity detection
PARODD	Enable odd parity

In addition, the input and output baud rates are stored in the `termios` structure. The following values are supported:

Name	Definition
B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud

The following interfaces are provided for getting and setting the values of the input and output baud rates in the `termios` structure:

`cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()` and `cfsetospeed()`.

The `CSIZE` bits specify the character size in bits per byte for both transmission and reception. This size does not include the parity bit, if any. If `CSTOPB` is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stop bits are normally used.

If `CREAD` is set, the receiver is enabled. If `CREAD` is not set, no characters are received.

If `PARENB` is set, parity generation and detection is enabled, i.e. a parity bit is added to each character. If parity is enabled, the `PARODD` bit specifies odd parity be used; otherwise, even parity is used.

If `HUPCL` is set, the line will be disconnected when the last process to use the line closes it or terminates. This means that the Data-Terminal-Ready (DTR) signal will be disabled, thus breaking the modem connection.

If `CLOCAL` is set, the existing line is assumed to be a local, direct connection with no modem control. The connection does not depend on line signals in this case. Otherwise, modem control is assumed, and the modem status lines are monitored.

Under normal circumstances, a call to the `open()` function waits for the modem connection to complete. However, if the `O_NONBLOCK` flag is set when calling `open()`, or if the `CLOCAL` bit is set, the `open()` function returns immediately without waiting for the connection.

If the object for which the control modes are set is not an asynchronous serial connection, some of the modes may be ignored; e.g., if an attempt is made to set the baud rate on a network connection to a terminal on another host, the baud rate may or may not be set on the connection between that terminal and the machine to which it is directly connected.

The initial value for the control modes (value of `c_oflag`) after a call to `open()` is that no flag is set.

## Local modes

The `c_lflag` field of the structure is used to control various functions:

Mask name	Definition
ECHO	Enable echo function
ECHOE	Echo ERASE character as BS-SP-BS (error correcting backspace)
ECHOK	Echo NL after KILL character
ECHONL	Echo NL character
ICANON	Enable canonical input (line-oriented input with ERASE and KILL processing)
IEXTEN	Enable extended functions
ISIG	Enable signals

Mask name	Definition
NOFLSH	Disable flushing of input or output queues after <code>INTERRUPT</code> or <code>QUIT</code> from keyboard
TOSTOP	Send <code>SIGTTOU</code> signal with output for background process group
XCASE	Canonical uppercase and lowercase representation. Will no longer be supported by the X/Open-Standard in future.

If `ECHO` is set, input characters are echoed back to the terminal as soon as they are received. If `ECHO` is clear, input characters are not echoed.

If `ICANON` and `ECHOE` are set, the `ERASE` character is echoed as a backspace-space-backspace sequence, which causes the terminal to clear the last character, if any, from the display. If `ECHOE` is set and `ECHO` is not set, the `ERASE` character is echoed as `SP BS`.

If `ECHOK` and `ICANON` are set, the `KILL` character causes the terminal to erase the line from the display or echoes the `NL` character after the `KILL` character to indicate that the line will be deleted.

If `ECHONL` and `ICANON` are set, the `NL` character is echoed even if `ECHO` is not set. This is useful for terminals set to the local echo (so-called half duplex) mode. The `EOF` character is echoed only if it is escaped. Since `EOT` (End Of Transmission) is used as the default `EOF` character, this prevents a connection cleardown from terminals that hang up when `EOT` is received.

If `ISIG` is set, each input character is checked to determine whether it is one of the special control characters `INTR`, `QUIT` or `SUSP` (job control only). If this is the case, the function associated with that character is performed. If `ISIG` is not set, no checking is done. In other words, these special input functions can only be performed if `ISIG` is set. They can, however, be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g. `0377`).

If `ICANON` is set, canonical processing is enabled. This enables the functions to process `ERASE` and `KILL` characters. The input characters are assembled into lines delimited by `NL`, `EOF` and `EOL`, as described in section "Canonical mode input processing" on page 97.

If `ICANON` is not set, read requests are satisfied directly from the input queue. This does not take place until at least `MIN` bytes have been received or the timeout value `TIME` has expired (see section "Non-canonical mode input processing" on page 98 for more details). The `TIME` value is specified in tenths of a second. If `NOFLSH` is set, the normal flush of input and output queues that follows reception of `INTR`, `QUIT` and `SUSP` (job control only) characters is not performed.

The initial value for the local modes (value of `c_local`) after a call to `open()` is that no flag is set.

## Special control characters

The values of special control characters are defined by the array `c_cc`. The subscript name and description for each array element in both canonical and non-canonical modes are listed in the table below:

Index name in the ...		Definition
Canonical mode	Non-canonical mode	
VEOF		EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character

The subscript names are constants that represent the subscript of each respective element (character) in the `c_cc` array. The character `c_cc[VSTOP]`, for example, is thus the STOP character in canonical as well as non-canonical mode.

The subscript names are unique, except that the `VMIN` and `VTIME` subscripts may have the same values as `VEOF` and `VEOL`, respectively.

Implementations (such as the POSIX subsystem) that do not support job control may ignore the SUSP character value indexed by the `VSUSP` subscript in the `c_cc` array.

If `{_POSIX_VDISABLE}` is defined for the terminal device file (i.e. the special file associated with the terminal) and the value of one of the modifiable special characters is the same as `{_POSIX_VDISABLE}` (see section "Special characters" on page 99), then that function is deactivated, i.e. no input character will be recognized as the disabled special character. If `ICANON` is not set, the value of `{_POSIX_VDISABLE}` has no special meaning for entries with the `VMIN` and `VTIME` subscripts in the `c_cc` array.

## Block terminal support

The terminal is mapped to the special file `/dev/tty`, so all terminal I/O essentially involves I/O on the special file `/dev/tty`. The size of the input and output buffers for `/dev/tty` is 12,264 bytes, respectively. Only the control characters `\n` (newline) and `\t` (8-character tabulator) are converted.

The input of `[EM] [DÜ1]` is interpreted as `\n` (newline). The tab key does not generate the tab character `\t`. Input from the terminal is buffered. If the buffer contains residual data, the maximum number of characters returned by a call to `read()` will be restricted to the number of bytes contained in the buffer. It is only when the buffer is empty that the user is prompted for input from the terminal.

Input cannot be aborted with the `[K2]` key. It is only after all input has been read in from the terminal that the user can switch to system mode. In other words, users must press `[EM] [DÜ1]` once before they enter system mode.

On output, `\n` generates a line feed, and `\t` produces a tab. All other control characters are not converted; they are simply mapped as scribble characters. Output occurs at any of the following events:

- a newline (`\n`) is encountered
- the buffer is full
- on input from the terminal (terminated by `[EM] [DÜ1]`)
- the program terminates

## Support for BS2000 consoles

The special file `/dev/console` can only be opened for writing. This must be done by opening the file with `open("/dev/console")`. The size of the output buffer for `/dev/console` is 230 bytes.

## Process control

In the POSIX subsystem, a program is run in a process; in BS2000, by contrast, it is run in a task. In other words, if a program is called in the POSIX shell, a child process is spawned, but no process will be generated if it is called from the BS2000 command interface.

## Signals

When a program is called in the POSIX subsystem, signal handling is performed by the C runtime system via the XPG4 Version 2-conformant facilities of the POSIX subsystem.

For programs called in BS2000, by contrast, signal handling is implemented by using the mechanisms in BS2000 (STXIT).

The `cstxit()` function can be used to bypass POSIX signal handling and to register STXIT routines at the system; however, users are cautioned against using this option. POSIX and STXIT signals cannot be otherwise handled in the same program.

Signal handling is based on the functions `signal()`, `sigaction()`, `sigprocmask()` and `kill()`. There are three possible settings for each signal (see `sigaction()`).

When a process aborts, the number of the signal that triggered the abort, the address at which the program was aborted, and a prompt to request a core dump if desired are output.

All signals supported in the POSIX subsystem are defined in the header file `signal.h` and are described in the corresponding section of this manual. These signals are generated when the specific event associated with them occurs. There are some restrictions that apply to the user:

- A registered STXIT routine will always be called before any registered signal handling by the system. If no signal was registered, then no registered STXIT routine will be called.
- To ensure that the implicit TU contingency of the signal handling is not interrupted, no contingency routines above level 125 should be registered in any case.
- A dialog key is defined for the following signals:

Signals	Dialog key
SIGINT	<code>[INTR]</code>
SIGQUIT	<code>[QUIT]</code>
SIGSTOP	<code>[STOP]</code>
SIGTSTP	<code>[SUSP]</code>
SIGCONT	<code>[START]</code>

The `[STOP]` and `[START]` keys are not supported on block terminals (see also section "Block terminal support" on page 110).

## Interprocess communication

The use of inter-process communication (IPC) functions affects some other services. The affected functions are shown in the table below:

Interfaces affected by IPC		
errno	execve()	execl()
execvp()	execl()	exit()
execlp()	fork()	execv()

## General description

The IPC package provides three facilities for inter-process communication:

- Messages, which are formatted streams that can be sent by processes to any other processes (the following system calls are used: `msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`).
- Shared memory, which allows processes to share parts of their virtual address space with other processes (the following system calls are used: `shmget()`, `shmat()`, `shmdt()`, `shmctl()`).
- Semaphores, which make it possible to synchronize process execution (the following system calls are used: `semget()`, `semop()`, `semctl()`).

The aspects common to the operation of all three facilities are described below under the following sections: The description is divided up among the following sections:

- Creation of a communication element (message queue, shared memory, semaphore)
- Data structures
- Requesting/modifying status information

Note that *xxx* stands for `msg`, `sem`, or `shm`, respectively.

Each communication element (message queue, shared memory, semaphore) is identified by means of a positive integer, which is assigned by the system on creation of the communication element `xxxget()`. The user may also specify a numerical key to name a communication element that he or she creates.

Associated with each facility is a table, with entries containing all communication elements for the respective facility.

Each entry is named by means of a user-selected numerical key which serves as its ID.



## Creation of a communication element

Each facility has a corresponding system call `xxxget()` with which a new element can be created or an existing one made available to a process. The parameters of the `xxxget()` system calls are a user-selected numerical key, *key*, designating the login name, and a flag *xxxflg*.

*key*           The operating system searches the appropriate table for an entry identified by the key. Processes may call the `xxxget()` system call with the `IPC_PRIVATE` key, thus ensuring that an unused entry is returned.

*xxxflg*       This flag determines whether and how an entry can be accessed, and may influence the access permissions. If the `IPC_CREAT` flag is set, a new entry is created (if none exists). The required access permissions are OR-ed with `IPC_CREAT`. The nine right-hand bits in the flag are then set as the access permissions for the new entry. The bit ordering corresponds to that of *oflag* in the `open()` system call, even if only read and write permissions are of significance.

If an entry already exists with the specified key, the nine right-hand bits of the flag must be a subset of the entry's access permissions; otherwise, the `xxxget()` system calls will fail. Thus, any permissions that extend beyond those available may not be requested. In order to modify access permissions, the `xxxctl()` system call must be executed (see below). When the `IPC_CREAT` flag is additionally OR-ed with the `IPC_EXCL` flag, `xxxget()` returns an error if an entry already exists for the key.

If the `IPC_CREAT` flag is not set, an entry must already exist; otherwise, the `xxxget()` system calls will fail.

The `xxxget()` system calls return a unique positive identifier (system identifier *xxxid*) which is selected by the operating system and is used in the other system calls associated with the facility. These identifiers operate in a similar way to the file descriptors, as returned by `open()`, `dup()` and `pipe()`, except that they may be used by all processes that know their value, i.e. inheritance is not required for them to be valid. Each shared memory segment, message queue, and semaphore set is identified by a shared memory identifier (*shmid*), a semaphore identifier (*semid*) and the message queue identifier (*msqid*), respectively.

## Data structures

Associated with each identifier is a data structure which contains data related to the operations which may be or have been performed. These data structures (`msqid_ds`, `semid_ds`, `shmid_ds`) are defined in `sys/shm.h`, `sys/sem.h` and `sys/msg.h`. They include the process ID of the last process executed by an operation (send or receive message, access shared memory, etc.) and the time of the last access operation.

Each of the data structures contains both ownership information and an `ipc_perm` structure (see `sys/ipc.h`), which are used in conjunction to determine whether or not read/write (read/alter for semaphores) permissions should be granted (or denied) to processes using the IPC facilities. The `ipc_perm` structure contains the effective user ID and group ID of the process that created the entry (`xxx_perm.cuid` and `xxx_perm.cgid`), in addition to a user ID and group ID (`xxx_perm.uid` and `xxx_perm.gid`) which may also be set by means of the `xxxctl()` system call. There is also a bit field for permissions in the `mode` member of the `ipc_perm` structure. The values of the bits are given below:

Bit	Meaning
0400	Read (owner)
0200	Write (owner)
0040	Read (group)
0020	Write (group)
0004	Read (others)
0002	Write (others)

The name of the `ipc_perm` structure is `shm_perm`, `sem_perm`, or `msg_perm`, depending on which facility is being used. In each case, read and write/alter permissions are granted to a process if one or more of the following conditions are true:

- The effective user ID is that of a process with appropriate privileges.
- The effective user ID of the process matches `xxx_perm.cuid` or `xxx_perm.uid` in the data structure associated with the IPC identifier, and the appropriate bit for the owner is set in `xxx_perm.mode`.
- The effective user ID of the process does not match `xxx_perm.cuid` or `xxx_perm.uid` but the effective group ID of the process matches `xxx_perm.cgid` or `xxx_perm.gid` in the data structure associated with the IPC identifier, and the appropriate bit for group is set in `xxx_perm.mode`.
- The effective user ID of the process does not match `xxx_perm.cuid` or `xxx_perm.uid`, and the effective group ID of the process does not match `xxx_perm.cgid` or `xxx_perm.gid` in the data structure associated with the IPC identifier, but the appropriate bit for others is set in `xxx_perm.mode`.

In all other cases, the permission is denied.

## Requesting/modifying status information

`xxxctl()`

Each facility has a corresponding system call `xxxctl()`, which allows the status of an entry to be requested, status information to be set, or an entry to be removed from the system.

- If a process requests the status of an entry, the operating system checks whether the process has read permission and then copies data from the table entry to the user-specified structure.
- If a process wishes to reset the parameters of the entry, the operating system checks whether the effective user ID of the process matches the user ID of the entry or the user ID of the entry creator, or whether the effective user ID is that of a process with appropriate privileges. Write permission alone is not sufficient to reset parameters. The operating system copies the user-specified data to the table entry, setting the user ID, group ID, access permissions, and other fields that depend on the type of facility. Since the fields with the user ID and group ID of the entry creator are not modified, the latter always retains control permissions.
- If a process wishes to remove an entry, the operating system checks whether the effective user ID of the process matches one of the user IDs in the `ipc_perm` structure. It is not possible to access a removed entry with the old identifier.

### Note

The IPC facilities must be used with great care, since unused or unneeded IPC members are not always recognized by the operating system. There are no records in the operating system as to which processes access an IPC member. Any process that knows the correct identifier and has access permission can essentially access an IPC member, even if the process has never executed a `xxxget()` system call. It is therefore not possible for the operating system to implicitly flush IPC structures (e.g. on completion of a process).

IPC facilities should only be used in the case of extreme performance requirements.

## Shared memory

Shared memory is provided for the C library functions (see the manual "BS2000/OSD-BC Executive Macros"). Shared memory segments are created by the `shmget()` function. On non-XS systems, shared memory is created in units of 64 Kbytes that are aligned on 64-Kbyte boundaries; on XS systems, it is allocated in units of 1 Mbyte in the upper address space and is aligned on a 1-Mbyte boundary. The *size* argument of the `shmget()` function is rounded accordingly.

`shmget()` returns an identifier for the shared memory called the shared memory ID; `shmat()` attaches the shared memory segment.

Shared memory segments are detached upon successful completion of a call to `shmdt` or at program termination. The associated shared memory ID is removed:

- by `shmctl()`
- when the last process using the shared memory has detached itself with `shmdt()`
- at program completion.

It is only then that the same shared memory ID may be reused.

A maximum of 150 IDs are available in BS2000 for shared memory. Up to 32 calls to the `shmat()` function are allowed per program.

**Note that in order to enable shared memory locking with the `SHM_LOCK` control option of the `shmctl()` function, the `RESIDENT-PAGES` operand must be specified at `/START-PROGRAM`.**

## Contingency and STXIT routines

This section explains how contingency and STXIT routines can be implemented in C.

Familiarity with the concept of contingency and STXIT routines is essential to understanding the material presented here. These concepts and the related BS2000 system macros are described in detail in the manual "BS2000/OSD-BC Executive Macros".

The library functions mentioned in this section (`signal()`, `raise()`, `alarm()`, `cenaco()`, `cdisco()`, `cstxit()`, `longjmp()`, `setjmp()`) are explained at length in the reference section of this manual.

### Important

The use of some C library functions from within STXIT routines may result in undefined behavior. Consistency in the library functions cannot always be guaranteed in the event of asynchronous interrupts. Undefined behavior results if the same library function or a library function belonging to the same group (see list below) as the one asynchronously interrupted by the STXIT event is to be executed within the STXIT routine.

The "critical" C library functions in connection with asynchronous interrupts are as follows:

- file access functions for opening and closing files:  
`fopen()`, `freopen()`, `open()`, `creat()`, `fclose()`, `close()`
- all file access, file management and input/output functions applied on the same file
- functions that generate random numbers: `rand()`, `srand()`
- time functions: `localtime()`, `gmtime()`
- functions for enabling and disabling contingency routines: `cenaco()`, `cdisco()`
- `atexit()`
- `strtok()`
- `setlocale()`
- input/output functions from the C++ standard library

## The C library functions `alarm()`, `raise()`, and `signal()`

The mechanism of contingency routines or STXIT contingency routines is primarily implemented by the following C library functions:

<code>alarm()</code>	sends the <code>SIGALRM</code> signal (STXIT event <code>RTIMER</code> )
<code>raise()</code>	sends signals (simulated STXIT events and user-defined events)
<code>signal()</code>	assigns signal handling routines

## STXIT contingency routines

The following STXIT event classes can be handled by using the `alarm()`, `raise()` and `signal()` functions:

- `PROCHK` (program check)
- `TIMER` (CPU time interval timer)
- `RUNOUT` (end of program runtime)
- `ERROR` (unrecoverable program errors)
- `INTR` (communication to the program)
- in the dialog only: `BREAK/ESCAPE` (`ESCPBRK`)
- `ABEND`
- `TERM` (normal termination of program)
- `RTIMER` (real time interval timer)

The `SVC` interrupt event class is not supported at present.

## Event-driven routines

`signal()` and `raise()` can be used to implement two event-driven routines via two user-defined signals (`SIGUSR1`, `SIGUSR2`).

Eventing using C library functions with only work within a single task, i.e. no intercommunication between different tasks is possible.

The event-driven routines are therefore not implemented internally as contingency routines but via a `CALL` interface.

## Free use of contingency routines

For special requirements that are not covered by the `signal()` and `raise()` functions, appropriate BS2000 functions for eventing can be freely programmed. Such requirements include, for example, a greater number of events (only two events can be defined with `raise()` and `signal()`) or inter-task communication (`raise()` and `signal()` permit eventing only within a single task).

Functions for actual eventing, such as starting event-driven processing and sending and receiving signals, must be implemented in Assembler program sections with the appropriate BS2000 macro calls (`POSSIG`, `SOLSIG`, `ENAEI`).

The macros for enabling, disabling and terminating contingency processes (`ENACO`, `DISCO`, `RETCO`) must not be used in the Assembler program section. Instead of these macros, the C library functions `cenaco()` and `cdisco()` must be called. `cenaco()` and `cdisco()` not only enable and disable contingency routines, but also perform specific actions that are required to ensure that the consistency of the C runtime stack is maintained.

The contingency routine itself can be written in C or in Assembler. Termination of this routine must be effected by means of a "normal" return (with `return()` or `longjmp()` in C, and with `@EXIT` in Assembler).

## Contingency routine in C

When the routine is started, a structure parameter is passed to it. This parameter is declared in the header file `cont.h` as follows:

```
struct contp
{
int    comess;           /* contingency message */
evcode indicat;        /* information indicator */
char   filler[2];      /* reserved for int. use */
evcode switchc;        /* event switch */
int    pcode;          /* post code */
int    reg4;           /* register 4 */
int    reg5;           /* register 5 */
int    reg6;           /* register 6 */
int    reg7;           /* register 7 */
int    reg8;           /* register 8 */
};

#define evcode      char
#define _normal    0      /* evceventnormal */
#define _abnormal  4      /* evceventabnormal */
#define _nmnpc     0      /* evcnocomessnopostcode */
#define _mnp      4      /* evccomessnopostcode */
#define _nmp      8      /* evcnocomesspostcode */
#define _mp       12     /* evccomesspostcode */
```

```
#define _etnm      0      /* evcelapsedtimenocomess */
#define _etm       4      /* evcelapsedtimecomess */
#define _dism      16     /* evceventdisablednocomess */
#define _dis       20     /* evceventdisabledcomess */
```

If the structure parameter described above is to be evaluated, the C routine must provide a formal parameter for a structure of type `contp` and could be constructed as shown below:

```
#include <cont.h>

void controut (struct contp contpar)
{
...
    return ...;
}
```

The C routine can be terminated in one of the following two ways:

- with the `return` statement, which causes the program to be continued at the point of interruption or
- by calling the `longjmp()` function, in which case the program is resumed at the position defined by a `setjmp()` call.

### Contingency routine in Assembler

The contingency routine must be written in Assembler if, for example, further BS2000 macro calls are to be made in it (such as SOLSIG for renewal of the contingency routine).

A structured ILCS Assembler program for a contingency routine is structured something like this:

```
PARLIST DSECT
COMESS DS F
IND DS C
FILLER DS CL2
EC DS C
...
CONTROUT @ENTR TYP=E,ILCS=YES
USING PARLIST,R1
...
SOLSIG
...
@EXIT
```

The `RETCO` macro must not be invoked in the contingency routine. The return must be effected with the `@EXIT` macro.



## Free use of STXIT contingency routines

STXIT contingency routines can be freely programmed in C for special requirements that are not covered by the `signal()` function. Such requirements typically include the transfer of large amounts of data or additional continuation and control options after the execution of the STXIT contingency routine.

The definition of a freely programmed STXIT contingency routine must be effected by calling the C library function `cstxit()`.

The SVC interrupt event class cannot be implemented even if the `cstxit` function is used.

When the STXIT contingency routine is started, it is supplied with a structure that is declared in the header file `stxit.h` as follows:

```
struct stxcontp
{
int      *intwghtp;    /* pointer to interrupt weight */
jmp_buf  *termlabp;   /* pointer to termination label */
int      *regsp;      /* pointer to register save area */
};
```

### Structure of the STXIT contingency routine

In order to use the structure parameter described above, the routine must provide a formal parameter for a structure of type `stxcontp` and could be set up something like this:

```
#include <stxit.h>

void stxrou(stxcontpar)
struct stxcontp stxcontpar;
{
/* ... */
}
```

This routine can be terminated in three different ways:

- with the `return` statement, which causes the program to be continued at the point of interruption or
- by calling the `longjmp()` function that is supplied by the `setjmp` call with a variable of type `jmp_buf`, in which case the program is resumed at the position defined by a `setjmp()` call or
- by calling the `longjmp()` function with the termination label passed in the `stxcontp` structure.

In the case of event class `TERM`, it is not possible to return from the STXIT contingency routine with a `longjmp()` call, since the entries for C functions, including the `main` function, will have already been cleared from the runtime stack at the time this event (`TERM-SVC`) occurs.

## Thread-safe C runtime library by supporting POSIX threads

Programs that work with the POSIX threads described in the XPG5 standard assume that the functions of the runtime system are thread-safe.

To guarantee the thread-safety of the C runtime library, access to global resources (files, global data from the C globals) must be forbidden or protected by a LOCK so that at most one thread can access these resources at a time. The call interface of the functions does not change when this is done. However, a calling thread-1 can be blocked by a thread-2 that has already allocated the resources requested. Only after thread-2 has released the resources can thread-1 access them.

For this reason a thread-safe variant is also supplied in CRTE V2.3A and higher.

Thread safety in the runtime library is realized using the following mechanisms:

- exclusive access to objects of type (FILE \*)

All functions that access objects of type (FILE \*) behave as if they internally use the `flockfile()` and `funlockfile()` functions to obtain exclusive access to these (FILE \*) objects.

- exclusive access to global data anchored in the globals

Functions that access global data are protected by a LOCK.

- `errno` is thread-specific

`errno` does not belong to the global data set any more, but is now thread-specific. This means that for every thread of a process, the value of `errno` is not affected by function calls or the assignment of a value to `errno` by a different thread.

- POSIX thread functions

POSIX thread functions implement exclusive access to objects of type (FILE \*).

The following categories of POSIX thread functions exist:

- POSIX thread functions that are reentrant (containing an "\_r" in the name of the function)
- POSIX thread functions that are automatically protected by a LOCK
- POSIX THREAD functions for locking and unlocking objects of type (FILE\*)
- POSIX thread functions for explicitly locking clients
- POSIX thread functions that affect a process or thread

The individual POSIX functions are described in detail in the chapter "Functions and variables in alphabetical order (a - m)" (see page 161).

- Extended header files

The following header files contain additional function prototypes, data types and constants to support POSIX threads:

- <dirent.h>
- <grp.h>
- <pthread.h>
- <pwd.h>
- <sched.h>
- <signal.h>
- <stdio.h>
- <stdlib.h>
- <string.h>
- <time.h>
- <unistd.h>

The individual header files are described in detail in the chapter "Header files" (see page 929).

Some functions are not thread-safe yet and may not be used in programs that apply multi-threading. This fact is pointed out in the description of this function in chapter "Functions and variables in alphabetical order (a - m)" (see page 161).

In addition, the functions of the `_POSIX_THREAD_SAFE_FUNCTIONS` group were added. A list of these functions can be found on page 140. These functions are described in detail in the chapter "Functions and variables in alphabetical order (a - m)" (see page 161).

## Programming notes

### Return values and result parameters

#### Return value pointer

```
<type> *funct(...)
```

Many functions that return a pointer write their result to an internal C data area that is overwritten whenever the function is called. Since this is a common source of errors, it is mentioned explicitly for all functions of the data type pointer.

#### Return value void \*

```
void * funct(...)
```

If the value of a `void *` function is assigned to a pointer variable, the type should be converted explicitly using the `cast` operator. If the call is made from within a C++ source, explicit type conversion is mandatory.

#### *Example*

```
long *long_ptr;  
.  
.  
long_ptr (long *)calloc(20, sizeof(long));
```

#### Return value int

```
int funct();
```

Character-processing functions have a return value of type `int`, since EOF (`=-1`) is a possible return value for such functions. If the function returns a value of type `char`, an error occurs in a program.

#### Result parameter pointer

```
<type1> funct(<typ2> *variable)
```

Result parameters are variables whose contents are changed by the function, i.e. the function stores a result in such variables. Result parameters are defined without the `const` suffix.

The address, i.e. a pointer, must always be passed as the argument. Furthermore, the memory for the result must be allocated explicitly before calling the function. Since this is often overlooked, reminders are provided in the pertinent function descriptions.

*Examples*

```
struct timeb tp;    /* structure */
ftime(&tp);

char erg;           /* char variable */
scanf("%c", &erg);

char array[10];    /* string variable */
scanf("%s", array);
```

## Error handling

In order to program effectively, it is worth checking most function calls to verify that the function executed successfully. This can be done as follows:

```
if(fct(...) == error result){      /* Check error return value */
    perror("fct:");                /* Output error information */
    exit(error code);              /* Respond to the error, e.g.*/
}                                  /* by terminating the program */
else...
```

Most functions return a value of -1 or the null pointer to indicate that an error occurred when executing the function. See the chapter "Functions and variables in alphabetical order (a - m)" on page 161ff for specific details. To the extent that a function is designed for it, the external variable `errno` may also be set in such cases. The value of the `errno` variable will be defined only after a call to a function where it is explicitly stated that the function sets this variable and will be preserved until it is modified by a subsequent function call. The `errno` variable should be checked only if this is warranted by the value of the function result or is explicitly recommended for a particular function in the "Notes" section. None of the library functions in this manual set `errno` to 0 to indicate an error.

`errno` is not reset by function calls that execute successfully. In some cases, checking `errno` is the only method of determining whether the function executed successfully.

Information specifying the error in more detail is prepared internally on the basis on the error code set in `errno`. The corresponding error message, which contains a brief error text explaining the error, can be written to the standard output by using the `perror()` function.

If more than one error occurs when processing a function call, any of the errors prescribed for the function may be returned, since the order in which the errors are detected is undefined.

All error codes to which `errno` can be set and the corresponding error information are defined in the header file `errno.h`. A detailed listing of these error codes can be found under the description of `errno.h`.

If various types of errors and thus different error codes are possible for a function, it may be useful to query the `errno` variable for the error code so as to vary the response (if appropriate) to the errors that occur. Each error code is represented by a symbolic constant defined in `errno.h`. For example, `ERANGE` indicates an overflow error. A typical query could be written as shown in the following example for the `signal()` function:

```
#include <errno.h>
...
errno = 0;
...
if(signal(sig, fct) == 1){    /* Check the error result */
    if (errno == EFAULT)    /* Responses to EFAULT */
        ...
    else if(errno == EINVAL) /* Responses to EINVAL */
        ...
}
else...
```

The conditions under which an error may occur are presented under the "Errors" heading of the individual function descriptions in the chapter "Functions and variables in alphabetical order (a - m)" on page 161ff.

## Debugging options

If a program is compiled with the option `TEST-SUPPORT=YES`, all the facilities provided by AID can be used for debugging the program (see the manual "AID - Debugging of C/C++ Programs"). Exception: No support for AID when accessing POSIX via rlogin or telnet (AID only works in the block terminal mode).





---

# Functions and variables arranged by theme

This section arranges the functions into groups from a thematic point of view.

## File processing

### File management

"basename - return last element of pathname" on page 184

"chdir - change working directory" on page 207

"chmod - change mode of file" on page 208

"chown - change owner and group of file" on page 210

"chroot - change root directory" on page 212

"clearerr - clear end-of-file and error indicators" on page 213

"closedir - close directory" on page 216

"creat - create new file or overwrite existing one" on page 223

"div - divide with integers" on page 241

"fchdir - change current directory" on page 270

"fchmod - change mode of file" on page 271

"fchown - change owner or group of file" on page 273

"fcntl - control open file" on page 276

"FD\_CLR, FD\_ISSET, FD\_SET, FD\_ZERO - macros for synchronous I/O multiplexing" on page 282

"fstat - get file status of open file" on page 363

"fstatvfs, statvfs - read file system information" on page 366

"ftw - traverse (walk) file tree" on page 377

"fwide - specify file orientation" on page 379

"getcwd - get pathname of current working directory" on page 403

"getdtablesize - get size of descriptor table" on page 412

"getwd - get pathname of current working directory" on page 455

"lchown - change owner/group of file" on page 514

"lstat - query file status" on page 548

"link - create link to file" on page 519

"mkdir - make directory" on page 569

"mknod - make directory, special file, or text file" on page 573

"mkstemp - make unique temporary file name" on page 576

"mktemp - make unique temporary file name (extension)" on page 577

"nftw - traverse file tree" on page 603

"opendir - open directory" on page 615

"readlink - read contents of symbolic link" on page 653

"remove - remove files" on page 672

"rename - rename file" on page 674

"rewinddir - reset file position indicator to start of directory stream" on page 678

"rmdir - remove directory" on page 681

"seekdir - set position of directory stream" on page 686

"stat - get file status" on page 756

"statvfs - read file system information" on page 760

"symlink - make symbolic link to file" on page 802

"sync - update superblock" on page 804

"telldir - get current location of named directory stream" on page 826

"tempnam - create pathname for temporary file" on page 827

"tmpfile - create temporary file" on page 833

"tmpnam - create base name for temporary file" on page 834

"umask - get and set file mode creation mask" on page 850

"unlink - remove link" on page 856

"utime - set file access and modification times" on page 860

**File access**

"access - check access permissions for file" on page 167

"bs2fstat - get BS2000 file names from catalog (BS2000)" on page 189

"close - close file" on page 215

"dup, dup2 - duplicate file descriptor" on page 244

"fattach - assign file descriptor under STREAMS to object in name space of file system" on page 268

"fclose - close stream" on page 274

"fdelrec - delete record in ISAM file (BS2000)" on page 283

"fdetach - cancel assignment to STREAMS file" on page 284

"fdopen - associate stream with file descriptor" on page 286

"feof - test end-of-file indicator on stream" on page 288

"ferror - test error indicator on stream" on page 289

"fflush - flush stream" on page 290

"fgetpos - get current value of file position indicator in stream" on page 295

"fileno - get file descriptor" on page 300

"flocate - set file position indicator in ISAM file (BS2000)" on page 301

"fopen - open stream" on page 312

"freopen - flush and reopen stream" on page 342

"fseek - reposition file position indicator in stream" on page 356

"fsetpos - set file position indicator for stream to current value" on page 361

"fsync - synchronize changes to file" on page 369

"ftell - get current value of file position indicator for stream" on page 370

"ftruncate, truncate - set file to specified length" on page 375

"ioctl - control devices and STREAMS" on page 468

"lockf - lock file section" on page 532

"lseek - move read/write file offset" on page 544

"isastream - test file descriptor" on page 486

"open - open file" on page 609

"rewind - reset file position indicator to start of stream" on page 677

"truncate - set file to specified length" on page 840

"select - synchronous I/O multiplexing" on page 687

"tell - get current value of file position indicator (BS2000)" on page 825

"utimes - set file access time and file modification time" on page 862

### **64-bit functions to support NFS V3.0**

"creat64 - create new file or overwrite existing file" on page 223

"fcntl64 - control open file" on page 276

"fgetpos64 - get current value of the read/write pointer in the stream" on page 295

"fopen64 - open stream" on page 312

"freopen64 - flush stream and open new stream" on page 342

"fseek64 - point read/write pointer in stream to current value" on page 356

"fsetpos64 - set position of read/write pointer in stream to current value" on page 361

"fstat64 - query status of an open file" to page 363

"fstatvfs64, statvfs64 - read file system information" on page 366

"ftell64 - get current value of read/write pointer in stream" on page 370

"ftruncate64, truncate64 - set file length to specified value" on page 375

"getdents64 - convert directory entries" on page 410

"getrlimit64, setrlimit64 - get or set limit value for a resource" on page 441

"lockf64 - lock file section" on page 532

"lseek64 - point read/write pointer to current value" on page 544

"lstat64 - query file status" on page 548

"mmap64 - map memory pages" on page 581

"open64 - open file" on page 609

"readdir64 - read from directory" on page 650

"setrlimit64 - set limit value for a resource" on page 711

"stat64 - query file status" on page 756

"statvfs64 - read file system information" on page 760

## I/O on terminal

- "fgetc - get byte from stream" on page 293
- "fgets - get string from stream" on page 296
- "fgetwc - get wide character string from stream" on page 297
- "fgetws - get wide character string from stream" on page 299
- "fprintf, printf, sprintf - write formatted output on output stream" on page 321
- "fputc - put byte on stream" on page 334
- "fputs - put string on stream" on page 336
- "fputwc - put wide-character code on stream" on page 337
- "fputws - put wide character string on stream" on page 339
- "fread - read binary data" on page 340
- "fscanf, scanf, sscanf - read formatted input" on page 345
- "fwprintf, swprintf, vfwprintf, vswprintf, vwprintf, wprintf - output formatted wide characters" on page 380
- "fwrite - output binary data" on page 392
- "fwscanf, swscanf, wscanf - formatted read" on page 386
- "getc - get byte from stream" on page 396
- "getc\_unlocked, getchar\_unlocked, putc\_unlocked, putchar\_unlocked - standard I/O with explicit lock by the client" on page 398
- "getmsg - get message from STREAMS file" on page 425
- "getopt, optarg, optind, opterr, optopt - command option parsing" on page 428
- "getpass - read string of characters without echo" on page 431
- "getpmsg - get message from STREAMS file" on page 434
- "gets - get string from standard input stream" on page 446
- "getw - read word from stream" on page 452
- "getwc - get wide character from stream" on page 453
- "getwchar - get wide character from standard input stream" on page 454
- „optarg, opterr, optind, optopt - variables for command options" (see getopt)
- "poll - multiplex STREAMS I/O" on page 624

„printf - write formatted output on standard output stream" (see fprintf)  
"putc, putc\_unlocked - put byte on stream" on page 630  
"putchar - put byte on standard output stream (thread-safe)" on page 631  
"putmsg, putpmsg - send message to STREAMS file" on page 633  
"puts - put string on standard output" on page 637  
"putw - put word on stream" on page 639  
"putwc - put wide character on stream" on page 640  
"putwchar - put wide character on standard output stream" on page 641  
"read - read bytes from file" on page 647  
"readv - read array from file" on page 654  
"readdir - read directory" on page 650  
"readdir\_r - read directory (thread-safe)" on page 652  
"readlink - read contents of symbolic link" on page 653  
„scanf - read formatted input from standard input stream" (see fscanf)  
"setbuf - assign buffering to stream" on page 698  
"setvbuf - assign buffering to stream" on page 714  
"sprintf - write formatted output to string" on page 754  
„sscanf - read formatted input from string" (see fscanf)  
"stderr, stdin, stdout - variables for standard I/O streams" on page 761  
"swprintf - output formatted wide characters" on page 801  
"swscanf - formatted read" on page 801  
"ungetc - push byte back onto input stream" on page 853  
"ungetwc - push wide character back onto input stream" on page 855  
"va\_arg - process variable argument list" on page 864  
"va\_end - end variable argument list" on page 865  
"va\_start - initialize variable argument list" on page 866  
"vfprintf, vprintf, vsprintf - formatted output of variable argument list" on page 869  
"vfwprintf - formatted output of wide characters" on page 870  
"vprintf - formatted output to standard out" on page 870

- "vsprintf - formatted output to a string" on page 871
- "vswprintf - formatted output of wide characters" on page 872
- "vwprintf - formatted output of wide characters" on page 872
- "wprintf - formatted output of wide characters" on page 919
- "write - write bytes to file" on page 920
- "writev - write to file" on page 926
- "wscanf - formatted read" on page 927

## Processes

### Process administration

- "cdisco - disconnect contingency routine (BS2000)" on page 200
- "cenaco - define contingency routine (BS2000)" on page 202
- "cstxit - define STXIT routine (BS2000)" on page 227
- "cuserid - get login name" on page 234
- "endgrent, getgrent, setgrent - group management" on page 249
- "endpwent, getpwent, setpwent - manage user catalog" on page 251
- "endutxent, getutxent, getutxid, getutxline, pututxline, setutxent - manage utmpx entries" on page 253
- "\_\_FILE\_\_ - macro for source file names" on page 300
- "getdtablesize - get size of descriptor table" on page 412
- "getegid - get effective group ID of process" on page 412
- "geteuid - get effective user ID of process" on page 414
- "getgid - get real group ID of process" on page 414
- "getgrgid - get group file entry for group ID" on page 415
- "getgrnam - get group file entry for group name" on page 417
- "getgroups - get supplementary group IDs" on page 419
- "gethostid - get ID of current host" on page 420
- "gethostname - get name of current host" on page 420

"getlogin - get login name" on page 423

"getpgmname - get program name (BS2000)" on page 432

"getpgid - get process group ID" on page 432

"getpgrp - get process group ID" on page 433

"getpid - get process ID" on page 433

"getppid - get parent process ID" on page 434

"getpriority, setpriority - get or set process priority" on page 435

"getpwnam - get user name" on page 437

"getpwuid - get user ID" on page 439

"getsid - get process group ID" on page 447

"gettsn - get TSN (task sequence number) (BS2000)" on page 450

"getuid - get real user ID" on page 450

"getutxent, getutxid, getutxline - get utmpx entry" on page 451

"\_\_LINE\_\_ - macro for current source program line number" on page 518

"setgid - set group ID of process" on page 700

"setpgid - set process group ID for job control" on page 707

"setpgrp - set process group ID" on page 708

"setregid - set real and effective group IDs" on page 709

"setreuid - set real and effective user IDs" on page 710

"setsid - create session and set process group ID" on page 712

"setuid - set user ID" on page 713

"\_\_STDC\_\_ - macro for ANSI conformance" on page 760

"\_\_STDC\_VERSION\_\_ - Amendment 1 conformity?" on page 760

"ttypslot - find entry of current user in utmp file" on page 845

"ulimit - get and set process limits" on page 849

### **Process control and signals**

"abort - abort process" on page 165

"alarm - schedule alarm signal" on page 172

"atexit - register function to run at process termination" on page 179



- "bs2exit - program termination with MONJV (BS2000)" on page 188
- "bsd\_signal - simplified signal handling" on page 191
- "exec: execl, execv, execl, execve, execlp, execvp - execute file" on page 259
- "exit, \_exit - terminate process" on page 264
- "fork - create new process" on page 318
- "kill - send signal to process or process group" on page 510
- "killpg - send signal to process group" on page 512
- "\_longjmp, \_setjmp - non-local jump (without signal mask)" on page 538
- "longjmp - execute non-local jump" on page 539
- "nice - change priority of process" on page 606
- "pause - suspend process until signal is received" on page 620
- "raise - send signal to calling process" on page 643
- "\_setjmp - set label for non-local jump (without signal mask)" on page 701
- "setjmp - set label for non-local jump" on page 702
- "sigaction - examine and change signal handling" on page 723
- "sigaddset - add signal to signal set" on page 731
- "sigaltstack - set/read alternative stack of signal" on page 732
- "sigdelset - delete signal from signal set" on page 734
- "sigemptyset - initialize and empty signal set" on page 735
- "sigfillset - initialize and fill signal set" on page 736
- "sighold, sigignore - add signal to signal mask / register SIG\_IGN for signal" on page 736
- "siginterrupt - change behavior of system calls in response to interrupts" on page 737
- "sigismember - test for member of signal set" on page 738
- "siglongjmp - execute non-local jump using signal" on page 739
- "signal - examine or change signal handling" on page 740
- "sigpause - remove signal from signal mask and deactivate process" on page 743
- "sigpending - examine pending signals" on page 743
- "sigprocmask - examine or change blocked signals" on page 744
- "sigrelse - remove signal from signal mask" on page 746

- "sigset - modify signal handling" on page 746
- "sigsetjmp - set label for non-local jump using signal" on page 747
- "sigstack - set or query alternative stack for signal" on page 749
- "sigsuspend - wait for signal" on page 750
- "sleep - suspend process for fixed interval of time" on page 752
- "wait, waitpid - wait for child process to stop or terminate" on page 873
- "vfork - generate new process in virtual memory" on page 868
- "wait3 - wait for status change of child processes" on page 877
- "waitid - wait for status change of child processes" on page 878

### **Interprocess communication**

- "ftok - interprocess communication" on page 374
- "mkfifo - create FIFO file" on page 571
- "msgctl - message control operations" on page 590
- "msgget - get message queue" on page 592
- "msgrcv - receive message from queue" on page 594
- "msgsnd - send message to queue" on page 596
- "pclose - close pipe stream" on page 621
- "pipe - create pipe" on page 623
- "popen - initiate pipe stream to or from process" on page 627
- "semctl - semaphore control operations" on page 689
- "semget - get semaphore ID" on page 692
- "semop - semaphore operations" on page 694
- "shmat - shared memory attach operation" on page 716
- "shmctl - shared memory control operations" on page 718
- "shmdt - shared memory detach operation" on page 720
- "shmget - create shared memory segment" on page 721

**Diagnostics and messages**

"assert - output diagnostic messages" on page 177

"catclose - close message catalog" on page 196

"catgets - read message" on page 197

"catopen - open message catalog" on page 198

"closelog, openlog, setlogmask, syslog - control system log" on page 217

"errno - variable for error return values" on page 258

"fmtmsg - output message to stderr and/or system console" on page 307

"perror - write error messages to standard error" on page 622

"strerror - get message string" on page 768

## Functions to support POSIX threads

### Reentrant POSIX thread functions

#### (`_POSIX_THREAD_SAVE_FUNCTIONS` group)

Functions with the "\_r" suffix in the name are functions that are the reentrant version of the corresponding function without the "\_r" suffix. Since these functions are also useful when working with threads, they are also supplied in the unthreaded version of CRTE (`$.SYSLNK.CRTE`).

"`asctime_r` - convert date and time to string (thread-safe)" on page 175

"`ctime_r` - thread-safe conversion of date and time to string" on page 233

"`getgrgid_r` - get group file entry for group ID (thread-safe)" on page 416

"`getgrnam_r` - get group file entry for group name (thread-safe)" on page 418

"`getlogin_r` - get login name (thread-safe)" on page 424

"`getpwnam_r` - get user name (thread-safe)" on page 438

"`gmtime_r` - convert date and time to UTC (thread-safe)" on page 457

"`localtime_r` - convert date and time to string (thread-safe)" on page 531

"`rand_r` - pseudo-random number generator (int, thread-safe)" on page 645

"`readdir_r` - read directory (thread-safe)" on page 652

"`strtok_r` - split string into tokens (thread-safe)" on page 789

"`ttyname` - find pathname of terminal" on page 843

These functions are to be used when working with threads instead of using the corresponding function that does not have the suffix "\_r". However, it is also advantageous to use the functions listed in a unthreaded environment.

### POSIX THREAD functions for locking and unlocking objects of type (FILE\*)

"`flockfile`, `ftrylockfile`, `funlockfile` - functions for locking standard input/output" on page 303

### POSIX thread functions for explicitly locking clients

The following functions are identical to the corresponding functions without "\_unlocked" in the name:

"getc\_unlocked, getchar\_unlocked, putc\_unlocked, putchar\_unlocked - standard I/O with explicit lock by the client" on page 398

In this case the user must guarantee thread safety himself by locking the object of type (FILE\*) used by calling the flockfile or ftrylockfile function and by calling the funlockfile function to unlock it.

### POSIX thread functions that affect a process or thread

There are two kinds of POSIX thread functions

- Functions that affect the process (just like before) and therefore affect all threads belonging to the process and
- Functions that only affect a special thread

You must also check when using signals if the signal is sent to the (entire) process or to a certain thread.

The following functions are available in this context:

"abort - abort process" on page 165

"alarm - schedule alarm signal" on page 172

"atexit - register function to run at process termination" on page 179

"exit, \_exit - terminate process" on page 264

"fcntl - control open file" on page 276

"fork - create new process" on page 318

"getcontext, setcontext - display or modify user context" on page 401

"getpid - get process ID" on page 433

"getrlimit - get limit value for a resource" on page 441

"getpriority - call process priority" on page 435

"kill - send signal to process or process group" on page 510

"lockf - lock file section" on page 532

"msgrcv - receive message from queue" on page 594

"msgsnd - send message to queue" on page 596

"nice - change priority of process" on page 606

"open - open file" on page 609

"pause - suspend process until signal is received" on page 620

"raise - send signal to calling process" on page 643

"read - read bytes from file" on page 647

"semop - semaphore operations" on page 694

"setcontext - modify user context" on page 699

"setlocale - set or query locale" on page 704

"sigaction - examine and change signal handling" on page 723

"sigpause - remove signal from signal mask and deactivate process" on page 743

"sigpending - examine pending signals" on page 743

"sigsetjmp - set label for non-local jump using signal" on page 747

"sigsuspend - wait for signal" on page 750

"sleep - suspend process for fixed interval of time" on page 752

"usleep - suspend process for defined interval" on page 859

"wait, waitpid - wait for child process to stop or terminate" on page 873

"wait3 - wait for status change of child processes" on page 877

"waitid - wait for status change of child processes" on page 878

"write - write bytes to file" on page 920

For the following functions the `SIGPIPE` signal is not sent to the process but is sent to the calling thread instead when an `EPIPE` error occurs:

"fclose - close stream" on page 274

"fflush - flush stream" on page 290

"fputc - put byte on stream" on page 334

"fputwc - put wide-character code on stream" on page 337

"fseek - reposition file position indicator in stream" on page 356

"write - write bytes to file" on page 920

## Functions that are not safe for threads

All functions that are defined in the C runtime library are safe for threads as delivered. The following functions are the only exceptions to this rule:

"asctime - convert date and time to string" on page 174 <sup>1</sup>

"basename - return last element of pathname" on page 184

"brk, sbrk - modify size of data segment" on page 186

"chroot - change root directory" on page 212

"ctime - convert date and time to string" on page 232 <sup>1</sup>

"cuserid - get login name" on page 234

"dbmclearerr - function for administering dbm databases" on page 236

"div - divide with integers" on page 241

"ecvt, fcvt, gcvt - convert floating-point number to string" on page 246

"endgrent, getgrent, setgrent - group management" on page 249

"endpwent, getpwent, setpwent - manage user catalog" on page 251

"endutxent, getutxent, getutxid, getutxline, pututxline, setutxent - manage utmpx entries" on page 253

"fcvt - convert floating-point number to string" on page 282

"gamma - compute logarithm of gamma function" on page 394

"gcvt - convert floating-point number to string" on page 395

"getdtablesize - get size of descriptor table" on page 412

"getenv - get value of environment variable" on page 413

"getgrent - get group file entry" on page 414

"getpwent - read user data from user catalog" on page 436

"getutxent, getutxid, getutxline - get utmpx entry" on page 451

"getgrgid - get group file entry for group ID" on page 415 <sup>2</sup>

"getgrnam - get group file entry for group name" on page 417 <sup>2</sup>

"getlogin - get login name" on page 423 <sup>2</sup>

"getpagesize - get current page size" on page 430

---

<sup>1</sup> use reentrant function ("\_r" extension)

<sup>2</sup> use reentrant function ("\_r" extension)

"getpass - read string of characters without echo" on page 431

"getpwnam - get user name" on page 437 <sup>2</sup>

"getw - read word from stream" on page 452

"initstate - generate pseudo-random number" on page 465 <sup>1</sup>

"localtime - convert date and time to local time" on page 530 <sup>2</sup>

"longjmp - execute non-local jump" on page 539 <sup>2</sup>

"ptsname - name of pseudoterminal" on page 629

"putenv - change or add environment variables" on page 632

"pututxline - write utmpx entry" on page 638

"putw - put word on stream" on page 639

"rand - pseudo-random number generator (int)" on page 645 <sup>2</sup>

"rand - pseudo-random number generator (int)" on page 645" <sup>3</sup>

"readdir - read directory" on page 650 <sup>2</sup>

"sbrk - modify size of data segment" on page 684

"setgrent - reset file position indicator to beginning of group file" on page 701

"setpwent - delete pointer to search user catalog" on page 708

"setutxent - reset pointer to utmpx file" on page 714

"siglongjmp - execute non-local jump using signal" on page 739 <sup>4</sup>

"signgam - variable for sign of lgamma" on page 743

"sigprocmask - examine or change blocked signals" on page 744 <sup>5</sup>

"sigset - modify signal handling" on page 746

"strtok - split string into tokens" on page 788 <sup>6</sup>

"ttyname - find pathname of terminal" on page 843 <sup>6</sup>

"ttypslot - find entry of current user in utmp file" on page 845

"wait3 - wait for status change of child processes" on page 877

---

<sup>1</sup> use `rand_r()` reentrant function

<sup>2</sup> The result of calling this function is undefined when the `jmp_buf` structure was not initialized in the calling thread.

<sup>3</sup> use `rand_r()` reentrant function

<sup>4</sup> The result of calling this function is undefined when the `jmp_buf` structure was not initialized in the calling thread.



**Note**

If you use one of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS` interfaces, you must call the `ctermid()` and `tmpnam()` functions with a parameter that is not equal to the null pointer in order to be thread-safe. Otherwise the result will be written to an internal static area, which can lead to an undefined response.

---

<sup>5</sup> use `pthread_sigmask` function

<sup>6</sup> use reentrant function ("\_r" extension)

## Memory management and memory operations

- "bcmp - compare memory areas" on page 185
- "bcopy - copy memory area" on page 185
- "brk, sbrk - modify size of data segment" on page 186
- "bzero - initialize memory with X'00'" on page 194
- "calloc - allocate memory" on page 195
- "free - free allocated memory" on page 341
- "garbcoll - release memory space to system (BS2000)" on page 395
- "malloc - memory allocator" on page 553
- "memalloc - memory allocator (BS2000)" on page 560
- "memchr - find byte in memory" on page 562
- "memcmp - compare bytes in memory" on page 563
- "memfree - free memory area (BS2000)" on page 565
- "memmove - copy bytes in memory with overlapping areas" on page 566
- "memset - initialize memory area" on page 567
- "mmap - map memory pages" on page 581
- "mprotect - modify access protection for memory mapping" on page 588
- "msync - synchronize memory" on page 598
- "munmap - unmap memory pages" on page 600
- "offsetof - get offset of structure component from start of structure (BS2000)" on page 608
- "realloc - memory reallocator" on page 656
- "swab - swap bytes" on page 800
- "valloc - request memory aligned with page boundary" on page 867

## System environment

- "bs2system - execute BS2000 command (extension)" on page 190
- "confstr - get string value of system variable" on page 220
- "\_edt - call EDT (BS2000)" on page 248
- "environ - external variable for environment" on page 256
- „fpathconf - get value of pathname variable" (see pathconf)
- "getcontext, setcontext - display or modify user context" on page 401
- "getenv - get value of environment variable" on page 413
- "getpagesize - get current page size" on page 430
- "getrlimit, setrlimit - get or set limit for resource" on page 441
- "getrusage - get information on usage of resources" on page 445
- "localeconv - change components of locale" on page 526
- "makecontext, swapcontext - set up user context" on page 551
- "mount - mount file system (extension)" on page 586
- "nl\_langinfo - get locale values" on page 607
- "pathconf, fpathconf - get value of pathname variable" on page 617
- "putenv - change or add environment variables" on page 632
- "setlocale - set or query locale" on page 704
- "setrlimit - set resource limit" on page 711
- "sysconf - get numeric value of configurable system variable" on page 805
- "sysfs - get information on file system type (extension)" on page 809
- "system - execute system command" on page 811
- "umount - unmount file system (extension)" on page 851
- "uname - get basic data on current operating system" on page 852

## Characters and strings

### Single character processing

"ffs - seek first set bit" on page 292

"isalnum - test for alphanumeric character" on page 483

"isalpha - test for alphabetic character" on page 484

"isascii - test for 7-bit ASCII character" on page 485

"iscntrl - test for control character" on page 487

"isdigit - test for decimal digit" on page 488

"isebcdic - test for EBCDIC character (BS2000)" on page 489

"isgraph - test for visible character" on page 490

"islower - test for lowercase letter" on page 491

"isprint - test for printing character" on page 492

"ispunct - test for punctuation character" on page 493

"isspace - test for white-space character" on page 494

"isupper - test for uppercase letter" on page 495

"iswalnum - test for alphanumeric wide character" on page 496

"iswalpha - test for alphabetic wide character" on page 497

"iswcntrl - test for control wide character" on page 498

"iswctype - test wide character for class" on page 499

"iswdigit - test for decimal digit wide character" on page 500

"iswgraph - test for visible wide character" on page 501

"iswlower - test for lowercase wide character" on page 502

"iswprint - test for printing wide character" on page 503

"iswpunct - test for punctuation wide character" on page 504

"iswspace - test for white-space wide character" on page 505

"iswupper - test for uppercase wide character" on page 506

"iswxdigit - test for hexadecimal digit wide character" on page 507

"isxdigit - test for hexadecimal digit" on page 508

"mblen - get number of bytes in multi-byte character" on page 554

"mbrlen - get number of bytes in multi-byte character" on page 554

"mbsinit - test for "initial conversion" state" on page 556

"wctype - define wide character class" on page 914

"wcwidth - get number of column positions of wide character code" on page 915

### **String processing**

"a64l, l64a - convert string to 32-bit integer" on page 163

"ascii\_to\_ebcdic - convert ASCII string to EBCDIC string (extension)" on page 173

"crypt - encode strings using algorithms" on page 226

"ebcdic\_to\_ascii - convert EBCDIC string to ASCII string (extension)" on page 245

"encrypt - encode strings blockwise" on page 248

"getsubopt - get suboptions from string" on page 448

"index - get first occurrence of character in string" on page 464

"rindex - get last occurrence of character in string" on page 679

"setkey - set encoding key" on page 703

"strcasecmp, strncasecmp - non-case-sensitive string comparison" on page 762

"strcat - concatenate two strings" on page 763

"strchr - scan string for characters" on page 763

"strcmp - compare two strings" on page 764

"strcoll - compare strings using collating sequence" on page 765

"strcpy - copy string" on page 766

"strcspn - get length of complementary substring" on page 766

"strdup - duplicate string" on page 767

"strfill - copy substring (BS2000)" on page 769

"strlen - get length of string" on page 777

"strlower - convert a string to lowercase letters (BS2000)" on page 777

"strncasecmp - non-case-sensitive string comparisons" on page 778

"strncat - concatenate two substrings" on page 778

"strncmp - compare two substrings" on page 779

"strncpy - copy substring" on page 780

"strpbrk - get first occurrence of character in string" on page 781

"strchr - get last occurrence of character in string" on page 785

"strspn - get length of substring" on page 786

"strstr - find substring in string" on page 786

"strtok - split string into tokens" on page 788

"strupper - convert string to uppercase letters (BS2000)" on page 798

"strxfrm - string transformation based on LC\_COLLATE" on page 799

"towctrans - map wide characters" on page 838

"wcscat - concatenate two wide character strings" on page 881

"wcschr - scan wide character string for wide characters" on page 882

"wcscmp - compare two wide character strings" on page 883

"wccoll - compare two wide character strings according to LC\_COLLATE" on page 884

"wcscspn - get length of complementary wide character substring" on page 886

"wcslen - get length of wide character string" on page 888

"wcsncat - concatenate two wide character strings" on page 889

"wcsncmp - compare two wide character substrings" on page 890

"wcsncpy - copy wide character substring" on page 891

"wcpbrk - get first occurrence of wide character in wide character string" on page 892

"wcsrchr - get last occurrence of wide character in wide character string" on page 893

"wcssp - get length of wide character substring" on page 895

"wcsstr - search for first occurrence of a wide character string" on page 896

"wcstok - split wide character string into tokens" on page 899

"wscwsc - find wide character substring in wide character string" on page 909

"wscwidth - get number of column positions of wide character string" on page 910

"wctrans - define wide character mappings" on page 913

"wmemchr - search for wide character in a wide character string" on page 916

"wmemcmp - compare two wide character strings" on page 917

"wmemcpy - copy wide character string" on page 917

"wmemmove - copy wide character string in overlapping area" on page 918

"wmemset - set first *n* wide characters in wide character string" on page 918

### **Character and string conversions**

"btowc - (one byte) convert multi-byte character to wide character" on page 193

"iconv - code conversion function" on page 461

"iconv\_close - deallocate code conversion descriptor" on page 462

"iconv\_open - allocate code conversion descriptor" on page 463

"mbrtowc - complete and convert multi-byte string to wide-character string" on page 555

"mbsrtowcs - convert multi-byte string to wide-character string" on page 557

"mbtowc - convert multi-byte character to wide character" on page 559

"strftime - convert date and time to string" on page 774

"strptime - convert string to date and time" on page 782

"\_tolower - convert uppercase letters to lowercase" on page 836

"tolower - convert characters to lowercase" on page 837

"\_toupper - convert lowercase letters to uppercase" on page 837

"toupper - convert characters to uppercase" on page 838

"towlower - convert wide characters to lowercase" on page 839

"towupper - convert wide characters to uppercase" on page 839

"wctomb - convert wide characters to multi-byte characters" on page 880

"wcsrtombs - convert wide character string to multi-byte string" on page 894

"wcstombs - convert wide character string to character string" on page 904

"wctomb - convert wide character code to character" on page 912

"wctob - convert wide character to 1-byte multi-byte character" on page 912

## Conversion of entities

- "atof - convert string to double-precision number" on page 180
- "atoi - convert string to integer" on page 181
- "atol - convert string to long integer" on page 182
- "atoll - convert string to long long integer (long long int)" on page 183
- "ecvt, fcvt, gcvt - convert floating-point number to string" on page 246
- "fcvt - convert floating-point number to string" on page 282
- "gcvt - convert floating-point number to string" on page 395
- "getdents - convert directory entries" on page 410
- "getsubopt - get suboptions from string" on page 448
- "l64a - convert 32-bit integer number to string" on page 513
- "strftime - convert date and time to string" on page 774
- "strtod - convert string to double-precision number" on page 787
- "strtol - convert string to long integer" on page 790
- "strtoll - convert string to long long integer" on page 792
- "strtoul - convert string to unsigned long integer" on page 794
- "strtoull - convert string to unsigned long long" on page 796
- "toascii - convert integer to legal value" on page 835
- "toebcdic - convert integer to legal value (BS2000)" on page 836
- "wcsftime - convert date and time to wide character string" on page 887
- "wcstod - convert wide character string to double-precision number" on page 897
- "wcstol - convert wide character string to long integer" on page 900
- "wcstoll - convert wide character string to long long integer" on page 902
- "wcstoul - convert wide character string to unsigned long" on page 905
- "wcstoull - convert wide character string to unsigned long long" on page 907



## Regular expressions

- "advance - pattern match given compiled regular expression" on page 171
- "compile - produce compiled regular expression" on page 219
- "loc1, loc2 - pointers to characters matched by regular expressions" on page 525
- "locs - stop regular expression matching in string" on page 535
- "re\_comp, re\_exec - compile and execute regular expressions" on page 658
- "regcmp, regex - compile and execute regular expression" on page 661
- "regexp: advance, compile, step, loc1, loc2, locs - compile and match regular expressions" on page 664
- "step - compare regular expressions" on page 762

## Time functions

- "asctime - convert date and time to string" on page 174
- "clock - report CPU time used by a process" on page 214
- "cputime - calculate CPU time used by current task (BS2000)" on page 222
- "ctime - convert date and time to string" on page 232
- "\_\_DATE\_\_ - macro for compilation date" on page 235
- "daylight - daylight savings time variable" on page 235
- "difftime - compute difference between two calendar time values" on page 239
- "gmtime - convert date and time to UTC" on page 456
- "ftime - get date and time" on page 372
- "getdate - convert time and date to user format" on page 405
- "getitimer, setitimer - read or set" on page 421
- "getsubopt - get suboptions from string" on page 448
- "localtime - convert date and time to local time" on page 530
- "mktime - convert local time into time since the Epoch" on page 579
- "\_\_TIME\_\_ - macro for compilation time" on page 829
- "time - get time since the Epoch" on page 830

"timezone - variable for difference between local time and UTC" on page 832

"tzname - array variable for timezone strings" on page 846

"tzset - set timezone conversion information" on page 847

"ualarm - set interval timer" on page 848

"usleep - suspend process for defined interval" on page 859

## Math functions

### Arithmetic with integers

"abs - return integer absolute value" on page 166

"div - divide with integers" on page 241

"labs - return long integer absolute value" on page 513

"ldiv - long division of integers" on page 517

"llabs - return absolute value of an integer (long long int)" on page 521

"lldiv - division of integers (long long int)" on page 522

"llrint, llrintf, llrintl - round to nearest integer value (long long int)" on page 523

"llround, llroundf, llroundl - round up to next integer value (long long int)" on page 524

"lrint, lrintf, lrintl - round to nearest integer value (long int)" on page 541

"lround, lroundf, lroundl - round up to next integer value (long int)" on page 542

"rint, rintf, rintl - round to nearest integer value" on page 680

"round, roundf, roundl - round up to next integer value" on page 683

### Arithmetic with floating-point numbers

"cabs - calculate absolute value of complex number (BS2000)" on page 194

"ceil, ceilf, ceill - round up floating-point number" on page 201

"cbrt - cube root" on page 199

"errno - variable for error return values" on page 258

"exp - use exponential function" on page 267

"expm1 - compute exponential function" on page 267

"fabs - compute absolute value of floating-point number" on page 268  
"floor, floorf, floorl- round off floating point number" on page 305  
"fmod - compute floating-point remainder value function" on page 306  
"frexp - extract mantissa and exponent from double precision number" on page 344  
"gamma - compute logarithm of gamma function" on page 394  
"hypot - Euclidean distance function" on page 460  
"ilogb - get exponent part of floating-point number" on page 464  
"isnan - test for NaN (not a number)" on page 491  
"j0, j1, jn - Bessel functions of first kind" on page 509  
"ldexp - load exponent of floating-point number" on page 516  
"lgamma - compute logarithm of gamma function" on page 518  
"log - natural logarithm function" on page 536  
"log10 - base 10 logarithm function" on page 536  
"log1p - compute natural log" on page 537  
"logb - get exponent part of floating-point number" on page 537  
"modf - split floating-point number into integral and fractional parts" on page 585  
"nextafter - next displayable floating-point number" on page 602  
"pow - power function" on page 628  
"remainder - remainder from division" on page 671  
"rint, rintf, rintl - round to nearest integer value" on page 680  
"scalb - load exponent of base-independent floating-point number" on page 684  
"signgam - variable for sign of lgamma" on page 743  
"sqrt - square root function" on page 754  
"y0, y1, yn - Bessel functions of the second kind" on page 927

### **Trigonometric, hyperbolic and arc functions**

"acos - arc cosine function" on page 169  
"acosh, asinh, atanh - inverse hyperbolic functions" on page 170  
"asin - arc sine function" on page 176  
"atan - arc tangent function" on page 177

"atan2 - arc tangent of x/y" on page 178

"cos - cosine function" on page 221

"cosh - hyperbolic cosine function" on page 221

"sin - sine function" on page 751

"sinh - hyperbolic sine function" on page 751

"tan - compute tangent" on page 814

"tanh - compute hyperbolic tangent" on page 814

### Random numbers

"drand48 - generate pseudo-random numbers between 0.0 and 1.0" on page 242

"erand48 - generate pseudo-random numbers between 0.0 and 1.0 with initialization value" (see drand48)

"initstate, random, setstate, srandom - generate pseudo-random numbers" on page 465

"jrand48 - generate pseudo random numbers between  $-2^{31}$  and  $2^{31}$  with initialization value" (see drand48)

„lcong48 - pseudo-random number (signed long int) generator" (see drand48)

"lrand48 - generate pseudo random numbers between 0 and  $2^{31}$ " (see drand48)

"mrand48 - generate pseudo random numbers between  $-2^{31}$  and  $2^{31}$ " (see drand48)

"nrand48 - generate pseudo random numbers between 0 and  $2^{31}$  with initialization value" (see drand48)

"rand - pseudo-random number generator (int)" on page 645

"seed48 - set seed (int) for pseudo-random numbers" on page 685" (see drand48)

"srand - generate pseudo-random numbers with seed" on page 754

„srand48 - seed (double-precision) pseudo-random number generator" (see drand48)

## Search and sort procedures

"bsearch - conduct binary search of sorted array" on page 192

"hsearch, hcreate, hdestroy - manage hash tables" on page 459

„lfind - find entry in linear search table" (see lsearch)

"lsearch, lfind - linear search and update" on page 543

"qsort - sort table of data" on page 642

„tdelete - delete node from binary search tree" (see tsearch)

„tfind - find node in binary search tree" (see tsearch)

"tsearch, tfind, tdelete, twalk - process binary search trees" on page 841

"twalk - traverse binary search tree" on page 846

"wcsoll - compare two wide character strings according to LC\_COLLATE" on page 884

## Terminal interface and data transmissions

- "cfgetispeed - get input baud rate" on page 204
- "cfgetospeed - get output baud rate" on page 204
- "cfsetispeed - set input baud rate" on page 205
- "cfsetospeed - set output baud rate" on page 206
- "ctermid - generate pathname for controlling terminal" on page 231
- "grantpt - grant access to the slave pseudoterminal" on page 458
- "isascii - test for 7-bit ASCII character" on page 485
- "ptsname - name of pseudoterminal" on page 629
- "tcdrain - wait for transmission of output" on page 815
- "tcflow - suspend or restart data transmission" on page 816
- "tcflush - discard non-transmitted data" on page 817
- "tcgetattr - get parameters associated with terminal" on page 818
- "tcgetpgrp - get foreground process group ID" on page 819
- "tcgetsid - get session ID of specified terminal" on page 820
- "tcsetpgrp - set foreground process group ID" on page 824
- "ttyname - find pathname of terminal" on page 843
- "unlockpt - remove lock from master/slave pseudoterminal pair" on page 858

## Database functions

"dbm\_clearerr, dbm\_close, dbm\_delete, dbm\_error, dbm\_fetch, dbm\_firstkey, dbm\_nextkey, dbm\_open, dbm\_store - functions for managing dbm databases" on page 236

## List processing

"insque, remque - Insert element in queue or remove element from queue" on page 467

## POSIX-IO macros

For functions in the C library that work with data, you must determine if the file is a POSIX file system file or a BS2000 file before executing the actual function. If you already know that you will only be working with files from the POSIX file system, then you can save yourself the effort and obtain better performance in doing so. For the following macros CRTE V02.3A contains a special version of the macro for working with files in the POSIX file system:

`getc(p)`:read characters from a file

`getchar()`:read characters from standard input

`putc(x, p)`:write characters to file

`putchar(x)`write characters to standard output

`clearerr(p)`:clear end-of-file and error flags

`feof(p)`: test for end-of-file

`ferror(p)`:test for file error

`fileno(p)`:get file descriptor

The macros are still stored in the `<stdio.h>` header as before. In order to generate the POSIX variant, the user must set the `__POSIX_MACROS` define before the `<stdio.h>` header file is included.



---

# Functions and variables in alphabetical order (a - m)

This chapter contains detailed descriptions, in alphabetical order, of the functions, macros and external variables that are supported by the C runtime system in both the POSIX subsystem as well as in BS2000/OSD.

## Format of entries

Each description begins with a title containing the symbolic name and some keywords to describe the functionality and is always followed by the same subsections:

Syntax	<p>Syntax of the function call or variable declaration and of the header file in which the relevant interface is defined or declared.</p> <p>A syntax line may be additionally identified as follows:</p> <p><i>Optional</i></p> <p>An include statement identified as optional need not be specified in newly created source code. Nor need it be deleted from existing source code. The end of each such section is indicated by the end marker. □</p>
Description	<p>Describes the functionality of the respective function, macro or external variable and explains the arguments to be specified.</p>
Return value	<p>Lists and describes the possible return values of a function.</p> <p>Some functions do not have a return value. In such cases and when describing external variables, the "Return value" section is omitted.</p>
Errors	<p>Lists and describes the error codes stored in the external variable <code>errno</code> in the case of an invalid function call or an execution error.</p> <p>Some functions do not store an error code in <code>errno</code> in the event of an error. In such cases and when describing external variables, the "Errors" section is omitted.</p>
Notes	<p>Typically includes explanations of concepts, information on interaction with other functions, and/or tips concerning application usage. This section may be omitted in some cases.</p>

See also      Contains cross-references to function descriptions, header files, sections in the chapters on concepts and other manuals.

Text segments that are not specially identified describe XPG4-conformant implementations. Extensions and deviations with respect to the Standard are indicated by the following markers:

*BS2000*

Information on extensions of the C runtime system describing functionality in connection with access to DMS and C runtime versions up to V2.1C (i.e. BS2000 functionality). The end of each such section is indicated by the end marker. □

*Extension*

Information on extensions of the C runtime system. The end of each such section is indicated by the end marker □.

If a function is an extension, as supported on many UNIX systems, it is marked as such: (*extension*).

*Restriction*

Information on current restrictions of the C runtime system as opposed to the XPG4 standard. The end of each such section is indicated by the end marker. □

## a64l, l64a - convert string to 32-bit integer

**Syntax**      `#include <stdlib.h>`  
                  `long a64l (const char *s);`  
                  `char *l64a (long value);`

**Description** These functions are used to manage numbers stored in radix-64 ASCII characters. These characters define a notation with which long integers can be represented by a maximum of six characters; each character represents a 'digit' in a base 64 notation.

The characters used to represent 'digits' are . for 0, / for 1, 0 through 9 for 2-11, A through Z for 12-37 and a through z for 38-63.

`a64l()` expects a pointer to a base 64 representation ending in a null byte, and returns the corresponding `long` value. If the string pointed to by `s` contains more than six characters, `a64l()` uses the first six characters. If the passed string was empty, the return value is 0L.

`a64l()` runs through the string from left to right (with the least significant digit on the left) and decodes each character as a 6-bit number in base 64. If the type `long` contains more than 32 bits, the result is prefixed with a sign. The behavior of `a64l()` is undefined if `s` is the null pointer or if the string pointed to by `s` was not generated by a previous call of `l64a()`.

`l64a()` expects a `long` argument and returns a pointer to the corresponding base 64 representation. If the argument is 0, `l64a()` returns a pointer to a null string. The behavior of `l64a()` is undefined if the value of the argument is negative.

### Return value

`a64l()`:

Integer value of type `long`

for strings with a structure like the one described above.

0L for empty strings.

Undefined if `s` is the null pointer or if the string was not generated by a previous call of `l64a()`. `errno` is set to indicate the error.

`l64a()`:

Pointer to a string represented in base 64

for `value > 0`

Pointer to an empty string

for `value = 0`

Undefined for `value < 0`

**Errors**      `a64l()` fails if the following applies:

`ERANGE`      The result cannot be represented.

**Notes**      The value returned by `l64a()` is a pointer to a static buffer, whose contents are overwritten with each call.

If the type `long` contains more than 32 bits, the result of `a64l(l64a(1))` occupies the 32 least significant bits.

**See also**    `strtoul()`, `stdlib.h`

## abort - abort process

Syntax      `#include <stdlib.h>`  
`void abort(void);`

Description If the function is called with POSIX functionality, its behavior conforms with XPG4 as described below:

- If the signal `SIGABRT` is not being caught and the signal handler does not return, `abort()` causes abnormal process termination to occur. The `SIGABRT` signal is sent to the calling process as if by means of the `raise()` function with the argument `SIGABRT`. Before the process is terminated, all open streams and message catalog descriptors are closed as if by a call to `fclose()`, and the default actions defined for `SIGABRT` are performed (see `signal.h`).
- The status made available to `wait()` or `waitpid()` by `abort()` will be that of a process terminated by the `SIGABRT` signal. The `abort()` function will override blocking or ignoring the `SIGABRT` signal.
- Process abort functions registered with `atexit()` are not called.

If threads are used, then the function affects the process or a thread in the following manner:

- The process is aborted and all its threads are aborted with it.

*BS2000*

- The following deviation in behavior must be noted when the function is called with BS2000 functionality:

If the program does not provide any signal handling function or if such a function returns to the interrupt point, the process is aborted with `_exit(-1)`. □

Notes      Catching the signal is intended to provide the application writer with a portable means to abort processing, free from possible interference from any proprietary library functions.

If `SIGABRT` is neither caught nor ignored, and the current directory is writable, a core dump may be produced.

See also    `atexit()`, `exit()`, `kill()`, `raise()`, `signal()`, `stdlib.h`, section "Signals" on page 111.

## abs - return integer absolute value

**Syntax**      `#include <stdlib.h>`  
                 `int abs(int i);`

**Description** `abs()` computes the absolute value of an integer *i*.

**Return value**  
Absolute value of *i* if successful.

**Notes**        The absolute value of the negative integer with the largest magnitude is not representable. If a negative number with the highest magnitude ( $-2^{31}$ ) is specified as the argument *i*, the program will terminate with an error.

**See also**     `cabs()`, `fabs()`, `labs()`, `stdlib.h`.

## access - check access permissions for file

**Syntax**      `#include <unistd.h>`  
                  `int access(const char *path, int amode);`

**Description** `access()` checks the file named by the *path* argument for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group

The following symbolic constants can be specified for *amode*:

R_OK	to check for read permission
W_OK	to check for write permission
X_OK	to check for execute (search) permission
F_OK	to check if the file exists

The value of *amode* is either the bit-wise inclusive OR of the access permissions to be checked (R\_OK, W\_OK, X\_OK) or the existence test, F\_OK (see also `unistd.h`).

### Extension

Other values for *amode* may be permitted in addition to those listed above (e.g. if a system has extended access controls). □

A process with appropriate privileges may search a file even if none of the execute file permission bits are set, but success will not indicated when checking for X\_OK.

### Return value

0	The requested access is permitted.
-1	The requested access is not permitted; <code>errno</code> is set to indicate the error.

### Errors

`access()` will fail if:

EACCES	Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix.
--------	--

### Extension

EFAULT	<i>path</i> is an invalid address.
EINTR	A signal was caught during the <code>access()</code> system call. □
EINVAL	An attempt was made to access a BS2000 file.
ELOOP	The maximum number of symbolic links in <i>path</i> was exceeded (or the maximum number of symbolic links is defined by <code>MAXSYMLINKS</code> in the header file <code>sys/param.h</code> ).

ENAMETOOLONG

The length of the path argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}`.

ENOENT

The *path* argument points to a non-existent file or to an empty string.

ENOTDIR

A component of the path prefix is not a directory.

EROFS

Write access is requested for a file on a read-only file system.

**Notes** `access()` is executed for POSIX files only.

**See also** `chmod()`, `stat()`, `unistd.h`.



## acos - arc cosine function

**Syntax**      `#include <math.h>`

`double acos(double x);`

**Description** `acos()` is the inverse function of `cos()`. It returns the principal value (i.e. corresponding angle in radians) of the arc cosine of a floating-point number  $x$  in the range  $[-1.0, +1.0]$ .

**Return value**

`arc cosine( $x$ )`

if successful. A floating-point number of type `double` in the range  $[0, \pi]$  is returned.

0              if  $x$  does not lie in the range  $[-1.0, +1.0]$ . `errno` is set to indicate the error.

**Errors**      `acos()` will fail if:

EDOM              The value of  $x$  is not in the range  $[-1.0, +1.0]$ .

**Note**              To make sure you catch an error, you should set `errno` to 0 before calling `acos()`. If after execution `errno`  $\neq$  0, an error has occurred.

**See also**      `asin()`, `atan()`, `atan2()`, `cos()`, `sin()`, `tan()`, `math.h`.

## acosh, asinh, atanh - inverse hyperbolic functions

**Syntax**      `#include <math.h>`

```
double acosh (double x);  
double asinh (double x);  
double atanh (double x);
```

**Description** `acosh()`, `asinh()` and `atanh()` compute respectively the inverse hyperbolic cosine, the inverse hyperbolic sine and the inverse hyperbolic tangent for the argument *x*.

**Return value**

`acosh()`:

`Arch(x)`      if successful.

`0.0`            if  $x < 1.0$ . `errno` is set to indicate the error.

`asinh()` :

`Arsh(x)`      the function is always successful.

`atanh()`:

`Arth(x)`      if successful.

`0.0`            if  $|x| > 1.0$ . `errno` is set to indicate the error.

**Errors**      `acosh()` will fail if:

`EDOM`           $x < 1.0$ .

`atanh()` will fail if:

`EDOM`           $|x| > 1.0$ .

**See also**    `cosh()`, `sinh()`, `tanh()`, `math.h`.

## advance - pattern match given compiled regular expression

Syntax        `#include <regex.h>`  
              `int advance(const char *string, const char *exbuf);`

Description See `regex()`.

Notes         This function will not be supported in future issues of the X/Open standard.

## alarm - schedule alarm signal

Syntax `#include <unistd.h>`

*Optional*

`#include <signal.h>`  $\square$

`unsigned int alarm(unsigned int seconds);`

Description `alarm()` causes the system to send the calling process a `SIGALRM` signal after the number of real-time seconds specified by *seconds* have elapsed (see also `signal.h`).

If *seconds* is 0, a pending alarm request, if any, is cancelled.

Alarm requests are not stacked; only one `SIGALRM` generation can be scheduled in this manner; if the `SIGALRM` signal has not yet been generated, the call will result in rescheduling the time at which the `SIGALRM` signal will be generated.

Interactions between `alarm()` and the functions `setitimer()`, `ualarm()` and `usleep()` are not defined.

If threads are used, then the function affects the process or a thread in the following manner:

- A `SIGALRM` signal is generated for the process when the specified time limit has expired.
- BS2000*
- If the signal is not caught (see also `signal()`), the program is terminated with `exit(-1)`.  $\square$

Return value

Number of seconds until the generation of a `SIGALRM` signal  
if there is a previous `alarm` request with time remaining on the alarm clock.

0 if there is no pending `alarm` request.

`alarm()` is always successful.

Notes

`fork()` clears pending alarms in the child process. A new process image created by one of the `exec` functions inherits the time left to an alarm signal in the old process image.

Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

*BS2000*

`SIGALRM` corresponds to the `STXIT` event class `RTIMER` (real-time interval timer).  $\square$

See also

`exec()`, `fork()`, `pause()`, `sigaction()`, `signal.h`, `unistd.h`, section "Signals" on page 111.

## altzone - variable for time zone *(extension)*

**Syntax**        `#include <time.h>`  
                 `extern long int altzone;`

**Description** The external variable `altzone` contains the difference, in seconds, between UTC (Universal Time Coordinated, January 1, 1970) and the alternative time zone.  
  
By default, `altzone` is 0 (UTC).  
  
`altzone` is set by `tzset()`.

**See also**        `asctime()`, `ctime()`, `daylight`, `environ`, `gmtime()`, `localtime()`, `setlocale()`, `timezone`, `tzname`, `tzset()`ms, `time.h`.

## ascii\_to\_ebcdic - convert ASCII string to EBCDIC string *(extension)*

**Syntax**        `int ascii_to_ebcdic(char *in, char *out);`

**Description** `ascii_to_ebcdic` converts ASCII strings to EBCDIC strings, where *in* is the input string in ASCII, and *out* is the output string in EBCDIC. The buffer must be supplied by the caller.  
  
The characters of the input string are interpreted as ASCII characters and translated into the corresponding characters of the EBCDIC code.

**Return value**  
0                if successful.  
1                if an error occurs.

**See also**        `ebcdic_to_ascii`.

## asctime - convert date and time to string

**Syntax** `#include <time.h>`

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

**Description** `asctime()` converts a time specification that is broken down in accordance with the structure `tm` (see below) into an EBCDIC string.

This structure can be specified with `*timeptr` as defined in the header `time.h`:

```
struct tm
{
    int    tm_sec;        /* Seconds [0,61] */
    int    tm_min;        /* Minutes [0,59] */
    int    tm_hour;       /* Hours [0,23] */
    int    tm_mday;       /* Day of month [1,31] */
    int    tm_mon;        /* Months since beginning of year [0,11]*/
    int    tm_year;       /* Years since 1900 */
    int    tm_wday;       /* Weekday [0,6] Sunday=0 */
    int    tm_yday;       /* Days since January 1 [0,365] */
    int    tm_isdst;      /* Daylight saving time (always 0) */
};
```

`asctime()` is not thread-safe. Use the reentrant function `asctime_r()` when needed.

**Return value**

Pointer to the generated EBCDIC string

if successful. The result string has a length of 26 (including the null byte) and the format of a date and time specification in English:

*weekday month-name day-of-month hours:minutes:seconds year*

e.g. Thu Jun 30 15:20:54 1994\n\n0

**Notes**

The functions `asctime()`, `ctime()`, `gmtime()` and `localtime()` write their results to an internal C data area. This data area is therefore overwritten each time a function of this group is called.

A structure of type `tm` is returned by the functions `gmtime()` and `localtime()`. These functions continue to be offered for reasons of compatibility. They support neither localized date formats nor localized time formats, i.e. regional peculiarities in the representation of the date or time. To be portable, applications should use the `strftime()` function instead.

**See also**

`ctime()`, `difftime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `time()`, `utime()`, `time.h`.

## asctime\_r - convert date and time to string (thread-safe)

**Syntax**      `#include <time.h>`

```
char *asctime_r(const struct tm *tm, char *buf);
```

**Description** `asctime_r()` converts a time specification pointed to by *tm* into the same form as `asctime()` and writes the result into the data area pointed to by *buf* (with at least 26 bytes).

**Return value**

Pointer to the string that *buf* points to  
if successful.

Null pointer      if unsuccessful.

**See also**      `asctime()`, `ctime()`, `ctime_r()`, `localtime()`, `localtime_r()`, `time()`.

## asin - arc sine function

**Syntax**     `#include <math.h>`  
              `double asin(double x);`

**Description** `asin()` is the inverse function of `sin()`. It returns the principal value (i.e. corresponding angle in radians) of the arc sine of a floating-point number  $x$  in the range  $[-1.0, +1.0]$ .

**Return value**  
    `arc sine(x)`  
                  if successful. A floating-point number of type `double` in the range  $[-\pi/2, +\pi/2]$  is returned.  
    0.0           for values of  $x$  that are not in the range  $[-1.0, +1.0]$ .  
                  `errno` is set to indicate the error.  
    0.0           if the result causes an underflow.

**Errors**     `asin()` will fail if:  
    EDOM         The value of  $x$  is not in the range  $[-1.0, +1.0]$ .

**Notes**     To be sure of catching an error, you should set `errno` to 0 before calling `asin()`. If after execution `errno`  $\neq$  0, an error has occurred.

**See also**   `acos()`, `atan()`, `atan2()`, `cos()`, `sin()`, `tan()`, `math.h`.

## asinh - inverse hyperbolic sine function

**Syntax**     `#include <math.h>`  
              `double asinh (double x);`

**Description** See `acosh()`.



## assert - output diagnostic messages

**Syntax**      `#include <assert.h>`

`void assert(int expression);`

**Description** `assert()` is implemented as a macro. When it is executed, it checks whether *expression* evaluates to false (0) at a specific position in the program. If an error occurs, `assert()` writes a comment about the particular call that failed on `stderr` and calls `abort()`. The message includes the text of the argument, the name of the source file (`__FILE__`), and the source file line number (`__LINE__`).

**Notes**      `assert` calls are not executed if `NDEBUG` is defined. This can be done by the following methods:

- by specifying a preprocessor option when calling the compiler (see the C and C++ User Guides)
- by inserting a preprocessor control statement `#define NDEBUG` in the source program before the `#include <assert.h>` statement.

**See also**      `abort()`, `__FILE__`, `__LINE__`, `stderr()`, `assert.h`.

## atan - arc tangent function

**Syntax**      `#include <math.h>`

`double atan(double x);`

**Description** `atan()` is the inverse function of `tan()`. It returns the principal value (i.e. corresponding angle in radians) of the arc tangent of a floating-point number *x*.

**Return value**

`arc tangent(x)`

if successful. A floating-point number of type `double` in the range  $[-\pi/2, +\pi/2]$  is returned.

**See also**      `acos()`, `asin()`, `atan2()`, `cos()`, `sin()`, `tan()`, `math.h`.

## atan2 - arc tangent of x/y

Syntax `#include <math.h>`

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

Description `atan2()` computes the arc tangent of  $x/y$ , using the signs of both arguments to determine the quadrant of the return value.

$x$  is the dividend of the expression for which the arc tangent is to be calculated.

$y$  is the divisor of the expression for which the arc tangent is to be calculated.

Return value

arc tangent( $x/y$ )

if neither argument is 0.0.

A floating-point number of type `double` in the range  $[-\pi/2, +\pi/2]$  is returned.

$-\pi/2$  or  $+\pi/2$

if the divisor is 0.0, depending on the sign of the dividend.

0

if the dividend is 0.0.

$\pi/2$

if both arguments are 0.0. `errno` is set to indicate the error.

Errors `atan2()` will fail if:

EDOM Both arguments are 0.0.

See also `acos()`, `asin()`, `atan()`, `cos()`, `sin()`, `tan()`, `math.h`.

## atanh - inverse hyperbolic tangent function

Syntax `#include <math.h>`

```
double atanh (double x);
```

Description See `acosh()`.

## atexit - register function to run at process termination

**Syntax**      `#include <stdlib.h>`  
`int atexit(void ( *func) (void));`

**Description** `atexit()` registers a function *func()* to be called without arguments at normal process termination. The registered functions are called in the reverse order to that in which they were registered. Functions for which multiple registrations exist are called more than once.

Functions registered with `atexit()` are called only if the process is terminated "normally" by one of the following methods:

- an explicit `exit()` call
  - termination of the `main` function without an explicit `exit` call
- BS2000*
- Process termination by the C runtime system with `exit(-1)`, i.e. on occurrence of a raise signal (not `SIGABRT`) which is handled by the default signal handling mechanism via `SIG_DFL` (see `signal()`) or not handled at all.

Up to 40 functions can be registered.

Following a successful call to any of the `exec` functions, all functions previously registered by `atexit()` will no longer be registered.

**Return value**

0	if the function is registered successfully.
≠ 0	if an error occurs.

**Notes** Functions registered by a call to `atexit()` must return to ensure that all registered functions are called.

The `sysconf()` function returns the value of `ATEXIT_MAX`, which specifies the total number of functions that can be registered. However, it is not possible to find out how many functions have already been registered (except by counting them).

**See also** `bs2exit()`, `exit()`, `signal()`, `stdlib.h`.

## atof - convert string to double-precision number

Syntax `#include <stdlib.h>`

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

Description `atof()` converts an EBCDIC string pointed to by `str` into a floating-point number of type `double`. The string to be converted may be formatted as follows:

$$[tab \dots] \left[ \begin{matrix} + \\ - \end{matrix} \right] [digit \dots] [.] [digit \dots] \left[ \begin{matrix} E \\ e \end{matrix} \right] \left[ \begin{matrix} + \\ - \end{matrix} \right] digit \dots$$

All control characters for white space are legal for `tab` (see definition under `isspace()`).

The `atof(str)` function differs from `strtod(str, (char**)NULL)` only in error handling.

Return value

Floating-point number of type `double`

for strings formatted as described above and representing a numeric value that is within the permissible floating point range.

*Extension*

0 for strings which do not correspond to the syntax described above.

HUGE\_VAL for strings whose numeric value lies outside the permissible floating-point range. `errno` is set to indicate the error.

Errors `atof()` will fail if:

ERANGE The return value causes an overflow or underflow. □

Notes `atof()` is completely contained in `strtod()`. However, the function continues to be offered because it is used in many existing applications.

The decimal character in the string to be converted is affected by the locale (category `LC_NUMERIC`). The decimal point is the default.

`atof()` also recognizes strings that begin with digits but then end with any character. `atof()` cuts off the numeric part, converts it according to the above description, and ignores the rest.

See also `atoi()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`, `stdlib.h`.

## atoi - convert string to integer

**Syntax**      `#include <stdlib.h>`

`int atoi(const char *str);`

**Description** `atoi()` converts an EBCDIC string to which `str` points into an integer. The string to be converted may be formatted as follows:

$$[tab \dots] \left[ \begin{array}{c} + \\ - \end{array} \right] digit \dots$$

All characters that produce white space are legal for `tab` (see `isspace()`).

The `atoi(str)` function differs from `strtol(str, (char**)NULL)` only in error handling.

**Return value**

Integer value of type `int`

for strings formatted as described above and representing a numeric value that lies in the permissible range of integers.

0 for strings that do not conform to the syntax described above.

`INT_MAX` or `INT_MIN`

in the case of an overflow, depending on the sign.

**Notes** `atoi()` is completely contained in `strtol()`. However, the function continues to be offered because it is used in many existing applications.

`atoi()` also recognizes strings that begin with digits but then end with any character.

`atoi()` cuts off the numeric part, converts it according to the above description, and ignores the rest.

**See also** `atof()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`, `stdlib.h`.

## atol - convert string to long integer

**Syntax**      `#include <stdlib.h>`  
`long int atol(const char *str);`

**Description** `atol()` converts an EBCDIC string to which *str* points into an integer of type `long`. The string to be converted may be formatted as follows:

$$[tab \dots] \left[ \begin{array}{c} + \\ - \end{array} \right] digit \dots$$

All characters that produce white space are legal for *tab* (see the definition under `isspace()`).

The `atol(str)` function differs from `strtol(str, (char**)NULL, 10)` only in error handling.

**Return value**

Integer value of type `long`  
for strings formatted as described above and representing a numeric value.

0  
for strings that do not conform to the syntax described above.

`LONG_MAX` or `LONG_MIN`  
in the case of an overflow, depending on the sign.

**Notes** `atol()` is completely contained in `strtol()`. However, the function continues to be offered because it is used in many existing applications.

`atol()` also recognizes strings that begin with digits but then end with any character. `atol()` cuts off the numeric part, converts it according to the above description, and ignores the rest.

**See also** `atof()`, `atoi()`, `strtod()`, `strtol()`, `strtoul()`, `stdlib.h`.

## atoll - convert string to long long integer (long long int)

**Syntax**      `#include <stdlib.h>`  
                  `long long int atoll(const char *str);`

**Description** `atoll()` converts an EBCDIC string to which *str* points into an integer of type `long long`. The string to be converted may be formatted as follows

$$[tab \dots] \left[ \begin{array}{c} + \\ - \end{array} \right] digit \dots$$

All characters that produce white space are legal for *tab* (see the definition under `isspace()`).

### Return value

Integer value of type `long long`  
                  for strings formatted as described above and representing a numeric value.

0  
                  for strings that do not conform to the syntax described above.

`LONG_MAX` or `LLONG_MIN`  
                  in the case of an overflow, depending on the sign.

**Notes**      `atoll()` also recognizes strings that begin with digits but then end with any character. `atoll()` cuts off the numeric part, converts it according to the above description, and ignores the rest.

If *zg* is a null pointer and *base* is equal to 10, then `atoll(str)` differs from `strtol(str, (char**)NULL, 10)` only in error handling.

**See also**    `atof()`, `atoi()`, `atol()`, `strtod()`, `strtol()`, `stroll()`, `strtoul()`, `stroull()`

## basename - return last element of pathname

**Syntax**      `#include <libgen.h>`  
`char *basename (char *path);`

**Description** When `basename()` is passed a pointer to a null-terminated string which contains a **pathname**, `basename()` returns a pointer to the last element of *path*. Terminating slash (/) characters are deleted.

If the passed string only consists of the '/' character, a pointer to the '/' string is returned.

If *path* or *\*path* is zero, a pointer to the '.' string is returned.

`basename()` is not reentrant.

**Return value**  
Pointer to the last component of *path*.

Example	Input string	Output pointer
	/usr/lib	lib
	/usr/	usr
	/	/

**Notes**      `basename()` works on the passed string. If necessary, the string is modified by overwriting terminating slashes (/) with '\0'.

**See also**    `dirname()`, `libgen.h`.



## bcmp - compare memory areas

**Syntax**     `#include <strings.h>`

```
int bcmp(const void *s1, const void *s2, size_t n);
```

**Description** `bcmp()` compares the first  $n$  bytes as of the memory address pointed to by  $s1$  with the memory area addressed via  $s2$ . It is assumed that both areas in the memory are at least  $n$  bytes long.

**Return value**

0               All  $n$  bytes are the same, or  $n=0$ .

$\neq 0$            The two memory areas are different.

**Note**         Portable applications should use the `memcmp()` function instead of `bcmp()`.

**See also**     `memcmp()`, `strings.h`.

## bcopy - copy memory area

**Syntax**     `#include <strings.h>`

```
void bcopy(const void *s1, const void *s2, size_t n);
```

**Description** `bcopy()` copies  $n$  bytes as of the memory address pointed to by  $s1$  into the memory area addressed via  $s2$ . Overlapping areas are corrected.

**Notes**        Portable applications should use the `memmove()` function instead of `bcopy()`.

The two function calls below are virtually equivalent. (Caution: the sequence of the arguments  $s1$  and  $s2$  is different!):

$$\text{bcopy}(s1, s2, n) \cong \text{memmove}(s2, s1, n)$$

**See also**     `memmove()`, `strings.h`.

## brk, sbrk - modify size of data segment

Syntax `#include <unistd.h>`

```
int brk(void *addr);
void *sbrk(int incr);
```

Description `brk()` and `sbrk()` are used for dynamic modification of the storage space allocated to the data segment of the calling process (cf. `exec`). The modification is made by resetting the space limit, or 'break value', of the process and allocating a corresponding area. The break value is the first unoccupied address above the data segment. The extent of the allocated storage space increases as the break value is increased. Newly allocated storage space is set to zero, but if the same storage space is reallocated to the same process, its contents are undefined.

`brk()` sets the break value to *addr* and modifies the allocated space accordingly.

`sbrk()` adds *incr* bytes to the break value and modifies the allocated space accordingly. *incr* can be negative. In this case, the extent of the assigned storage space is reduced. The current break value is returned by `sbrk(0)`.

If an application also uses additional functions for storage space management, e.g. `malloc()`, `mmap()` or `free()`, the behavior of `brk()` and `sbrk()` is undefined. Other functions may use these other memory management functions silently.

`brk()` and `sbrk()` are not reentrant.

Return value

`brk()`:

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

`sbrk()`:

previous break value

if successful.

(void\*)-1 if an error occurs. `errno` is set to indicate the error.

Errors `brk()` and `sbrk()` are unsuccessful and do not modify the allocated storage space if:

`ENOMEM` Such a modification would cause more space to be allocated than is allowed by the system-dependent maximum process size (see `ulimit()`).

**Notes**        The functions `brk()` and `sbrk()` used to be needed in special cases where no other memory management function would have offered the same possibilities. Now, however, the `mmap()` function is recommended, as it can be used simultaneously with all other memory management functions without problems.

The pointer returned by `sbrk()` is not suitable for any other use.

**See also**     `exec()`, `malloc()`, `mmap()`, `ulimit()`, `unistd.h`.

## bs2exit - program termination with MONJV (BS2000)

**Syntax**      `#include <stdlib.h>`

```
void bs2exit(int status, const char *monjv_rcode);
```

**Description** `bs2exit()` terminates the calling program. Before this is done, all files opened by the process are closed, and the following messages are output on `stderr`:

- "CCM0998 used CPU-time *t* seconds", if CPU-TIME=YES is set in the RUNTIME option.
- "CCM0999 exit *status*", if *status* ≠ EXIT\_SUCCESS (value 0).
- "CCM0999 exit FAILURE", if *status* = EXIT\_FAILURE (value 9990888).
- "EXC0732 ABNORMAL PROGRAMM TERMINATION. ERROR CODE NRT0101"

The status indicator of the monitoring job variable (1st to 3rd byte) is set to the value "\$A" in accordance with the *status* argument just like for the `exit()` function if *status* = EXIT\_FAILURE. The monitoring job variable is set to "\$T" for all other values of *status*.

The return code of the MONJV (4th - 7th byte) can be additionally supplied with *monjv\_rcode*. The *monjv\_rcode* parameter may be specified as a pointer to 4 bytes of data (the return code) that is loaded in the MONJV when the program terminates.

The contents and evaluation of the *status* argument are the same as for `exit()`.

**Notes**      When a program is terminated with `bs2exit()`, the termination routines registered with `atexit()` are not called (see `exit()`).

In order to set and query monitoring job variables, the C program must be started with the following command

```
/START-PROG program,MONJV=monjvname
```

The contents of the job variable can then be queried, e.g. with the following command:

```
/SHOW-JV JV-NAME(monjvname)
```

Further information on job monitoring using MONJV can be found in the "Job Variables" manual.

**See also**      `exit()`, `_exit()`.

**bs2fstat - get BS2000 file names from catalog** (*BS2000*)

**Syntax**        `#include <stdlib.h>`  
                  `int bs2fstat(const char *pattern, void (*function)(const char *filename, int len));`

**Description** `bs2fstat` returns the fully-qualified file names (`:catid:$userid.filename`) of files that satisfy the selection criterion given by *pattern* along with the length of each respective file name, including the terminating null byte (`\0`).

For each file found, `bs2fstat` calls a given *function* (which must be supplied by the user) and passes to it the particular *filename* (string char `*`) and the name length *len* (integer) as current arguments.

`const char *pattern` is a string specifying the selection criterion for one or more files.

*pattern* is a fully or partially qualified file name with wildcard syntax

For compatibility reasons, further parameters may be specified to determine which files are selected, e.g.:

file and catalog attributes (FCBTYPE, SHARE, etc.)

creation and access date (CRDATE, EXDATE, etc.)

These parameters must be specified in the syntax of the ISP command FSTAT.

The pattern `"*,crdate=today"`, for example, returns the names of all files that were created or updated on today's date.

`void (*function)(const char *filename, int len)` is a user-supplied function with the parameters *filename* (file name) and *len* (name length). These parameters are supplied with current values by `bs2fstat` at each function call. The function calls are made automatically by `bs2fstat` (in a `while` loop) .

**Return value**  
                  `0`                    if successful.  
                  DMS error message code  
                                     if an error occurs.

**Notes**        The flag for DMS error message codes can be only queried from outside the user-defined function, since the function is not called if the search was unsuccessful

**See also**     `system()`, `stdio.h`.

## bs2system - execute BS2000 command *(extension)*

Syntax `#include <stdlib.h>`

```
int bs2system(const char *command);
```

Description `bs2system()` executes the BS2000 command given in the string *command*.

### Return value

- |           |  |
|-----------|--|
| 0         | if the BS2000 command was executed successfully (return value of the corresponding BS2000 command: 0).         |
| -1        | if the BS2000 command was not executed successfully (return value of the BS2000 command: error code $\neq$ 0). |
| undefined | if control is not returned to the program after execution of the BS2000 command (see Notes).                   |

### Notes

`bs2system()` passes the *command* string as input to the BS2000 command processor MCLP without any changes (see also the manual "BS2000/OSD-BC Executive Macros"). No conversion to uppercase letters occurs, so the BS2000 command to be executed must be specified in uppercase. It may consist of up to 2048 characters and must not be specified with the system slash (/).

In the case of some BS2000 commands (e.g. START-PROG, LOAD-PROG, CALL-PROCEDURE, DO, HELP-SDF), control is not returned to the calling program after they are called. Programs that permit premature terminations should therefore flush all buffers (`fflush()`) and/or close files before the `bs2system` call.

See also `system()`, `stdlib.h`.

## bsd\_signal - simplified signal handling

**Syntax**      `#include <signal.h>`  
                  `void ( *bsd_signal(int sig, void ( *func)(int))) (int);`

**Description** The `bsd_signal()` function provides a partially compatible interface for programs that were written for old-style system interfaces (see “Notes” below).

The function call `bsd_signal(sig, func)` acts as though it were implemented as follows:

```
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    act.sa_flags = SA_RESTART;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, sig);
    if (sigaction(sig, &act, &oact) == -1)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

The event handling function should be declared as follows:

```
void handler(int sig);
```

where *sig* stands for the signal number. The behavior is not defined if *func* is a function which has more than one argument or an argument of a different type.

**Return value**  
 The preceding action for *sig*  
                  if successful.  
 SIG\_ERR            if an error occurs. `errno` is set to indicate the error.

**Errors**        See `sigaction()`

**Notes**        This function is a direct substitute for the BSD function `signal()` for simple applications for which a signal handling function with an argument is installed. If a BSD signal handling function which expects more than one argument is installed, the application must be modified such that it uses `sigaction()`. The `bsd_signal()` function differs from `signal()` in that the `SA_RESTART` flag is set and `SA_RESETHAND` is deleted if `bsd_signal()` is used. The status of these flags is not specified for `signal()`.

**See also**      `sigaction()`, `sigaddset()`, `sigemptyset()`, `signal()`, `signal.h`.

## bsearch - conduct binary search of sorted array

**Syntax**      `#include <stdlib.h>`  
`void *bsearch(const void *key, const void *base, size_t nel,`  
                  `size_t width, int (*compar) (const void *, const void *));`

**Description** `bsearch()` is a binary search function. It searches *nel* elements of an array *base* for the value in the data item *key*. The size of each element in the array is specified by *width*.

*compar()* is a user-supplied comparison function which is called by `bsearch()` with two arguments, a pointer to *key* and a pointer to an array element

*compar()* must return an integer less than, equal to, or greater than 0, depending on whether the first argument is less than, equal to or greater than the second argument. The array must consist of the following objects in the order given: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the *key* object, in that order.

### Return value

Pointer to the sought element  
if successful. If more than one instance of the element is found, there is no indication as to which element the pointer refers to.

Null pointer    if no element has been found.

**Notes**        The pointers to the *key* and the element at the base of the array should be of type *pointer-to-element*.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

In practice, the elements of the array are usually sorted according to the comparison function.

If the number of elements in the array is less than the size reserved for the array, *nel* should be the lower number.

#### *BS2000*

If, for example, the `qsort()` function is used for sorting the array, it makes sense to use the same comparison function *compar()* that is used by `bsearch()`. The current arguments of `qsort()` are then pointers to two array elements to be compared. □

**See also**    `hsearch()`, `lsearch()`, `qsort()`, `tsearch()`, `stdlib.h`.



## btowc - (one byte) convert multi-byte character to wide character

**Syntax**        `#include <stdio.h>`  
                 `#include <wchar.h>`  
  
                 `wint_t btowc(int c);`

**Description** `btowc()` converts the multi-byte character `c` that consists of one byte and that must be in the “initial shift” state to a wide character.

**Return value**  
wide character if successful.  
  
`WEOF`            if `c` contains the value `EOF` or `(unsigned char)c` does not represent a (1 byte) multi-byte character in the “initial shift” state.

**Notes**            In this version of the C runtime system only 1 byte characters are supported as multi-byte characters.  
                     The shift state of the multi-byte character is ignored.

**See also**        `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

## bzero - initialize memory with X'00'

**Syntax**     `#include <strings.h>`  
              `void bzero(void *s, size_t n);`

**Description** `bzero()` overwrites  $n$  bytes as of the address pointed to by  $s$  with X'00'.

**Note**        Portable applications should use the `memset()` function instead of `bzero()`.

**See also**    `memset()`, `strings.h`.

## cabs - calculate absolute value of complex number (BS2000)

**Syntax**     `#include <math.h>`  
              `double cabs(_ _complex z);`

**Description** `cabs()` calculates the absolute value of a complex number  $z$ .  
`struct (_ _complex z)` is a complex number  $z$  with real part  $x$  and imaginary part  $y$ .  
`_ _complex` is a type predefined in the header `math.h`:  
`#typedef struct{double x, y;} _ _complex`

**Return value**  
Absolute value of the complex number  $z$  if successful.  
If an overflow occurs, the program aborts (signal SIGFPE).

**See also**    `abs()`, `fabs()`, `labs()`, `sqrt()`, `math.h`.

## calloc - allocate memory

**Syntax**        `#include <stdlib.h>`  
`void *calloc(size_t nelem, size_t elsize);`

**Description** `calloc()` allocates unused memory space at execution time for an array of *nelem* elements, where the size of each element in bytes is *elsize*. `calloc()` initializes each element of the new array with binary zeros.

`calloc()` is part of a C-specific memory management package that internally handles the requested and released memory areas. As far as possible, new requests are satisfied first from areas already being managed, and only then from the operating system.

*nelem* is an integer value that specifies the number of array elements.

*elsize* is an integer value that specifies the size of an array element.

If memory areas were assigned via successive calls of `calloc()`, the arrangement of these areas in the memory is undefined. The pointer that is returned if allocation is successful is aligned with a doubleword boundary, so that it can be assigned to a pointer to any type of object. After the assignment, the object or an array of such objects in the newly assigned memory area can be accessed (until the area is explicitly released or reassigned).

### Return value

Pointer to the new memory space  
if *nelem* and *elsize* are not 0 and sufficient memory is available.

Null pointer     if the available memory does not suffice for the request.  
`errno` is set to indicate the error.

### Errors

`calloc()` will fail if:

`ENOMEM`        Insufficient memory is available.

### Notes

The new data area begins on a doubleword boundary.

To ensure that the correct size for an array element is requested, the `sizeof` operator should be used when calculating *elsize*.

If the length of the allocated memory area is exceeded during writing, serious errors in the working memory may occur.

`calloc()` is interrupt-protected, i.e. the function can be used in signal handling and contingency routines.

**See also**        `free()`, `malloc()`, `realloc()`, `stdlib.h`.

## catclose - close message catalog

**Syntax**      `#include <nl_types.h>`  
                `int catclose(nl_catd catd);`

**Description** `catclose()` closes the message catalog identified by the message catalog descriptor *catd*. If a file descriptor is used to define the type `nl_catd`, this file descriptor is also closed.

**Return value**

0	if successful.
-1	if unsuccessful. <code>errno</code> is set to indicate the error.

**Errors**      `catclose()` will fail if:

EBADF	The catalog descriptor is not valid.
EINTR	<code>catclose()</code> is interrupted by a signal.

**See also**      `catgets()`, `catopen()`, `nl_types.h`, section "Locale" on page 53.

## catgets - read message

**Syntax**      `#include <nl_types.h>`

```
char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

**Description** `catgets()` attempts to read message *msg\_id*, in set *set\_id*, from the message catalog identified by *catd*.

*catd* is a message catalog descriptor returned from an earlier call to `catopen()`.

The *s* argument points to a default message string which will be returned by `catgets()` if it cannot retrieve the identified message

**Return value**

Pointer to an internal buffer area containing the message which ends in X'00'  
if successful.

*s* if unsuccessful. `errno` is set to indicate the error.

**Errors**      `catgets()` will fail if:

EBADF          The message catalog descriptor is not valid for reading.

EINTR          The read operation is interrupted by a signal, and no data is transferred.

**See also**      `catopen()`, `nl_types.h`, section "Locale" on page 53.

## catopen - open message catalog

**Syntax**      `#include <nl_types.h>`  
                  `nl_catd catopen(const char *name, int oflag);`

**Description** `catopen()` opens a message catalog and returns a message catalog descriptor.

The *name* argument specifies the name of the message catalog to be opened. If *name* contains a "/", then *name* is interpreted as an absolute pathname for the message catalog. Otherwise, the environment variable NLSPATH is evaluated and used with *name* substituted for %N (see also section "Locale" on page 53).

If the environment variable NLSPATH does not exist or if a message catalog cannot be found in any of the path components specified by NLSPATH, then the default path is used (see `nl_types.h`).

If the value of *oflag* is NL\_CAT\_LOCALE, the default is determined by the category LC\_MESSAGES.

If *oflag* is 0, only the LANG environment variable is evaluated, regardless of the contents of the LC\_MESSAGES category (see also section "Environment variables" on page 71).

A message catalog descriptor remains valid in a process until it is closed by that process or by a successful call to one of the `exec` functions. A change in the LC\_MESSAGES category can make existing open catalogs invalid.

If a file descriptor is used to define message catalog descriptors, the FD\_CLOEXEC bit is set (see also `fcntl.h`).

### Return value

Message catalog descriptor  
                  if successful. This message catalog descriptor can then be used on subsequent calls to `catgets()` and `catclose()`.

(`nl_catd`) -1 if unsuccessful. `errno` is set to indicate the error.

### Errors

`catopen()` will fail if:

EACCES          Search permission is denied for a component of the path prefix of the message catalog  
                  or read permission is denied for the message catalog.

EMFILE          The process uses more than {OPEN\_MAX} file descriptors at one time.

ENAMETOOLONG    The length of the pathname of the message catalog exceeds {PATH\_MAX}, or a pathname component is longer than {NAME\_MAX}, or the resolution of a symbolic link produces an interim result that is longer than {PATH\_MAX}.

ENFILE	Too many files are currently open in the system.
ENOENT	The message catalog does not exist, or the name argument points to an empty string
ENOMEM	Insufficient storage space is available.
ENOTDIR	A component of the path prefix of the message catalog is not a directory.

**Notes** `catopen()` uses `malloc()` to allocate space for internal buffer areas. The `catopen()` function may fail if there is insufficient storage space available to accommodate these buffers.

Portable applications must assume that message catalog descriptors are not valid after a call to one of the `exec` functions.

Every application must store the associated message catalog in one of the default directories defined by `DEF_NLSPATH` in a format that allows it to be found on substitution of *name* for `%N` (see also `nl_types.h`).

**See also** `catclose()`, `catgets()`, `fcntl.h`, `nl_types.h`, section "Locale" on page 53 and section "Environment variables" on page 71.

## cbrt - cube root

**Syntax** `#include <math.h>`  
`double cbrt (double x);`

**Description** `cbrt()` returns the cube root of *x*.

**Return value**  
Cube root of *x*  
if successful.

**See also** `math.h`.

## cdisco - disconnect contingency routine *(BS2000)*

Syntax `#include <cont.h>`

```
void cdisco(struct enacop *enacopar);
```

Description `cdisco()` disconnects a contingency routine (TU or P1) defined with `cenaco()`. Detailed information on contingency routines can be found in the section "Contingency and STXIT routines" on page 117 and in the manual "BS2000/OSD-BC Executive Macros".

The structure `enacop` is defined in `cont.h` as follows:

```
struct enacop
{
    char  resrv1 [7];           /* reserved for int. use */
    char  coname [54];         /* name of cont. routine */
    char  resrv2 [15];        /* reserved for int. use */
    char  level;              /* priority of cont.rout. */
    int   (*econt)();         /* start adr of cont.rout. */
    int   comess;             /* contingency message */
    char  coidret [4];        /* contingency identifier */
    errcod secind;           /* secondary indicator */
    char  resrv3 [2];        /* reserved for int. use */
    errcod rcode1;          /* return code */
};
```

```
#define errcod      char
#define _norm      0        /* normterm */
#define _abnorm    4        /* abnormend */
#define _enabled   4        /* codefenabled */
#define _preven    12       /* coprevenabled */
#define _parerr    16       /* coparerror */
#define _maxexc    24       /* comaxexceed */
```

`cdisco()` evaluates only the `coidret` structure component (identifier of the contingency process).

Structure components supplied by `cdisco()`:

<code>secind</code>	"Secondary Indicator", as stored in the most significant byte of register 15 (values 4 or 20) after execution of the ENACO macro.
<code>rcode1</code>	"Return Code", as stored in the least significant byte of register 15 (values 0 or 4) after execution of the ENACO macro.

See also `cenaco()`.



## ceil, ceilf, ceill - round up floating-point number

**Syntax**      `#include <math.h>`

```
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

**Description** `ceil()`, `ceilf()` and `ceill()` round up the floating-point number *x*.

**Return value**

Smallest integer of type `double`, `float` or `long double` (greater than or equal to *x*) if successful.

`HUGE_VAL`      if an overflow occurs.  
`errno` is set to indicate the error.

**Errors**      `ceil()` will fail if:

`ERANGE`      Overflow; the return value is too high.

**Note**      The integer value returned by `ceil()`, `ceilf()` and `ceill()` as `double`, `float` or `long double` cannot always be represented as `int` or `long int`. The result should always be checked before it is assigned to a variable of type `int`, so that an integer overflow can be caught.

To make sure that any errors are caught, `errno` should be set to 0 before `ceil()`, `ceilf()` or `ceill()` is called. If after the execution `errno`  $\neq$  0, an error has occurred.

The result of `ceil()`, `ceilf()` and `ceill()` can only overflow if the following applies for the representation of the floating-point numbers: `DBL_MANT_DIG` > `DBL_MAX_EXP`.

**See also**      `abs()`, `fabs()`, `floor()`, `floorf`, `floorl()`, `()isnan()`, `math.h`.

## cenaco - define contingency routine (BS2000)

Syntax `#include <cont.h>`

```
void cenaco(struct enacop *enacopar);
```

Description `cenaco()` defines a contingency routine (TU or P1) and can thus be used to assign a routine written by the user as a contingency routine. For more detailed information on contingency routines, refer to section "Contingency and STXIT routines" on page 117 and the manual "BS2000/OSD-BC Executive Macros"

The structure `enacop` is defined in `cont.h` as follows:

```
struct enacop
{
    char resrv1 [7];           /* reserved for int. use */
    char coname [54];         /* name of cont. routine */
    char resrv2 [15];        /* reserved for int. use */
    char level;              /* priority of cont.rout. */
    int (*econt)();          /* start adr of cont.rout. */
    int comess;              /* contingency message */
    char coidret [4];        /* contingency identifier */
    errcod secind;           /* secondary indicator */
    char resrv3 [2];         /* reserved for int. use */
    errcod rcode1;           /* return code */
};

#define errcod      char
#define _norm      0        /* normterm */
#define _abnorm    4        /* abnormend */
#define _enabled   4        /* codefenabled */
#define _preven    12       /* coprevenabled */
#define _parerr    16       /* coparerror */
#define _maxexc    24       /* comaxexceed */
```

Some entries for the structure components can or must be supplied by the user before the call to `cenaco()`; others are used by `cenaco()` to store information during the run.

Entries supplied by the user:

<code>coname</code>	Name of the contingency process. The name can have a maximum length of 54 bytes (without the null byte), must be in uppercase, and must be end with at least one blank (a null byte immediately after the actual name is not recognized as an end criterion by the system). The <code>strfill()</code> function, for example, is suitable for supplying <code>coname</code> . This input is mandatory.
<code>level</code>	Priority level of the contingency process. This input is mandatory. Values from 1 - 126 are legal.

<code>econt</code>	Start address of the contingency routine. This input is mandatory.
<code>comess</code>	Contingency message. This input is optional. The value is passed to the contingency routine as a parameter.

**Entries supplied by `cenaco()`:**

<code>coidret</code>	Short ID of the contingency process. This short ID must be used in further macros (e.g. <code>SOLSIG</code> ) for the identification of the contingency process.
<code>secind</code>	"Secondary Indicator", as stored in the most significant byte of register 15 (values 4 or 20) after execution of the <code>ENACO</code> macro.
<code>rcode1</code>	"Return Code", as stored in the least significant byte of register 15 (value 0 or 4) after execution of the <code>ENACO</code> macro.

**Notes** A maximum of 255 contingency routines can be defined.

**See also** `cdisco()`, `cstxit()`, `signal()`, `alarm()`, `raise()`, `sleep()`.

## cfgetispeed - get input baud rate

**Syntax**      `#include <termios.h>`

```
speed_t cfgetispeed(const struct termios *termios_p);
```

**Description** `cfgetispeed()` extracts the input baud rate from the `termios` structure to which the `termios_p` argument points. It returns exactly the value in the `termios` data structure.

*Extension*

Since different baud rates are not supported by the hardware, it is only relevant whether this value is zero or non-zero. See `tcsetattr()` for details. □

**Return value**

Input baud rate of type `speed_t`  
if successful.

**See also** `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`, `tcgetattr()`, `termios.h`, section "General terminal interface" on page 94.

## cfgetospeed - get output baud rate

**Syntax**      `#include <termios.h>`

```
speed_t cfgetospeed(const struct termios *termios_p);
```

**Description** `cfgetospeed()` extracts the output baud rate from the `termios` structure to which the `termios_p` argument points. It returns exactly the value in the `termios` data structure.

**Return value**

Output baud rate of type `speed_t`  
if successful.

*Extension*

Since different baud rates are not supported by the hardware, it is only relevant whether this value is zero or non-zero. See `tcsetattr()` for details. □

**See also** `cfgetispeed()`, `cfsetispeed()`, `cfsetospeed()`, `tcgetattr()`, `termios.h`, section "General terminal interface" on page 94.

## cfsetispeed - set input baud rate

**Syntax**      `#include <termios.h>`

```
int cfsetispeed(struct termios *termios_p, speed_t speed);
```

**Description** `cfsetispeed()` sets the input baud rate in the `termios` structure pointed to by `termios_p` to the value of `speed`.

`cfsetispeed()` has no effect on the baud rates set in the hardware until a subsequent successful call to `tcsetattr()` on the same `termios` structure.

### *Extension*

Only the corresponding value in the `termios` structure is changed. Since different baud rates are not supported by the hardware, it is only relevant whether or not this value is equal to zero. All baud rates defined in `termios.h` can, however, be specified and stored in the `termios` structure. If baud rates which are not defined in `termios.h` are specified, they are not stored: -1 is returned, and `errno` is set to the value `EINVAL`. See `tcsetattr()` for more details.

If the input baud rate is set to zero, it is assigned the value of the output baud rate. Attempts to set unsupported baud rates are ignored. This applies to changes to baud rates not supported by the hardware and to the setting of different input and output baud rates (if this is not supported by the hardware). □

**Return value**

0                    if successful.

-1                   if an error occurs. `errno` is set to indicate the error.

**Errors**            `cfsetispeed()` will fail if:

`EINVAL`            `speed` is not a valid baud rate (e.g. 9999) or the value of `speed` is not within the permitted range of values defined in `termios.h`.

**See also**          `cfgetispeed()`, `cfgetospeed()`, `cfsetospeed()`, `tcsetattr()`, `termios.h`, section "General terminal interface" on page 94.

## cfsetospeed - set output baud rate

**Syntax**      `#include <termios.h>`  
`int cfsetospeed (struct termios *termios_p, speed_t speed);`

**Description** `cfsetospeed()` sets the *output* baud rate stored in the `termios` structure pointed to by *termios\_p* to *speed*.

`cfgetospeed()` has no effect on the baud rates set in the hardware until a subsequent successful call to `tcsetattr()` on the same `termios` structure.

### *Extension*

Only the corresponding value in the `termios` structure is changed. Since different baud rates are not supported by the hardware, it is only relevant whether or not this value is equal to zero. All baud rates defined in `termios.h` can, however, be specified and stored in the `termios` structure. If baud rates which are not defined in `termios.h` are specified, they are not stored: -1 is returned, and `errno` is set to the value `EINVAL`. See `tcsetattr()` for more details. The zero baud rate, `B0`, is used to terminate the connection. If `B0` is specified, the modem control lines are no longer be asserted. Normally, this disconnects the line. □

**Return value**  
0                    if successful.  
-1                   if an error occurs. `errno` is set to indicate the error.

**Errors**      `cfsetospeed()` will fail if:  
`EINVAL`      *speed* is not a valid baud rate or the value of *speed* is not within the permitted range of values defined in `termios.h`.

**See also**      `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `tcsetattr()`, `termios.h`, section "General terminal interface" on page 94.

## chdir - change working directory

Syntax `#include <unistd.h>`

```
int chdir(const char *path);
```

Description `chdir()` causes the directory pointed to by the *path* argument to become the current working directory, i.e. the starting point for path searches for pathnames not beginning with a *.*

*path* points to the pathname of a directory.

Return value

0 if successful. The specified directory becomes the current working directory.

-1 if an error occurs. `errno` is set to indicate the error.

Errors

`chdir()` will fail if:

EACCES Search permission is denied for any component of the pathname.

*Extension*

EFAULT *path* is an invalid address.

EINTR A signal was caught during the execution of the `chdir()` system call.

EIO An I/O error occurred while reading from or writing to the file system.

ELOOP Too many symbolic links were encountered in resolving *path*. □

ENAMETOOLONG

The length of *path* exceeds `{PATH_MAX}` or a component of *path* is longer than `{NAME_MAX}` and `{POSIX_NO_TRUNC}` is set.

ENOENT A component of *path* does not exist or is a null pathname.

ENOTDIR A component of the pathname is not a directory.

Notes

The effect of changing the working directory applies only for the duration of the current program (or current shell). If a new program or shell is started, the home directory is reset as the current working directory.

In order to make a directory the current working directory, a process must have execute (search) permission for that directory.

`chdir()` will work only in the currently active process and only until termination of the active program. `chdir()` is executed only for POSIX files.

See also `chroot()`, `getcwd()`, `unistd.h`.

## chmod - change mode of file

Syntax `#include <sys/stat.h>`

*Optional*

`#include <sys/types.h>` □

`int chmod(const char *path, mode_t mode);`

**Description** `chmod()` changes `S_ISUID`, `S_ISGID` and the file permission bits of the file pointed to by the *path* argument to the corresponding bits in the *mode* argument. The effective user ID of the process must match the owner of the file or have appropriate privileges for this purpose.

`S_ISUID`, `S_ISGID` and the file permission bits are described in `sys/stat.h`.

If the calling process does not have appropriate privileges, and if the group ID of the file does not match the effective group ID or one of the supplementary group IDs, and if the file is a regular file, the `S_ISGID` (set-group-ID on execution) bit in the file's mode will be cleared upon successful return from `chmod()`.

In the C runtime system, `chmod()` is also executed for open files. Other X/Open-compatible systems can define other specifications for this case.

Upon successful completion, `chmod()` will mark for update the `st_ctime` field of the file.

**Return value**

0	if successful. The access permissions of the specified file are set as defined.
-1	if an error occurs. The file mode is not changed, and <code>errno</code> is set to indicate the error.

**Errors**

`chmod` will fail if:

`EACCES` Search permission is denied on a component of the path prefix.

*Extension*

`EFAULT` *path* points outside the allocated address space of the process.

`EINTR` A signal was caught during execution of the system call. □

`EINVAL` The value of *mode* is invalid.

An attempt was made to access a BS2000 file.



*Extension*

EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in resolving <i>path</i> . □
ENAMETOOLONG	The length of the <i>path</i> argument exceeds <code>{PATH_MAX}</code> or a pathname component is longer than <code>{NAME_MAX}</code> .
ENOENT	<i>path</i> points to the name of a file that does not exist or to an empty string.
ENOTDIR	A component of <i>path</i> is not a directory.
EPERM	The effective user ID does not match the owner of the file and the process does not have appropriate privileges.
EROFS	The named file resides on a read-only file system.

**Notes** `chmod()` is executed only for POSIX files

**See also** `chown()`, `fchmod()`, `mkdir()`, `mkfifo()`, `open()`, `stat()`, `sys/types.h`, `sys/stat.h`.

## chown - change owner and group of file

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h>` □

`int chown(const char *path, uid_t owner, gid_t group);`

Description *path* points to a pathname naming a file. The user ID and group ID of the named file are set to the numeric values contained in *owner* and *group*, respectively.

Only processes with an effective user ID equal to the user ID of the file or processes with appropriate privileges may change the user ID and group ID of a file. The following applies if `{_POSIX_CHOWN_RESTRICTED}` has been activated for *path*:

- Changing the user ID is restricted to processes with appropriate privileges.
- Changing the group ID is permitted to a process with an effective user ID equal to the user ID of the file, but without appropriate privileges, if and only if *owner* is equal to the user ID of the file and *group* is equal to either the effective group ID of the process or to one of its supplementary group IDs.

If *path* refers to a regular file, the set-user-ID (`S_ISUID`) and set-group-ID (`S_ISGID`) bits of the file mode are cleared upon successful return from `chown()`, unless the call is made by a process with appropriate privileges, in which case these bits are not altered under POSIX. If `chown()` is successfully invoked on a file that is not a regular file, these bits may be cleared. These bits are defined in `sys/stat.h`.

If *owner* or *group* is specified as `(uid_t)-1` or `(gid_t)-1`, respectively, the corresponding ID of the file is unchanged.

Upon successful completion, `chown()` will mark for update the `st_ctime` field of the file.

Return value

- 0 if successful.
- 1 if an error occurs; `errno` is set to indicate the error.

Errors

`chown()` will fail if:

`EACCES` Search permission is denied on a component of *path*.

*Extension*

`EFAULT` An invalid address was passed as an argument.

`EINTR` A signal was caught during the `chown` call. □

EINVAL	The value of the specified user ID or group ID is not supported (e.g. if the value is less than 0) or an attempt was made to access a BS2000 file.
<i>Extension</i>	
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in resolving <i>path</i> . □
ENAMETOOLONG	The length of the <i>path</i> argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
ENOENT	<i>path</i> points to the name of a file that does not exist or to an empty string.
ENOTDIR	A component of <i>path</i> is not a directory.
EPERM	The effective user ID does not match the owner of the file or the calling process does not have the appropriate privileges, although {_POSIX_CHOWN_RESTRICTED} indicates that such privileges are required.
EROFS	The named file resides on a read-only file system.

**Notes** chown() is executed only for POSIX files

**See also** chmod(), sys/stat.h, sys/types.h, unistd.h.

## chroot - change root directory

Syntax `#include <unistd.h>`  
`int chroot(const char *path);`

Description *path* points to a pathname naming a directory. The `chroot()` function causes the named directory to become the root directory, i.e. the starting point for path searches for pathnames beginning with `/`. The working directory of the user is not affected by `chroot()`.

The process must have appropriate privileges to change the root directory.

The `..` (dot-dot) entry in the root directory is interpreted to mean the root directory itself. Thus, `..` cannot be used to access files outside the subtree rooted at the root directory.

`chroot()` is not reentrant.

### Return value

0 if successful  
-1 if an error occurs. `errno` is set to indicate the error.

### Errors

`chroot()` will fail if:

`EACCES` Search permission is denied for a component of *path*.

#### *Extension*

`EFAULT` An invalid address was passed as an argument.

`EINTR` A signal was caught during the `chroot()` system call.

`ELOOP` Too many symbolic links were encountered in resolving *path*. □

#### `ENAMETOOLONG`

The length of the *path* argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}`.

`ENOENT` *path* points to the name of a directory that does not exist or to an empty string.

`ENOTDIR` A component of the pathname *path* is not a directory .

`EPERM` The effective user ID is not that of a process with appropriate privileges.

### Notes

`chroot()` is executed only for POSIX directories.

`chroot()` works only in the currently active process and remains in effect only until the termination of that process.

See also `chdir()`, `unistd.h`.

## clearerr - clear end-of-file and error indicators

**Syntax**     `#include <stdio.h>`

```
void clearerr(FILE *stream);
```

**Description** `clearerr()` clears the end-of-file and error indicators for the stream to which *stream* points.

*BS2000*

`clearerr()` is implemented both as a macro and as a function.

`clearerr()` can also be used on files with record I/O. □

**Notes**       The program environment determines whether `clearerr()` is executed for a BS2000 or POSIX file.

**See also**    `feof()`, `ferror()`, `stdio.h`.

## clock - report CPU time used by a process

**Syntax**      `#include <time.h>`  
`clock_t clock(void);`

**Description** The behavior of `clock()` is determined by the selected functionality (see section "Scope of the supported C library" on page 18) as described below:

When called with POSIX functionality, `clock()` returns the elapsed CPU time since the first `clock` call.

*BS2000*

When called with BS2000 functionality, `clock()` returns the CPU time since the start of the program. □

**Return value**

Amount of CPU time elapsed since the first call to `clock()`  
if successful. The return value for the first call to `clock()` is 0.

*BS2000*

CPU time since the start of the program  
if successful. □

`(clock_t)-1` if the CPU time is not available or cannot be represented. This applies to both POSIX and BS2000 functionality.

**Notes** The value returned by `clock()` is defined in ten thousandths of a second for compatibility across systems that have CPU clocks with high resolutions. Consequently, the value returned by `clock()` may wrap around to 0 on some systems. For example, on a machine with 32-bit values for `clock_t`, it will wrap after 2147 seconds or 36 minutes.

If the CPU time is to be specified in seconds, the return value of `clock()` must be divided by the value of the macro `CLOCKS_PER_SEC` (see `time.h`).

**See also** `asctime()`, `cputime()`, `ctime()`, `difftime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `strptime()`, `system()`, `time()`, `times()`, `utime()`, `wait()`, `time.h`, section "Scope of the supported C library" on page 18.

## close - close file

Syntax `#include <unistd.h>`

```
int close(int fildev);
```

Description *fildev* is a file descriptor obtained from a `creat()`, `open()`, `dup()`, `fcntl` or `pipe()` system call. `close()` closes the file descriptor indicated by *fildev*. All existing record locks owned by the process on the file specified with *fildev* are removed.

If `close()` is interrupted by a signal that is to be caught, it will return -1 with *errno* set to `EINTR`, and the state of *fildev* is unspecified.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO will be discarded.

When all file descriptors associated with a file description have been closed, the file description will be freed.

If the link count of the file is 0, when all file descriptors associated with the file are closed, the space occupied by the file will be freed and the file will no longer be accessible.

### Extension

If a stream-based file is closed and the calling process was previously registered to receive a `SIGPOLL` signal (see `signal()` and `sigset()`) for events associated with that stream, the calling process will be unregistered for events associated with the stream. The last `close()` for a stream causes the stream associated with *fildev* to be dismantled. If `O_NDELAY` and `O_NONBLOCK` are not set and there have been no signals posted for the stream, `close()` waits for up to 15 seconds for each module and driver for any queued output to drain before dismantling the stream. If `O_NDELAY` or `O_NONBLOCK` is set, or if there are any pending signals, `close()` does not wait for the output to drain and dismantles the stream immediately. □

### Return value

0 if successful. The specified file is closed.  
-1 if an error occurs; *errno* is set to indicate the error.

### Errors

`close()` will fail if:

`EBADF` *fildev* is not a valid file descriptor. or  
the BS2000 file is not accessible in the process.

`EINTR` `close()` was interrupted by a signal.

- Notes** If the file was opened with `fopen()`, it must be closed with `fclose()` instead of `close()`.  
When a program is terminated (normally or with `exit()`), all open files are automatically closed.  
The program environment determines whether `close()` is executed for a BS2000 or POSIX file.
- See also** `creat()`, `dup()`, `fcntl()`, `lseek()`, `open()`, `read()`, `tell()`, `write()`, `unistd.h`.

## closedir - close directory

**Syntax** `#include <dirent.h>`  
*Optional*  
`#include <sys/types.h> □`  
`int closedir(DIR *dirp);`

**Description** `closedir()` closes the directory stream referred to by the argument *dirp*. Upon return, the value of *dirp* will no longer point to an accessible object of the type `DIR`. The file descriptor used in the `DIR` structure is closed.

**Return value**  
0 if successful.  
-1 if an error occurs; `errno` is set to indicate the error.

**Errors** `closedir()` will fail if:  
`EBADF` The *dirp* argument does not refer to an open directory stream.  
`EINTR` `closedir()` was interrupted by a signal.

**Notes** `closedir()` is executed only for POSIX files.

**See also** `opendir()`, `dirent.h`, `sys/types.h`.



## closelog, openlog, setlogmask, syslog - control system log

**Syntax**

```
#include <syslog.h>

void closelog(void)

void openlog(const char *ident, int logopt, int facility);

int setlogmask(int maskpri);

void syslog(int priority, const char *message, ... / * arguments */);
```

**Description** `syslog()` writes a message *message* to the `/var/adm/messages` file. The message consists of a message header and the message text. The message header contains the specification of the message weight, the time specification, the `syslog` ID and, optionally, the process ID.

The message text is created from the *message* argument and the subsequent arguments as though these arguments had been passed to `printf()`, except that `%m` in the format string to which *message* points is replaced by the error message that corresponds to the current value of `errno`. A trailing newline character is added if necessary.

Values for *priority* are formed by inclusive OR from the message weight and, if applicable, the function value. If no function was specified for *facility*, the predefined default function is used.

Possible values for the message weight are:

LOG_EMERG	A "panic" condition. This condition is normally passed to all users.
LOG_ALERT	A condition that should be corrected immediately, e.g. a damaged system database.
LOG_CRIT	Critical condition, e.g. device error.
LOG_ERR	Error.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions which do not involve error conditions but may require special steps.
LOG_INFO	Information messages.
LOG_DEBUG	Messages containing information which are normally only used during program debugging.

*facility* indicates which application or which system component has generated the message. Possible values are:

LOG_USER	Messages created by optional user processes. This is the default function ID if no other one is specified.
LOG_LOCAL0	Reserved for local use.
LOG_LOCAL1	Reserved for local use.
LOG_LOCAL2	Reserved for local use.

LOG_LOCAL3	Reserved for local use.
LOG_LOCAL4	Reserved for local use.
LOG_LOCAL5	Reserved for local use.
LOG_LOCAL6	Reserved for local use.
LOG_LOCAL7	Reserved for local use.

`openlog()` sets process attributes which control subsequent calls of `syslog()`. The *ident* argument is a string that is prefixed to every message. *logopt* is a bit field in which log options are displayed. Values for *logopt* are generated from any number of the following values via bit-wise inclusive OR. The following are current values for *logopt*:

LOG_PID	Logs the process ID with each message. This is useful for the identification of special processes.
LOG_CONS	Outputs messages on the system console if they cannot be written to the <code>/var/adm/messages</code> file. This option can be used safely in processes which do not have a control terminal, because <code>syslog()</code> generates a child process before the console is opened.
LOG_NDELAY	Opens the <code>/var/adm/messages</code> file on execution of <code>openlog()</code> . Normally the opening is delayed until the first message is logged. This option is useful for programs which have to pay attention to the sequence when allocating file descriptors.
LOG_ODELAY	Delays the opening until <code>syslog()</code> is called.
LOG_NOWAIT	Does not wait for child processes which were split up for the logging of messages on the console. This option should be used by processes which, with the help of SIGCHLD, output notification of the termination of a child process, otherwise <code>syslog()</code> might be blocked by the wait for a child process whose end status has already been reached.

The *facility* argument encodes a standard function which is to be assigned to all messages that do not have an already encoded explicit function. The default for the standard function is LOG\_USER.

The `openlog()` and `syslog()` functions can assign file descriptors. It is not necessary to call `openlog()` before `syslog()`.

The `closelog()` function closes all open file descriptors that were assigned by previous calls of `openlog()` and `syslog()`.

`setlogmask()` sets the log priority mask for the current process to *maskpri* and returns the previous mask. If *maskpri* has the value 0, the current log priority mask is not changed. `syslog()` calls by the current process whose priority is not specified in *maskpri* are rejected. The mask for a specific priority *pri* is computed from the `LOG_MASK(pri)` macro. The masks for all priorities up to and including *toppri* are specified in the `LOG_UPTO(toppri)` macro. By default all priorities can be logged.

Symbolic constants which are used as values for *logopt*, *facility*, *priority* and *maskpri* are defined in the `syslog.h` header file.

**Return value**`setlogmask():`

Previous log priority mask.

**See also** `printf()`, `syslog.h`.

## **compile - produce compiled regular expression**

**Syntax**`#include <regex.h>``int compile(char *instring, char *exbuf, const char *endbuf, int eof);`**Description** See `regex()`.**Notes** This function will not be supported by the X/Open standard in the future.

## confstr - get string value of system variable

**Syntax**      `#include <unistd.h>`  
`size_t confstr(int name, char *buf, size_t len);`

**Description** `confstr()` provides a method of obtaining the current string values of a configurable system variable. Its use and purpose are similar to `sysconf()`, but it is used where string values rather than numeric values are returned.

The implementation supports only value of `_CS_PATH`, defined in `unistd.h`, for *name*.

If *len* is not 0, and if *name* has a configuration-defined value, `confstr()` copies that value into the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes, including the terminating null byte, then `confstr()` truncates the string to *len*-1 bytes and null-terminates the result. The application can detect that the string was truncated by comparing the value returned by `confstr()` with *len*.

If *len* is 0 and *buf* is a null pointer, then `confstr()` still returns the integer value as defined below, but does not return a string. If *len* is 0 but *buf* is not a null pointer, the result is unspecified.

### Return value

Buffer size for the value of *name*  
    if *name* has a configuration-defined value. If this return value is longer than *len*, the string returned in *buf* is truncated.

0  
    if *name* does not have a configuration-defined value. `errno` is not set.  
    if *name* has an invalid value. `errno` is set to indicate the error.

### Errors

`confstr()` will fail if:  
`EINVAL`      The value of the *name* argument is invalid.

### Notes

An application can distinguish between an invalid *name* parameter value and one that corresponds to a configurable environment variable that has no configuration-defined value by checking if `errno` is modified. This mirrors the behavior of `sysconf()`.

`confstr()` was originally needed as a way of finding the configuration-defined default value for the environment variable `PATH`. Since `PATH` can be extended by the user, applications need a way to determine the system-supplied `PATH` environment variable value that contains the correct search path for the XPG4 commands.

**See also**      `sysconf()`, `pathconf()`, `unistd.h`.

## cos - cosine function

**Syntax**     `#include <math.h>`  
              `double cos(double x);`

**Description** `cos()` computes the cosine of the floating-point number  $x$ , which specifies an angle in radians.

**Return value**  
`cos(x)`                   if successful. The return value is a floating-point number in the range  $[-1.0, +1.0]$ .

**See also**     `acos()`, `asin()`, `atan()`, `atan2()`, `sin()`, `tan()`, `math.h`.

## cosh - hyperbolic cosine function

**Syntax**     `#include <math.h>`  
              `double cosh(double x);`

**Description** `cosh()` computes the hyperbolic cosine of the floating-point number  $x$ .

**Return value**  
`cosh(x)`                   if successful.  
`HUGE_VAL`                 if an overflow occurs.  
                              `errno` is set to indicate the error.

**Errors**     `cosh()` will fail if:  
`ERANGE`         The value of  $x$  causes an overflow.

**See also**     `acos()`, `asin()`, `atan()`, `cos()`, `sinh()`, `tanh()`, `math.h`.

**cputime - calculate CPU time used by current task** *(BS2000)*

Syntax        `#include <stdlib.h>`  
              `int cputime(void);`

Description `cputime()` returns the CPU time used by the current task (since LOGON).

Return value  
              CPU time used in ten thousandths of a second.

See also      `clock()`, `stdlib.h`, section "Scope of the supported C library" on page 18.

## creat - create new file or overwrite existing one

**Name**        **creat, creat64**

**Syntax**      `#include <fcntl.h>`

*Optional*

`#include <sys/types.h>`

`#include <sys/stat.h>`

`int creat(const char *path, mode_t mode);`

`int creat64(const char *path, mode_t mode);`

*BS2000*

`int creat(const char *path, int mode);`

**Description** If POSIX files are created, the behavior of this function conforms to the XPG standard as described below:

`creat()` creates a new file or prepares to rewrite an existing file named by the pathname pointed to by *path*.

If the file exists, its length is truncated to 0, and the mode and owner are unchanged.

If the file does not exist, the file's owner ID is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process, unless the `S_ISGID` bit is set in the parent directory, in which case the group ID of the file is inherited from the parent directory. The access permission bits of the file mode are set to the value of *mode* modified as follows:

- If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the `S_ISGID` bit is cleared.
- All bits set in the process's file mode creation mask are cleared (see `umask()`).
- The 'save text image after execution bit' of the mode is cleared (see `chmod()`).

Upon successful completion, a write-only file descriptor is returned, and the file is opened for writing even if the mode does not permit writing. The file position indicator is set to the beginning of the file. The file descriptor is set to remain open across `exec` system calls (see `fcntl()`). A new file may be created with a mode that forbids writing.

The call `creat(path, mode)` is equivalent to:

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)
```

There is no difference in functionality between `creat()` and `creat64()` except that the identifier for a large file is stored in the file description linked to the file descriptor, i.e. the `O_LARGEFILE` bit is set. A file identifier is returned that can be used to increase the size of the file beyond 2 GB.

*BS2000*

The following must be noted when creating BS2000 files:

*path* can be:

- any valid BS2000 file name
- "link=linkname"  
*linkname* denotes a BS2000 link name.

*mode* is ignored. However, it is required for the creation of portable programs since it controls the protection bit assignment in the UNIX operating system.

The BS2000 file name or link name may be written in lowercase and uppercase letters. It is automatically converted to uppercase letters.

If the file does not exist, the following file is created by default:

for KR functionality (only available with C/C++ versions lower than V3), a SAM file with variable record length and standard block length;

for ANSI functionality, an ISAM file with variable record length and standard block length.

When using a link name, the following file attributes can be changed by means of the SET-FILE-LINK command: access method, record length, record format, block length and block format.

If an existing file is truncated to length 0, the catalog attributes of the file are preserved.

A maximum of \_NFILE files may be open simultaneously. \_NFILE is defined as 2048 in `stdio.h`.

## Return value

File descriptor

if successful.

-1

if an error occurs. `errno` is set to indicate the error. No file is opened or modified.

## Errors

`creat()` will fail if:

EACCES

Search permission is denied on a component of the path.

The file does not exist and the directory in which the file is to be created does not permit writing.

The file exists and write permission is denied.

*Extension*

EAGAIN

The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see `chmod()`).

EEXIST

`O_CREAT` and `O_EXCL` are set, and the file name already exists.



*Extension*

EFAULT *path* points outside the allocated address space of the process.

EINTR A signal was caught during the `creat()` system call.

EISDIR The specified file is a directory.

*Extension*

ELOOP Too many symbolic links were encountered in resolving *path*.

EMFILE The process has too many open files (see `getrlimit()`).

ENAMETOOLONG

The length of the *path* argument exceeds `{PATH_MAX}` or a *path* component is longer than `{NAME_MAX}`.

ENFILE The system file table is full.

ENOENT A component of the pathname does not exist or *path* points to an empty string.

ENOSPC The file system is out of inodes.

ENOTDIR A component of the pathname is not a directory.

ENXIO The named file is a character special or block special file, and the device associated with this special file does not exist.

EROFS The specified file resides or would reside on a read-only file system.

ETXTBSY The file is a pure program file that is currently being executed.

**Notes** The program environment determines whether a BS2000 or POSIX file is created.

**See also** `chmod()`, `close()`, `dup()`, `fcntl()`, `getrlimit()`, `lseek()`, `open()`, `read()`, `umask()`, `write()`, `stat()`, `fcntl.h`, `sys/stat.h`, `sys/types.h`.

## crypt - encode strings using algorithms

**Syntax**      `#include <unistd.h>`  
`char *crypt(const char *key, const char *salt);`

**Description** `crypt()` is a string encoding function. It is based on a one-way encoding algorithm with variations intended to prevent the use of hardware implementations for a key search.

*key* is the input string to be encoded, typically a user's password. *salt* is a two-character string chosen from the set of characters (a-z, A-Z, 0-9, . , /).

This string is used to vary the encoding algorithm in one of 4096 different ways, after which the input string is used as the key to repeatedly encode a constant string. The returned value points to the encoded input string.

**Return value**

Pointer to the encoded string.  
The first two characters of the returned value are those of the *salt* argument.

Null pointer      if an error occurs;  
                  `errno` is set to indicate the error.

**Notes**      The return value of `crypt()` points to static data that is overwritten at each call.

**See also**      `encrypt()`, `setkey()`, `unistd.h`.

**cstxit - define STXIT routine** (BS2000)

Syntax `#include <stxit.h>`

```
void cstxit(struct stxtp stxtpar);
```

Description `cstxit()` defines an STXIT routine and can thus be used to assign a routine written by the user as an STXIT routine.

Detailed information on the programming of STXIT routines can be found in the section "Contingency and STXIT routines" on page 117 and in the manual "BS2000/OSD-BC Executive Macros".

The structure of `stxtp` is defined in `stxit.h` as follows:

```
struct stxtp
{
    addr      bufadr;      /* Address of the message for the program (OPINT) */
    enum err_set retcode; /* Return code */
    struct cont contp;    /* Address of the STXIT routines */
    struct nest nestp;   /* Max. nesting level */
    struct stx stxp;     /* Control of the cstxit call */
    struct diag diagp;   /* Diagnostic controlling */
    struct type typep;   /* Parameter transfer mode */
};

struct cont                /* Address of the STXIT routine for */
{                          /* the particular event class */
    int (*prchk) ();
    int (*timer) ();
    int (*opint) ();
    int (*error) ();
    int (*runout) ();
    int (*brkpt) ();
    int (*abend) ();
    int (*pterm) ();
    int (*rtimer) ();
};

struct nest                /* Max. nesting level for the */
{                          /* particular event class */
    char prchk;
    char timer;
    char opint;
    char error;
    char runout;
};
```

```
    char brkpt;
    char abend;
    char pterm;
    char rtimer;
    char filler;
};
```

```
struct stx          /* Control of the cstxit call for */
{                  /* the particular event class */
    stx_set prchk;
    stx_set timer;
    stx_set opint;
    stx_set error;
    stx_set runout;
    stx_set brkpt;
    stx_set abend;
    stx_set pterm;
    stx_set rtimer;
    stx_set filler;
};
```

```
struct diag        /* Diagnostic control for */
{                  /* the particular event class */
    diag_set prchk;
    diag_set timer;
    diag_set opint;
    diag_set error;
    diag_set runout;
    diag_set brkpt;
    diag_set abend;
    diag_set pterm;
    diag_set rtimer;
    diag_set filler;
};
```

```
struct type        /* Parameter transfer mode for */
{                  /* the particular event class */
    type_set prchk;
    type_set timer;
    type_set opint;
    type_set error;
    type_set runout;
    type_set brkpt;
    type_set abend;
    type_set pterm;
    type_set rtimer;
};
```

```

    type_set filler;
};
#define stx_set      char
#define old_stx     0
#define new_stx     4
#define del_stx     8

#define diag_set    char
#define ful_diag    0
#define min_diag    4
#define no_diag     8

#define err_set     char
#define no_err      0
#define par_err     4
#define stx_err     8
#define mem_err     12

#define type_set    char
#define par_opt     0
#define par_std     4

```

### Control of the `cstxit` call:

This information is used to control the execution of the `cstxit` call. It defines which actions are to be performed for the particular event class. .

- |                      |  |
|----------------------|--|
| <code>old_stx</code> | No change is required for the corresponding event class. A previously assigned STXIT routine is retained. The remaining data for that event class is not evaluated.  |
| <code>new_stx</code> | A new STXIT routine is assigned for the corresponding event class. The remaining data for the event class is evaluated in this case. The address of the routine, in particular, must be present in the corresponding entry of <code>contp</code> . |
| <code>del_stx</code> | The STXIT routine that was assigned to this point is deleted for the corresponding event class. The remaining data for the event class is not evaluated.   |

### Diagnostic control:

- |                         |  |
|-------------------------|--|
| <code>ful_diag</code> , | Diagnostic control parameters are accepted syntactically for compatibility reasons, but are no longer evaluated since the conversion to ILCS. The routine is activated without a preceding diagnostic message. |
| <code>min_diag</code> , |  |
| <code>no_diag</code>    |  |

**Parameter transfer mode:**

- `par_opt`     The parameters are passed in registers 1-4.
- `par_std`     The parameters are passed in a parameter list. This is the only value permitted in C.

**Return code:**

- `no_err`     The STXIT routine was defined correctly.
- `par_err`     The parameter structure *stxitpar* was incorrectly supplied.
- `stx_err`     Error on activating the STXIT routine.
- `mem_err`     Error in the memory space request (when activating the STXIT routine).

**Notes**     The parameter structure *stxitpar* must be supplied by the user.

To standardize initialization, a defined prototype (`stxit_pr`) is provided in the `stxit.h` header. This prototype can be copied into any user-defined structure of type `stxitp`, so that only the fields for the event classes for which the assignment of an STXIT routine is to be changed need to be set.

For event class `INTR`, the address (*stxitpar.bufadr*) at which the information for the program is to be provided must also be supplied. The STXIT contingency routine can then fetch the message from the given address and evaluate it.

**See also**     `alarm()`, `cenaco()`, `raise()`, `signal()`, `sleep()`.

## ctermid - generate pathname for controlling terminal

**Syntax**        `#include <stdio.h>`  
`char *ctermid(char *s);`

**Description** `ctermid()` generates a string that, when used as a pathname, refers to the current controlling terminal for the current process.

If *s* is a null pointer, the string is generated in an internal static area that is overwritten by the each call to `ctermid()`, and the address of this area is returned. Otherwise, *s* is assumed to point to a character array of at least `L_ctermid` elements; the pathname is placed in this array and the value of *s* is returned. The symbolic constant `L_ctermid` is defined in the header file `stdio.h`.

**Return value**  
In non-conformance with the XPG4 standard, `/dev/tty` is always returned.

**Notes**        The difference between `ctermid()` and `ttyname()` is that `ttyname()` must be handed a file descriptor and returns the pathname of the terminal associated with that file descriptor, while `ctermid()` returns a string (such as `/dev/tty`) that will refer to the current controlling terminal if used as a pathname. In other words, `ttyname()` is only useful if the process has already opened at least one file for a terminal.

**See also**     `ttyname()`, `stdio.h`.

## ctime - convert date and time to string

**Syntax**      `#include <time.h>`

```
char *ctime(const time_t *clock);
```

**Description** `ctime()` converts the time specified by *clock* to a local time specification. The function returns a pointer to a string consisting of 26 characters (see return value).

*clock* is represented as the time in seconds since 00:00:00 UTC (Universal Time Coordinated; January 1, 1970).

A call to `ctime()` has the same effect as `asctime(localtime(clock))`.

`ctime()` is not thread-safe. Use the reentrant function `ctime_r()` when needed.

### *BS2000*

`ctime()` interprets *clock* (see return values of `mktime()` and `time()`) as the number of seconds that have elapsed since January 1, 1950 00:00:00. From this number, `ctime()` calculates the local time (CET) and converts the result into an EBCDIC string. □

**Return value**

Pointer to a string

if successful. The resulting string has a length of 26 (incl. the null byte) and is formatted as a date and time specification in the form:

*weekday month day hrs:min:sec year*

e.g. Wed Dec 14 15:20:54 1988\n\n0

**Notes**      The return values of `ctime()` point to static data that is overwritten with each call.

**See also**    `altzone`, `asctime()`, `ctime_r()`, `daylight`, `gmtime()`, `localtime()`, `timezone`, `tzname`, `tzset()`, `time.h`.



## ctime\_r - thread-safe conversion of date and time to string

**Syntax**      `#include <time.h>`

```
char *ctime_r(const time_t *clock, char *buf);
```

**Description** `ctime_r()` converts the time specified by *clock* to the same format as `ctime()` and writes the result to the data area pointed to by *buf* (at least 26 bytes).

**Return value**

Pointer to the string pointed to by *buf*,  
if successful.

Null pointer    if an error occurs. (`errno` is set to indicate the error.)

**See also**    `asctime()`, `asctime_r()`, `ctime()`, `localtime()`, `localtime_r()`, `time()`.

## cuserid - get login name

Syntax `#include <stdio.h>`  
`char *cuserid(char *s);`

Description `cuserid()` generates a character representation of the name associated with the real user ID of the current process.

If `s` is a null pointer, this string is generated in an area that may be static and thus overwritten by subsequent calls to `cuserid()`. The address of this area is returned.

If `s` is not a null pointer, `s` is assumed to point to an array of at least `{L_cuserid}` bytes, and the string representation of the login name is placed in this array. The symbolic constant `{L_cuserid}` is defined in `stdio.h` and has a value greater than 0.

`cuserid()` is not thread-safe.

### Return value

`s` if `s` is not a null pointer. If the login name cannot be found, the null byte 0 will be placed at `*s`.

Address of the buffer containing the login name if `s` is a null pointer and the login name can be found.

Null pointer if `s` is a null pointer and the login name cannot be found.

Notes The functionality of `cuserid()` defined in the POSIX.1-1988 standard and XPG3 differs from that of historical implementations and XPG2. In the ISO POSIX-1 standard, the `cuserid` function has been removed entirely. Both functionalities are allowed in XPG4, but both are also marked **to be withdrawn**.

The XPG2 functionality can be obtained by using the following syntax:

```
getpwuid(getuid())
```

The XPG3 functionality can be obtained by using the following syntax:

```
getpwuid(geteuid())
```

See also `getlogin()`, `getpwnam()`, `getpwuid()`, `getuid()`, `geteuid()`, `stdio.h`, and the manual "POSIX Basics (BS2000/OSD)".

## \_\_DATE\_\_ - macro for compilation date

Syntax `__DATE__`

Description This macro generates the compilation date of a source file as a string in the form:

```
"dd Mmm yyyy\0"
```

where:

*dd* is the day (without leading zero for days < 10)

*Mmm* is the name of the month (abbreviated as with `asctime()`)

*yyyy* is the year

Notes This macro need not be defined in any header file. Its name is recognized and replaced by the compiler.

See also `asctime()`, `__TIME__`.

## daylight - daylight savings time variable

Syntax `#include <time.h>`  
`extern int daylight;`

Description The external variable `daylight` indicates whether time should reflect daylight savings time. `daylight` is non-zero if an alternate time zone exists. The timezone names are contained in the external variable `tzname`, which is set by default as follows:

```
char *tzname[2] = { "GMT", " " };
```

The functions `ctime()`, `localtime()`, `gmtime()` and `asctime()` take the peculiarities of the conversions for various time periods for the U.S. (specifically, the years 1974, 1975, and 1987) into account. They handle the new daylight savings time starting with the first Sunday in April, 1987.

Notes The system administrator must change the start and end date for daylight savings time each year if the Julian calendar format is used.

See also `altzone`, `asctime()`, `ctime()`, `gmtime()`, `localtime()`, `timezone`, `tzname`, `tzset()`, `time.h`.

## dbm\_clearerr, dbm\_close, dbm\_delete, dbm\_error, dbm\_fetch, dbm\_firstkey, dbm\_nextkey, dbm\_open, dbm\_store - functions for managing dbm databases

Syntax

```
#include <ndbm.h>

int dbm_clearerr(DBM *db);

void dbm_close(DBM *db);

int dbm_delete(DBM *db, date key);

int dbm_error(DBM *db);

date dbm_fetch(DBM *db, date key);

date dbm_firstkey(DBM *db);

date dbm_nextkey(DBM *db);

DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);

int dbm_store(DBM *db, date key, date content, int store_mode);
```

Description These functions manage pairs made up of a key and appropriate contents (*key/content*) of at least 1024 bytes in a database. The functions process very large databases (with one billion blocks) and access an object which has a key in one or two accesses to the file system. This package replaces the earlier `dbm` library, which can only manage one database at a time.

*key* and *content* are described by the type definition (typedef) *date*, where *date* specifies a string of *dsize* bytes to which *dptr* points. Both random binary data and normal ASCII strings are permitted.

The database is stored in two files. One file is a directory with the suffix `.dir` which contains a bit mask. The second file with the suffix `.pag` contains the data.

`dbm_open()` opens a database. The *file* argument must contain the pathname of the database. In this way, the files `file.dir` and `file.pag` are opened and/or created, depending on the *open\_flags* argument. The meaning of *open\_flags* corresponds to the meaning of *oflag* in the `open()` function (see page 609), except that in the case of the files of the database which are opened in `WRITE-ONLY` mode, write **and** read access is permitted. *file\_mode* has the same meaning as the third argument of `open()`.

`dbm_open()` returns a pointer to a structure of type `DBM`. This pointer must be passed by all remaining functions of this group as the *db* argument.

`dbm_close()` closes a database.

`dbm_fetch()` reads a record from the database. *key* is of the type `date` and must contain the value of the corresponding key of the record that is to be read.

`dbm_store()` writes a record to the database. *key* is of the type `date` and must contain the value of the corresponding key of the record that is to be written. Under this key the record can be read, modified or deleted at a later stage. *content* is also of the type `date` and contains the contents of the record that is to be written. The *store\_mode* argument can be either `DBM_INSERT` or `DBM_REPLACE`. With `DBM_INSERT`, only new entries are included in the database; an existing entry with the same key is not modified. With `DBM_REPLACE`, an existing entry is replaced if it has the same key, while with `DBM_INSERT` an existing entry with the same key is not replaced. If the specified key is not found in the database, `dbm_store()` enters the record in the database, regardless of whether *store\_mode* is set to `DBM_INSERT` or `DBM_REPLACE`.

`dbm_delete()` deletes a record and the associated key from the database. *key* is of the type `date` and must contain the value of the corresponding key of the record that is to be deleted.

`dbm_firstkey()` returns the first key in the database.

`dbm_nextkey()` returns the next key in the database each time. To be able to work with `dbm_nextkey()`, you must previously have called up `dbm_firstkey()`. Consecutive calls of `dbm_nextkey()` return the next key each time, until all keys in the database have been processed.

The `dbm_error()` function returns the error condition of the database. The *db* argument is a pointer to a database structure that was returned by a `dbm_open()` call.

The `dbm_clearerr()` function deletes the error condition of the database. The *db* argument is a pointer to a database structure that was returned by a `dbm_open()` call. `dbm_clearerr()` is not thread-safe.

## Return value

`dbm_open()`:

Pointer to a structure of type `DBM`  
                                   if successful.  
`(DBM *)0`                  if an error occurs.

`dbm_store()`:

0                          if successful.  
 1                          if *flags* has the value `DBM_INSERT` and the database already contains a record with the specified key.

Negative value if an error occurs.

`dbm_fetch()`:

`date content`  
                                   if successful.

`dptr = null pointer`  
                                   if the specified key was not found in the database or if an error occurs.

dbm\_delete():

0                   if successful.  
Negative value   if an error occurs.

dbm\_firstkey(), dbm\_nextkey():

date key        if successful.  
dptr = null pointer  
                  if the end of the database is reached or if an error occurs. In the event of an error, the error indicator of the database is also set.

dbm\_error():

0                   if the error condition is not set.  
≠ 0                if the error condition is set.

dbm\_clearerr():

The return value is undefined.

## Notes

The following code runs through the entire database:

```
for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

The `dbm_` functions made available in this library can on no account be compared with the functions of a general database management system. They do not allow multiple search key words in the same entry, they do not protect against multiple access (i.e. they do not lock records or files) and they also do not provide the variety of additional database functions that are offered in powerful database management systems. Because of the data copies after hash collisions, creating and updating databases with these functions is a relatively slow process. The `dbm_` functions are useful for applications that want to manage, without great expense, relatively static information that is indexed via a single key.

The `dptr` pointers returned by these functions point to a static memory, which can be modified via subsequent calls.

`dbm_delete()` does not physically restore the file area, but it does make it available for further use.

If the database is modified via `dbm_store()` or `dbm_delete()` calls during a sequential run through the database with the `dbm_firstkey()` and `dbm_nextkey()` functions, it is advisable to reset to the start of the database by calling `dbm_firstkey()`.

See also `open()`, `ndbm.h`

## **difftime - compute difference between two calendar time values**

**Syntax**      `#include <time.h>`  
`double difftime(time_t time1, time_t time0);`

**Description** `difftime()` computes the difference between two calendar times.

*time1* and *time0* are time values of type `time_t`. These time values are supplied by the `mktime()` and `time()` functions.

**Return value**  
*time1* - *time0*      if successful. The time difference is indicated in seconds and is of type `double`.

**See also**      `ctime()`, `mktime()`, `time()`, `time.h`.

## dirname - parent directory of pathname

Syntax `#include <libgen.h>`

```
char *dirname(char *path);
```

Description `dirname()` determines the parent directory of the pathname to which *\*path* points, and returns a pointer to a string which contains the name of this parent directory or the string `"."`. Trailing slashes (`/`) at the end of the pathname are not interpreted as part of the path.

If *path* does not contain a slash, `dirname()` returns a pointer to the `"."` string.

If *path* is a null pointer or points to an empty string, `dirname()` likewise returns a pointer to the `"."` string.

`dirname()` is not reentrant.

Return value

pointer to the name of the parent directory, or

pointer to `"."` string

If *path* does not contain a slash,  
is a null pointer or points to an empty string.

Example

Input value in *path*

Return value

```
"/usr/lib"
```

```
"/usr"
```

```
"/usr/"
```

```
"/"
```

```
"usr"
```

```
"."
```

```
"/"
```

```
"/"
```

```
"."
```

```
"."
```

```
".."
```

```
"."
```

The following code fragment reads a pathname, makes the parent directory into the current working directory, and opens the file:

```
char path(MAXPATHLEN), *pathcopy;
int fd;
fgets(path, MAXPATHLEN, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

Notes

`dirname()` can change the *path* string. The return value of `dirname()` can point to a static area that is overwritten by a subsequent `dirname()` call.

`dirname()` and `basename()` together produce a complete pathname. `dirname(path)` determines the pathname of the directory in which `basename(path)` resides.

See also

`basename()`, `libgen.h`.



## div - divide with integers

**Syntax**      `#include <stdlib.h>`  
`div_t div(int numer, int denom);`

**Description** `div()` computes the quotient and remainder of the division *numer* / *denom*.

The sign of the quotient is that of the algebraic quotient, and the magnitude of the quotient is the highest integer less than or equal to the absolute value of the algebraic quotient.

The remainder is expressed by the following equation:

$$\textit{quotient} * \textit{divisor} + \textit{remainder} = \textit{dividend}$$

**Return value**

Structure of type `div_t`  
if successful. The structure contains both the quotient `quot` and the remainder `rem` as integer values.

**See also**      `ldiv()`, `stdlib.h`.

## drand48 - generate pseudo-random numbers between 0.0 and 1.0

**Syntax**      `#include <stdlib.h>`

```
double drand48 (void);
double erand48 (unsigned short int xsubi[3]);
long int jrand48 (unsigned short int xsubi[3]);
void lcong48 (unsigned short int param[7]);
long int lrand48 (void);
long int mrand48 (void);
long int nrand48 (unsigned short int xsubi[3]);
unsigned short int *seed48 (unsigned short int seed16v[3]);
void srand48 (long int seedval);
```

**Description** This family of functions generates pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

`drand48()` and `erand48()` return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0 , 1.0].

`lrand48()` and `nrand48()` return non-negative, long integers, uniformly distributed over the interval [0,  $2^{31}$ ].

`mrand48()` and `jrand48()` return signed long integers uniformly distributed over the interval [ $-2^{31}$ ,  $2^{31}$ ].

`srand48()`, `seed48()` and `lcong48()` are initialization entry points, one of which should be invoked before either `drand48()`, `lrand48()` or `mrand48()` is called. Although it is not recommended, `drand48()`, `lrand48()` or `mrand48()` can be invoked without a prior call to an initialization entry point, since default initializer values are supplied automatically in such cases.

`erand48()`, `nrand48()` and `rand48()` do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values,  $X_i$ , according to the linear congruential formula:

$$X_{n+1} = (aX_n + c) \bmod m \quad n \geq 0$$

The parameter  $m = 2^{48}$ ; hence 48-bit integer arithmetic is performed. Unless `lcong48()` has been invoked, the multiplier value  $a$  and the addend value  $c$  are given by:

$$a = 5DEECE66D_{16} = 273673163155_8$$

$$c = B_{16} = 13_8$$

The value returned by any of the `drand48()`, `erand48()`, `jrand48()`, `lrand48()`, `mrand48()` or `nrand48()` functions is computed by first generating the next 48-bit  $X_i$  in the sequence. Then the appropriate number of bits, according to the type of variable to be returned, are copied from the high-order (leftmost) bits of  $X_i$  and transformed into the returned value.

The `drand48()`, `lrand48()` and `mrand48()` functions store the last 48-bit  $X_i$  generated in an internal buffer and must therefore be initialized prior to being invoked. The `erand48()`, `nrand48()` and `jrand48()` functions require the calling program to provide storage for the successive  $X_i$  values in the array specified as an argument when the functions are invoked. Consequently, these functions do not have to be initialized; the calling program merely has to place the desired initial value of  $X_i$  into the array and pass it as an argument.

By using different arguments, `erand48()`, `nrand48()` and `jrand48()` allow separate modules of a large program to generate several independent streams of pseudo-random numbers, i.e. the sequence of numbers in each stream will not depend on how many times the routines are called to generate numbers for the other streams.

The initializer function `srand48()` sets the high-order 32 bits of  $X_i$  to the value of the `{LONG_BIT}` bits contained in its argument. The low-order 16 bits of  $X_i$  are set to the arbitrary value  $330E_{16}$ .

The initializer function `seed48()` sets the value of  $X_i$  to the 48-bit value specified in the argument array. In addition, the previous value of  $X_i$  is copied into a 48-bit internal buffer, used only by `seed48()`, and a pointer to this buffer is the value returned by `seed48()`. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time; the pointer can be used to get at and store the last  $X_i$  value, and this value can then be used to reinitialize via `seed48()` when the program is restarted.

The initializer function `lcong48()` allows the user to specify default values for  $X_i$ , the multiplier value  $a$ , and the addend value  $c$ . Argument array elements `param[0]` to `param[2]` specify  $X_i$ , `param[3]` to `param[5]` specify the multiplier  $a$ , and `param[6]` specifies the 16-bit addend  $c$ . After `lcong48()` is called, a subsequent call to either `srand48()` or `seed48()` will restore the "standard" multiplier and addend values,  $a$  and  $c$ , specified above.

#### Return value

As described in the "Description" section above.

#### See also

`rand()`, `stdlib.h`.

## dup, dup2 - duplicate file descriptor

Syntax      `#include <unistd.h>`  
`int dup(int fildes);`  
`int dup2(int fildes, int fildes2);`

Description *fildes* is a file descriptor obtained from a `creat()`, `open()`, `dup()`, `fcntl`, or `pipe()` system call. `dup()` returns a new file descriptor having the following in common with the original file descriptor:

- the same open file or pipe
- the same file position indicator
- the same access mode (read, write or read/write)

*fildes2* is a non-negative integer that is less than `{OPEN-MAX}`. `dup2` causes *fildes2* to point to the same file as *fildes*. If *fildes2* already points to an open file other than *fildes*, the open file is first closed; however, if *fildes2* points to *fildes* or if *fildes* is not a valid file descriptor, *fildes* will not be first closed.

The `dup()` and `dup2()` functions provide an alternative interface to the service provided by `fcntl()` using the `F_DUPFD` command. The call:

```
fid = dup (fildes);
```

is equivalent to:

```
fid = fcntl (fildes, F_DUPFD, 0);
```

The call

```
fid = dup2 (fildes, fildes2);
```

is equivalent to:

```
close (fildes2);
```

```
fid = fcntl (fildes, F_DUPFD, fildes2);
```

except for the following:

If *fildes* is a valid file descriptor and is equal to *fildes2*, `dup2()` returns *fildes2* without closing it.

Return value

Non-negative integer (the file descriptor)

if successful.

-1 if an error occurs; `errno` is set to indicate the error.

Errors	<code>dup()</code> and <code>dup2()</code> will fail if:
EBADF	<i>fdes</i> is not a valid open file descriptor or the argument <i>fdes2</i> is negative or greater than or equal to <code>{OPEN_MAX}</code> .
EINTR	<code>dup2()</code> was interrupted by a signal.
<i>Extension</i>	
EINVAL	<i>fdes</i> and <i>fdes2</i> designate BS2000 files. □
EMFILE	The number of file descriptors in use by the process would exceed <code>{OPEN_MAX}</code> , or no <i>fdes2</i> file descriptors are available.
Notes	<code>dup()</code> and <code>dup2()</code> are executed only for POSIX files.
See also	<code>close()</code> , <code>fcntl()</code> , <code>open()</code> , <code>unistd.h</code> .

## ebcdic\_to\_ascii - convert EBCDIC string to ASCII string *(extension)*

Syntax	<code>int ebcdic_to_ascii(char *in, char *out);</code>
Description	<code>ebcdic_to_ascii</code> converts EBCDIC strings to ASCII strings, where <i>in</i> is the input string in EBCDIC code, and <i>out</i> is the output string in ASCII. The buffer must be supplied by the caller.  The characters of the input string are interpreted as EBCDIC characters and translated into corresponding characters in ASCII code.
Return value	0 if successful. 1 if an error occurs.
See also	<code>ascii_to_ebcdic</code> .

## ecvt, fcvt, gcvt - convert floating-point number to string

**Syntax**

```
#include <stdlib.h>

char *ecvt(double val, int num, int *dec_p, int *sign);
char *fcvt (double value, int ndigit, int *decpt, int *sign);
char *gcvt (double value, int ndigit, char *buf);
```

**Description** `ecvt()` converts a floating-point number *value* to a string of *ndigit* EBCDIC digits and returns a pointer to this string as its result. The output format corresponds to the `%f` format of `printf()`.

The string begins with the first non-zero digit of the floating-point number, i.e. leading zeros are not included.

The decimal character and a negative sign, if any, do not form a part of the string. However, `ecvt()` returns the position of the decimal point and the sign in result parameters.

*value* is a floating-point value that is to be edited for output.

*ndigit* is the number of digits in the result string (calculated from the first non-zero digit of the floating-point number to be converted). If *ndigit* is less than the number of digits in *value*, the least significant digit is rounded. If *ndigit* is greater, zero padding is used for right justification. The accuracy of the converted number is restricted by the maximum number of significant digits that can be represented in the type `double`.

*decpt* is the pointer to an integer specifying the position of the decimal character in the result string. If *\*decpt* is a positive number, the position of the decimal character relative to the beginning of the result string is specified. If *\*decpt* is a negative number or 0, the decimal character is to the left of the first digit. If the integer part of *value* cannot be represented in full with *ndigit* digits, *\*decpt* is greater than *ndigit*.

*sign* is the pointer to an integer specifying the sign of the result string. If *\*sign* is 0, the sign is positive; if *\*sign* is not 0: the sign is negative.

`fcvt()` is identical to `ecvt()`, except that *ndigit* specifies the number of digits after the decimal character.

If *ndigit* is less than the number of digits in *value* after the decimal character, the least significant digit is rounded. If *ndigit* is greater, zero padding is used for right justification.

`gcvt()` converts a floating-point number *value* into a string of EBCDIC digits according to the `%g` format of `printf()` and writes the prepared string to an array which is pointed to by *buf*. A pointer to this area is returned as the result. *ndigit* significant digits are generated (upper limit for *ndigit* is the number of significant digits which corresponds to the precision

of the type `double`). If *ndigit* is less than the number of digits in *value*, the least significant digit is rounded. If *ndigit* is greater, the string ends with the last digit that is not 0. If *value* represents an integer, *buf* is zero-padded for right justification.

In addition the string contains a minus sign if the value is  $< 0$ , and the decimal character if *value* is not an integer. The decimal character used is based on the current locale and is determined there by the category `LC_NUMERIC`. If the locale was not explicitly changed using `setlocale()`, the default value "POSIX" applies. In the POSIX locale the decimal character is a period (`.`).

Depending on the structure of the floating-point number to be converted, the output format corresponds to

- the `%f` format of `printf()`, or
- the `%e` format of `printf()` (exponential / scientific notation).

*ndigit* is the number of digits in the result string (calculated as of the first non-zero digit from the floating-point number to be converted).

*\*buf* is the pointer to the converted string.

The memory area pointed to by *buf* should be at least  $(ndigit + 4)$  bytes in size!

`ecvt()`, `fcvt()` and `gcvt()` are not thread-safe.

#### Return value

`ecvt()`, `fcvt()`:

Pointer to the converted EBCDIC string

if successful. The string is terminated with the null byte (`\0`).

`gcvt()`:

*\*buf* if successful. The string is terminated with the null byte (`\0`).

#### Notes

An invalid parameter, such as an integer value instead of a double value, will cause the program to abort.

Portable applications should use the `sprintf()` function instead of `ecvt()`, `fcvt()` and `gcvt()`.

`ecvt()` and `fcvt()`: The result is stored in an internal C data area which is overwritten with each subsequent call of one of these functions.

#### See also

`printf()`, `setlocale()`, `sprintf()`, `stdlib.h`.

**\_edt - call EDT** (*BS2000*)

Syntax        `#include <stdlib.h>`  
              `void _edt(void);`

Description `_edt` calls the BS2000 file editor EDT. Subsequently, when the file editor is terminated normally, the program continues at the next C statement that follows the `_edt` call.

Notes         Programs that call `_edt` require modules from the `EDTLIB` module library (under the `$TSOS` ID by default) during execution. A `RESOLVE` statement for this library must be issued when the modules are linked.

**encrypt - encode strings blockwise**

Syntax        `#include <unistd.h>`  
              `void encrypt(char block[64], int edflag);`

Description `encrypt()` provides access to an encoding algorithm. The key that is generated by `setkey()` is used as the *key* to encrypt the string *block* with the `encrypt()` function. *block* is a character array of length 64 bytes containing only the bytes with values 0 and 1. The argument array is modified in place to a similar array which contains the bits of the argument after modification by the encoding algorithm using the key set by `setkey()`. If *edflag* is 0, the argument is encoded. The argument cannot be decoded; if this is attempted (*edflag* = 1), `errno` is set to `ENOSYS`.

Errors        `encrypt()` will fail if:  
`ENOSYS`        The functionality is not supported by the system.

Notes         Since `encrypt()` does not return a value, errors can only be detected as follows: by setting `errno` to 0, calling `encrypt()`, and then testing `errno`. If `errno` is non-zero, it may be assumed that an error has occurred.

See also      `crypt()`, `setkey()`, `unistd.h`.



## endgrent, getgrent, setgrent - group management

**Syntax**

```
#include <grp.h>

void endgrent (void);

void setgrent (void);

struct group *getgrent (void);
```

**Description** `getgrent()` returns a pointer to an object with the structure shown below, which contains the individual fields of a line of the `/etc/group` file. Each line contains an object of the group structure, which is declared in the header file `grp.h`, with the following elements:

```
struct      group {
    char     *gr_name;      /* Name of the group */
    char     *gr_passwd;    /* Encoded group password */
    gid_t    gr_gid;       /* Numerical group ID */
    char     **gr_mem;     /* Pointer to names of the group members*/
};
```

`getgrent()` returns a pointer to the first group structure in the file the first time it is called; after this it returns a pointer to the next group structure in the file. In this way, consecutive calls can be used to search through the entire file.

`setgrent()` resets the file position indicator to the beginning of the group file, thus making a repeat search possible.

`endgrent()` can be called at the end of processing in order to close the group file.

`endgrent()`, `getgrent()` and `setgrent()` are not thread-safe.

### Return value

`getgrent()`:

Pointer to the first group structure of the group file  
at the first call

Pointer to the next group structure of the group file  
at subsequent calls

Null pointer at EOF or if an error occurs. `errno` is set to indicate the error.

**Errors** `getgrent()` will fail if:

EINTR `getgrent()` was interrupted by a signal.

EIO an I/O error occurred during reading or writing.

EMFILE The calling process contains `{OPEN_MAX}` open file descriptors.

ENFILE           The maximum permissible number of files is open in the system.

*Extension*

ENOMEM           There is not enough memory for storing the global data of `getgrent()`. □

**Notes**           The return value of `getgrent()` can point to an area that will be overwritten by a subsequent call of `getgrgid()`, `getgrnam()` or `getgrent()`.

These functions continue to be offered because they were common in the past. However, the format of the `group` structure depends on the implementation, which is why applications that use these functions are not portable. Portable applications should therefore use `getgrnam()` and `getgrgid()`.

**See also**        `getgrgid()`, `getgrnam()`, `getlogin()`, `getpwent()`, `grp.h`.

## endpwent, getpwent, setpwent - manage user catalog

**Syntax**

```
#include <pwd.h>

void endpwent (void);

struct passwd *getpwent (void);

void setpwent (void);
```

**Description** `getpwent()` returns a pointer to an object with the structure shown below, which contains the individual fields of a line of the `/etc/passwd` file. Each line contains an object of the `passwd` structure, which is declared in the header file `pwd.h`, with the following elements:

```
struct passwd {
    char    *pw_name;
    char    *pw_passwd;
    uid_t   pw_uid;
    gid_t   pw_gid;
    char    *pw_age;
    char    *pw_comment;
    char    *pw_gecos;
    char    *pw_dir;
    char    *pw_shell;
};
```

The components of this structure are read serially from the user catalog.

`getpwent()` returns a pointer to the first password structure in the user catalog the first time it is called; after this it returns a pointer to the next password structure in the file. In this way, consecutive calls can be used to search through the entire user catalog.

`setpwent()` deletes the pointer with which the user catalog is to be serially searched by means of `getpwent()`. A subsequent `getpwent` call returns a pointer to the first password structure.

`endpwent()` can be called at the end of processing in order to close the user catalog.

`endpwent()`, `getpwent()` and `setpwent()` are not thread-safe.

### Return value

`getpwent()`:

Pointer to the structure of type `passwd`  
if successful.

Null pointer at EOF and if an error occurs. `errno` is set to indicate the error.

Errors	<p>endpwent() will fail if:</p> <p>EACCES            the user ID (uid) of the caller is invalid.</p> <p>getpwent(), setpwent() and endpwent() will fail if:</p> <p>EFAULT            errors occur during creation of the passwd structure .</p> <p>ENOENT            the user does not exists.</p>
Notes	<p>The return value of getpwent() can point to an area that will be overwritten by a subsequent call of getpwuid(), getpwnam() or getpwent().</p> <p>There is no /etc/passwd password file in the POSIX subsystem. The user data is stored internally in the user catalog (see manual "POSIX Basics (BS2000/OSD)").</p> <p>These functions are only supported for reasons of compatibility.</p> <p>The characteristics of a current process can be defined as follows:</p> <ul style="list-style-type: none"><li>- getpwuid(geteuid()) returns the name of the effective user ID of the process</li><li>- getlogin() returns the login name of the process</li><li>- getpwuid(getuid()) returns the name of the real user ID of the process.</li></ul> <p>If error situations are to be investigated, errno must be set to 0 before getpwent() is called.</p>
See also	<p>endgrent(), getlogin(), getpwnam(), getpwuid(), putpwent(), pwd.h, manual "POSIX Basics (BS2000/OSD)".</p>

## endutxent, getutxent, getutxid, getutxline, pututxline, setutxent - manage utmpx entries

```
Syntax    #include <utmpx.h>
          void endutxent (void);
          struct utmpx *getutxent (void);
          struct utmpx *getutxid (const struct utmpx *id);
          struct utmpx *getutxline (const struct utmpx *line);
          struct utmpx *pututxline (const struct utmpx *utmpx);
          void setutxent (void);
```

**Description** These functions allow access to the user accounting file `/var/adm/utmpx`.

`getutxent()`, `getutxid()` and `getutxline()` return a pointer to a structure of the following type:

```
struct    utmpx {
    char    ut_user[32]; /* Login name of the user */
    char    ut_id[4]; /* /sbin/inittab ID (normally line no.) */
    char    ut_line[32]; /* Device name (console, lnxx) */
    pid_t   ut_pid; /* Process ID */
    short   ut_type; /* Type of entry */
    struct  exit_status {
        short e_termination; /* End status */
        short e_exit; /* Exit status */
    } ut_exit; /* Exit status of a process marked as DEAD_PROCESS */
    struct timeval ut_tv; /* Time entry made */
    short ut_syslen; /* Significant length of ut_host */
                /* including trailing zero */
    char    ut_host[257]; /* Host name if given */
};
```

`getutxent()` reads the next entry from a `utmpx`-similar file. If the file is not yet open, it will be opened. If the end of the file is reached, the operation fails.

`getutxid()` searches forward from the current position in the `utmpx` file until an entry is found whose `ut_type` matches the `id->ut_type` if the specified type is `RUN_LVL`, `BOOT_TIME`, `OLD_TIME` or `NEW_TIME`. If the type specified in `id` is `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS` or `DEAD_PROCESS`, then `getutxid()` returns a pointer to the first entry whose type matches one of these four types and whose `ut_id` component matches the value of the transferred `id->ut_id`. If the end of the file is reached before a matching entry is found, the operation fails.

In all entries that are found with `getutxid()`, the `ut_type` component identifies the type of the entry. Depending on the value of `ut_type`, each entry contains further data that is significant for the processing:

Value of <code>ut_type</code>	Other components
EMPTY	No further data
BOOT_TIME	<code>ut_tv</code>
OLD_TIME	<code>ut_tv</code>
NEW_TIME	<code>ut_tv</code>
USER_PROCESS	<code>ut_id</code> , <code>ut_user</code> (login name), <code>ut_line</code> , <code>ut_pid</code> , <code>ut_tv</code>
INIT_PROCESS	<code>ut_id</code> , <code>ut_pid</code> , <code>ut_tv</code>
LOGIN_PROCESS	<code>ut_id</code> , <code>ut_user</code> (implementation-specific name of the login process), <code>ut_pid</code> , <code>ut_tv</code>
DEAD_PROCESS	<code>ut_id</code> , <code>ut_pid</code> , <code>ut_tv</code>

`getutxline()` searches forwards from the current position in the `utmpx` file until an entry with the type `LOGIN_PROCESS` or `USER_PROCESS` is found, whose `ut_line` string matches `line->ut_line`. If the end of the file is reached before a matching entry is found, the operation fails.

`pututxline()` writes the specified `utmpx` structure to the `utmpx` file. `getutxid()` is used to search for the correct position in the file if this is not given. It is expected that the user of `pututxline()` has searched for the corresponding entry with one of the `getutx()` functions. If this is the case, `pututxline()` does not perform a search. If `pututxline()` does not find an appropriate position for the new entry, the entry is added to the end of the file. A pointer to the `utmpx` structure is returned.

To be able to use `pututxline()`, the process must have the appropriate privileges.

`setutxent()` sets the position of the input stream to the beginning of the file. This should be done before the whole file is searched for a new entry.

`endutxent()` closes the opened file.

`endutxent()`, `getutxent()`, `getutxid()`, `getutxline()`, `pututxline()` and `setutxent()` are not thread-safe.

## Return value

`getutxent()`, `getutxid()` and `getutxline()`:

Pointer to a `utmpx` structure

if successful. The returned structure contains a copy of the desired entry in the user accounting file.

Null pointer     ar EOF or if an error occurs.

`pututxline()`:

Pointer to a `utmpx` structure

if successful. The returned structure contains a copy of the entry that was written to the user accounting file.

Errors `pututxline()` will fail if:

`EPERM` the process does not have sufficiently high privileges.

Notes The return value points to a static area that will be overwritten by a subsequent call of `getutxid()` or `getutxline()`.

The latest entry is stored in a static structure. Before the file is accessed again, this entry must be copied. When `getutxid()` or `getutxline()` are called, the routines check the static structure before further I/O operations are performed. If the contents of the static structure match the pattern being sought, the search is discontinued. If several identical entries are to be sought with `getutxline()`, the static structure must be deleted after every successful search operation; otherwise `getutxline()` will keep returning the same structure.

The implicit reading via `pututxline()` (if the correct position in the file has not yet been reached) does not change the contents of the static structure that is returned by `getutxent()`, `getutxid()` or `getutxline()`, as `pututxline()` saves the contents of the structure before reading.

The size of the arrays in the structure can be determined via the `sizeof` operator.

See also `utmpx.h`.

## environ - external variable for environment

**Syntax**      `extern char **environ;`

**Description** `environ` is an external variable that points to an array of strings with environment variables, called the "environment" in short. Each string in the array has the form "*name=value*", where *name* designates the environment variable and *value* represents its current value. Environment variables provide a way to make information about a program's environment available to applications (see section "Environment variables" on page 71).

**Notes**        The `environ` array should not be directly accessed by the application.

**See also**     `exec`, `getenv()`, `putenv()`, section "Environment variables" on page 71.

## erand48 - generate pseudo-random numbers between 0.0 and 1.0 with initialization value

**Syntax**        `#include <stdlib.h>`  
`double erand48 (unsigned short int xsubi[3]);`

**Description** See `drand48()`.



## erf, erfc - error and complementary error functions

**Syntax**      `#include <math.h>`  
`double erf(double x);`  
`double erfc(double x);`

**Description** `erf()` computes the error function of the floating-point number  $x$ . The error function is defined as follows:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

`erfc()` computes the complementary error function of the floating-point number  $x$ :

$$1.0 - \text{erf}(x).$$

**Return value**

Value of the error function of  $x$   
if `erf()` was completed successfully.

Value of the complementary error function of  $x$   
if `erfc()` was completed successfully.

**Notes** `erfc()` is provided due to the resulting loss of accuracy when the error function `erf()` is called for large values of  $x$ .

**See also** `math.h`.

## errno - variable for error return values

**Syntax**      `#include <errno.h>`

**Description** `errno` is used by many functions to return error values. Programs obtain the definition of `errno` by the inclusion of the `errno.h` header. `errno` is set to an error number of type `int` (see `errno.h` and section "Error handling" on page 126).

The value of `errno` is 0 at program startup, but it is never set to 0 to indicate an error by any function described in this manual. The value of `errno` will be defined only after a function call (see the "Errors" section for each function) and may be modified by a subsequent function call.

A program that uses `errno` for error checking should therefore set it to 0 before a function call and subsequently inspect it before a new function call.

**Notes**      `errno` should not be declared in the source code; however, existing source code need not be modified.

A mapping between the numeric values and symbolic names of the error numbers is not guaranteed. Correct behavior is guaranteed only when using the symbolic constant names. Furthermore, the mapping of error conditions to `errno` values is guaranteed only for the cases required by X/Open.

**See also**      `perror()`, `strerror()`, `errno.h`, section "Error handling" on page 126.

**exec: execl, execlv, execlx, execve, execlp, execvp - execute file**

**Syntax**

```
#include <unistd.h>
extern char **environ;

int execl (const char *path, const char *arg0, ... , (char *)0 );
int execlv (const char *path, char *const argv[ ] );
int execlx (const char *path, const char *arg0, ... , (char *)0, char *const envp[ ] );
int execve (const char *path, char *const argv[ ], const char *envp[ ] );
int execlp (const char *file, const char *arg0, ... , (char *)0 );
int execvp (const char *file, char *const argv[ ] );
```

**Description** The `exec` family of functions replaces the current process image with a new process image. The new image is constructed from a regular, executable file (*path* or *file*) called the new process image file. There is no return from a successful `exec`, because the calling process image is overlaid by the new process image.

When a C program is executed as a result of a call to an `exec` function, it is entered as a C function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. *argc* is at least 1, and the first element of the array points to a string containing the name of the executable file.

In addition, the following variable is initialized as the address of an array of `char` pointers to the environment variables:

```
extern char **environ;
```

*argv* and *environ* are each terminated by a null pointer. The null pointer terminating the *argv* array is not counted in *argc*.

The arguments specified by a program with one of the `exec` functions are passed on to the new process image in the corresponding `main()` arguments.

*path* points to a pathname that identifies the new process image file.

*file* is used to construct a pathname that identifies the new process image file. If *file* contains a slash character, then the *file* argument is used as the pathname for the process image file. Otherwise, the path prefix for this file is obtained by a search of the directories defined by the environment variable `PATH` (see section "Environment variables" on page 71). The environment is typically supplied by the POSIX shell (see also the manual "POSIX Basics (BS2000/OSD)"). Other X/Open-compatible systems may define alternate mechanisms for this purpose.

If the process image file is not a valid executable object, `exec1p()` and `execvp()` use the contents of that file as standard input to a command interpreter conforming to `system()`. In this case, the command interpreter becomes the new process image.

*arg0*, ... are pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. The list is terminated by a null pointer. The argument *arg0* should point to a filename that is associated with the process being started by one of the `exec` functions.

*argv* is an array of character pointers to null-terminated strings. The last element in this array must be a null pointer. These strings constitute the argument list for the new process image. The value in *argv*[0] should point to a filename that is associated with the process being started by one of the `exec` functions.

*envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The *envp* array is terminated by a null pointer.

In the case of functions which do not pass *envp* as an argument (`exec1()`, `execv()`, `exec1p()` and `execvp()`), the environment for the new process image is taken from the external variable `environ` in the calling process.

The number of bytes available for the combined argument and environment lists of the process is `{ARG_MAX}`. In the POSIX subsystem, the `{ARG_MAX}` constant includes the space for null terminators, pointers, and/or any alignment bytes. This may be implemented differently on other X/Open-compatible systems.

File descriptors open in the calling process image remain open in the new process image, except for those whose close-on-exec flag `FD_CLOEXEC` is set (see also `fcntl()`). For those file descriptors that remain open, all attributes of the open file description, including file locks, remain unchanged.

The state of conversion descriptors and message catalog descriptors in the new process image is undefined. For the new process, the equivalent of:

```
setlocale(LC_ALL, "C")
```

is executed at startup.

Signals set to the signal action `SIG_DFL` in the calling process image are set to the default signal action in the new process image. Signals set to be ignored (`SIG_IGN`) by the calling process image are also ignored by the new process image. Signals set to be caught by the calling process image are set to the default signal action in the new process image (see also `signal.h`).

After a successful call to any of the `exec` functions, any functions previously registered by the `atexit()` function are no longer registered.

If the set-user-ID mode bit is set for the new process image file (see also `chmod()`), the effective user ID of the new process image is set to the user ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved as the saved set-user-ID and the saved set-group-ID for use by `setuid()`.

Any shared memory segments attached to the calling process image will not be attached to the new process image (see also `shmat()`).

The new process also inherits the following attributes from the calling process image:

- nice value (see also `nice()`)
- `semadj` values (see also `semop()`)
- process ID
- parent process ID
- process group ID
- session ID
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see also `alarm()`)
- current working directory
- root directory
- file mode creation mask (see also `umask()`)
- file size limit (see also `ulimit()`)
- process signal mask (see also `sigprocmask()`)
- pending signals (see also `sigpending()`)
- `tms_utime`, `tms_stime`, `tms_cutime` and `tms_cstime` (see also `times()`)

All other process attributes of the XPG4-compliant library functions will be the same in the new and old process images.

Upon successful completion, the `exec` functions mark for update the `st_atime` field of the file. If an `exec` function failed but was able to locate the process image file, whether the `st_atime` field is marked for update is unspecified. Should the `exec` function succeed, the process image file is considered to have been opened with `open()`. The corresponding `close()` is considered to occur at a time after this `open`, but before process termination or successful completion of a subsequent call to one of the `exec` functions.

POSIX files are closed on calling an `exec` function only if the `CLOSE_ON_EXEC` flag is set.

If threads are used, then the function affects the process or a thread in the following manner:

- When one of the `exec()` functions are called in a process with more than one thread, all threads are terminated and then the new executable program is loaded and executed. No destructor functions are called.

*BS2000*

- BS2000 files are always closed on calling an `exec()` function. □

#### Return value

- 1 if an error occurs. `errno` is set to indicate the error.

#### Errors

The `exec` functions will fail if:

**E2BIG** The number of bytes used by the argument list and environment list of the new process image is greater than the system-imposed limit of `{ARG_MAX}` bytes.

**EACCES** Search permission is denied for a directory listed in the path prefix of the new process image, or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.

*Extension*

**EFAULT** The program could not be loaded.

**EINTR** A signal was caught.

**ELOOP** Too many symbolic links were encountered in resolving *path* or *file*. □

**ENAMETOOLONG**

The length of the *path* or *file* arguments, or an element of the environment variable `PATH` prefixed to a file, exceeds `{PATH_MAX}`, or a pathname component is longer than `{NAME_MAX}`.

*Extension*

**ENOENT** One or more components of the pathname of the new process image file does not exist, or *path* or *file* points to an empty string.

**ENOMEM** A new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

**ENOTDIR** A component of the path prefix of the new process image file is not a directory. □

The `exec` functions - except for `execlp()` and `execvp()` - will fail if:

`ENOEXEC`      The new process image file has the appropriate access permissions, but is not in the proper format.

**Notes**      Since the state of conversion descriptors and message catalog descriptors in the new process image is undefined, portable applications should not rely on their use and should close them prior to calling one of the `exec` functions.

Before the program to be executed is loaded, the environment variables `BLSLIB $nm$`  (where  $00 \leq nm \leq 98$ ) are evaluated in ascending order, starting with `BLSLIB00`. The contents of the variables are interpreted as BS2000 file names, and a link to each respective file name is set up using the variable name. The search is aborted at the first variable that does not exist; however, a link to the file `$.SYSLNK.CRTE` is created with the link name `BLSLIB99` in any case. This mechanism allows incompletely linked programs, which need to load modules dynamically, to be executed in a child process that does not inherit the TFT (terminal file table) from its parent process.

**See also**    `alarm()`, `atexit()`, `exit()`, `fcntl()`, `fork()`, `getenv()`, `nice()`, `putenv()`, `semop()`, `setlocale()`, `shmat()`, `sigaction()`, `system()`, `times()`, `ulimit()`, `umask()`, `unistd.h`, section "Environment variables" on page 71.

## exit, \_exit - terminate process

Syntax `#include <stdlib.h>`  
`void exit (int status);`  
  
`#include <unistd.h>`  
`void _exit (int status);`

Description `_exit()` and `exit()` terminate the calling process.

A call to `exit` triggers the following actions:

1. `exit()` first calls all functions registered by `atexit()`, in the reverse order of their registration. If a function registered by a call to `atexit()` fails to return, the remaining registered functions are not called and the execution of `exit()` is aborted. If `exit()` is called more than once, the effects are undefined.
2. `exit()` then flushes all output streams, closes all open streams, and removes all files created by `tmpfile()`.

In contrast to `exit()`, the `_exit()` function does not call the process termination functions registered with `atexit()` and does not close the opened files.

`_exit()` and `exit()` terminate the calling process with the following consequences:

- All of the file descriptors, directory streams and message catalog descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a `wait()` or `waitpid()`, it is notified of the termination of the calling process, and the low-order eight bits (i.e. bits 0377) of *status* are made available to it (see also `wait()` and `waitpid()`).
- If the parent is not waiting, the child's status will be made available to it when the parent subsequently executes a `wait()` or `waitpid()`.
- If the parent process of the calling process is not executing a `wait()` or `waitpid()`, the calling process is transformed into a **zombie process**. A zombie process is an inactive process that will be deleted at some later time when its parent process executes a `wait()` or `waitpid()`.
- The termination of a process does not directly terminate its children. The sending of a `SIGHUP` signal as described below indirectly terminates children in some circumstances.
- In the POSIX subsystem, the `SIGCHLD` signal is also sent to the parent process. Other X/Open-compatible implementations may provide alternate mechanisms for this purpose.



- The parent process ID of all of the calling process's existing child processes and zombie processes is set to the process ID of a special system process. In other words, these processes are inherited by the special system process `init` (whose process ID is 1).
- Each attached shared-memory segment is detached, and the value of `shm_nattch` (see `shmget()`) in the data structure associated with its shared memory ID is decremented by 1.
- For each semaphore for which the calling process has set a `semadj` value (see `semop()`), that `semadj` value is added to the `semval` of the specified semaphore.
- If the process is a controlling process, the `SIGHUP` signal will be sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.
- If the exit of the process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a `SIGHUP` signal followed by a `SIGCONT` signal will be sent to each process in the newly-orphaned process group.

The symbolic names `EXIT_SUCCESS` and `EXIT_FAILURE` are defined in `stdlib.h` and may be used as the value of `status` to indicate successful or unsuccessful completion.

`exit()` and `_exit()` do not return.

If threads are used, then the function affects the process or a thread in the following manner:

- The process is terminated. Threads that are terminated by calling `_exit()` do not call their cancellation cleanup handler or the data destructors of the thread.

*BS2000*

- The monitor job variable `MONJV` is supplied in accordance with the following rules:
- Depending on the value of the `status` argument, the status indicator of the monitoring job variable `MONJV` (1st to 3rd byte) is set to the value `"$T "` or `"$A "`, and the variables `SUBCODE1`, `SUBCODE2` and `MAINCOD`, which can be queried with the identically named predefined functions of `SDF-P`, are supplied.

*status* may contain the symbolic constants `EXIT_SUCCESS` and `EXIT_FAILURE` (defined in the header file `stdlib.h`) or any integer value:

`EXIT_SUCCESS` (value 0)

causes normal program termination.

The status indicator of the `MONJV` is assigned the value "\$T". In addition, the following settings are made: `SUBCODE=0`, `MAINCODE=CCM0998` and `SUBCODE2=status modulo 256`.

`EXIT_FAILURE` (value 9990888)

causes a so-called **job-step termination**:

- The program is terminated abnormally.
- In a `DO` or `CALL` procedure, the system branches to the next `ABEND`, `END-PROCEDURE`, `SET-JOB-STEP` or `LOGOFF` command.
- The system message "ABNORMAL PROGRAM TERMINATION" is issued.

The status indicator of the `MONJV` is assigned the value "\$A", and `SUBCODE=1`, `MAINCODE=CCM0999` and `SUBCODE2=status modulo 256` are set.

integer value  $\neq 0$  and  $\neq 9990888$

A job-step termination is performed, and the status indicator of the `MONJV` is assigned the value "\$T". Furthermore, `SUBCODE=1`, `MAINCODE=CCM0999`, and `SUBCODE2=status modulo 256` are set.

If this value corresponds to the predefined values `EXIT_SUCCESS` or `EXIT_FAILURE`, the actions indicated above are performed. □

Notes Applications should normally use `exit()` rather than `_exit()`.

*BS2000*

In order to supply and query monitoring job variables, the C-language program must be started from `BS2000` with the command:

```
/START-PROG program,MONJV=monjvname
```

The contents of the job variables can then be checked, e.g. with the following command:

```
/SHOW-JV JV-NAME(monjvname)
```

Further information on the use of monitoring job variables for runtime monitoring can be found in the "Job Variables (BS2000)" manual. □

See also `abort()`, `atexit()`, `bs2exit()`, `close()`, `fclose()`, `semop()`, `shmget()`, `sigaction()`, `wait()`, `stdlib.h`, `unistd.h`.

## exp - use exponential function

**Syntax**      `#include <math.h>`  
                 `double exp(double x);`

**Description** `exp()` computes the exponential function for a permitted floating-point number  $x$ .

**Return value**  
 $e^x$                     if successful.  
HUGE\_VAL                if an overflow occurs. `errno` is set to indicate the error.

**Errors**        `exp()` will fail if:  
ERANGE            Overflow; the return value is too high.

**See also**      `log()`, `log10()`, `pow()`, `math.h`.

## expm1 - compute exponential function

**Syntax**        `#include <math.h>`  
                 `double expm1(double x);`

**Description** The `expm1()` function computes  $e^x - 1.0$ .

**Return value**  
 $e^x - 1.0$                 if successful.  
HUGE\_VAL                if an overflow occurs. `errno` is set to indicate the error.

**Notes**         For small  $x$  values, the result of `expm1(x)` can be more accurate than the value of `exp(x) - 1.0`. The functions `expm1()` and `log1p()` are helpful for computing the expression  $((1+x)^n - 1)/x$ , in the format: `expm1(n * log1p(x))/x` in the case of very small values of  $x$ .  
This function can also be used to precisely represent inverse hyperbolic functions.

**See also**      `exp()`, `ilogb()`, `log1p()`, `math.h`.

## fabs - compute absolute value of floating-point number

Syntax        `#include <math.h>`  
              `double fabs(double x);`

Description `fabs()` computes the absolute value of the floating-point number *x*.

Return value  
              Absolute value of the floating-point number *x*  
                                  if successful.

See also      `abs()`, `cabs()`, `ceil()`, `floor()`, `math.h`.

## fattach - assign file descriptor under STREAMS to object in name space of file system

Syntax        `#include <stropts.h>`  
              `int fattach (int fildes, const char *path);`

Description The `fattach()` function assigns a file descriptor under STREAMS to an object (file or directory) in the name space of the file system, and *fildes* is assigned a pathname. *fildes* must be a valid, open file descriptor which represents a STREAMS file. *path* is a pathname of an existing object. The process must have appropriate privileges, or must be the owner of the file *path* and have write permission. All subsequent operations on *path* work with the STREAMS file until the assignment of the STREAMS file to the node is canceled. *fildes* can be assigned to more than one path, i.e. the file descriptor can be assigned more than one name.

The attributes of the given stream are initialized as follows (see also `stat()`): access rights, user and group IDs and file times are the same as those of *path*, the number of links is set to 1 and the size and device identifier are set to the same values as the STREAMS device of *fildes*. If any attributes of the given are then subsequently modified (e.g. with `chmod()`), neither the attributes of the underlying object nor the attributes of the STREAMS file referred to by *fildes* are affected.

File descriptors which refer to the underlying object and were opened before an `fattach()` call continue to refer to the underlying object.

## Return value

- 0 if successful.
- 1 if an error occurs. `errno` is set to indicate the error.

## Errors

`fattach()` will fail if:

- EACCES Search permission is denied for a component of the path, or if the user is the owner of *path* but does not have write permission for *path*.
- EBADF *fdes* is not a valid open file descriptor.
- ENOENT A component of the pathname does not exist, or *path* points to an empty string.
- ENOTDIR A component of the pathname prefix is not a directory.
- EPERM The effective user ID of the process is not that of the owner of the file identified by *path* and the process does not have the appropriate access permissions.
- EBUSY *path* is currently a mount point or a STREAMS file is assigned to this path.
- ENAMETOOLONG The length of *path* exceeds `{PATH_MAX}`, or a component of the pathname is longer than `{NAME_MAX}`, while `{_POSIX_NO_TRUNC}` is active; or The resolving of a symbolic link of the pathname generates an interim result which is longer than `{PATH_MAX}`.
- ELOOP Too many symbolic links were encountered in resolving *path*.
- EINVAL *fdes* does not represent a STREAMS file.
- EREMOTE *path* is a file in a remotely mounted directory.

## Notes

`fattach()` behaves similarly to the older `mount()` function in that an object is temporarily replaced by the root directory of the mounted file system. With `fattach()`, the replaced object need not be a directory and the replacing file is a STREAMS file.

## See also

`fdetach()`, `isastream()`, `stropts.h`.

## fchdir - change current directory

**Syntax**      `#include <unistd.h>`  
`int fchdir(int fildev);`

**Description** Like `chdir()`, `fchdir()` also changes the current directory. The new directory is identified by the file descriptor *fildev*. The current directory is the starting point for the search for pathnames which do not begin with “/”. The *fildev* argument is an open file descriptor referencing a directory.  
To make a directory the current directory, a process must have execute (search) permission for the directory.

**Return value**

0	if successful.
-1	if an error occurs. The current working directory remains unchanged. <code>errno</code> is set to indicate the error.

**Errors**      `fchdir()` will fail if:

EACCES	There is no search permission for <i>fildev</i> .
EBADF	<i>fildev</i> is not a file descriptor for an open file.
ENOTDIR	The open file descriptor does not point to a directory.
EINTR	A signal was caught during the <code>fchdir()</code> system call.
EIO	An I/O error occurred during reading or writing from the file system.
ENOLINK	<i>fildev</i> refers to a remote computer and the link to this computer is no longer active.

**Notes**      The change of the current directory is effective for the duration of the current program (or of the current shell). If a new program or shell is started, the home directory is again set as the current directory.  
To make a directory the current directory, a process must have execute permission (search) for the directory.  
`fchdir()` is only effective in the currently active process and only until the active program terminates.  
`fchdir()` is only executed for POSIX files.

**See also**      `chdir()`, `chroot()`, `unistd.h`.

## fchmod - change mode of file

Syntax `#include <sys/types.h>`  
`#include <sys/stat.h>`

```
int fchmod(int fildev, mode_t mode);
```

Description Like `chmod()`, `fchmod()` changes `S_ISUID`, `S_ISGID` and the file mode bits of the addressed file into the corresponding bits of *mode*, except that the file whose access permissions are to be changed is identified not by the pathname but by the file descriptor *fildev*. The file mode bits are interpreted as follows (see also `sys/stat.h`):

Symbolic name	Bit pattern	Meaning
<code>S_ISUID</code>	04000	Set user ID on execution
<code>S_ISGID</code>	020#0	Set group ID on execution if the value of # is 7, 5, 3 or 1. Remove mandatory lock on files and file records if # is 6, 4, 2 or 0
<code>S_ISVTX</code>	01000	Save text segment after execution
<code>S_IRWXU</code>	00700	Read, write or execute (search) by owner
<code>S_IRUSR</code>	00400	Read by owner
<code>S_IWUSR</code>	00200	Write by owner
<code>S_IXUSR</code>	00100	Execute (search if a directory) by owner
<code>S_IRWXG</code>	00070	Read, write or execute (search) by group
<code>S_IRGRP</code>	00040	Read by group
<code>S_IWGRP</code>	00020	Write by group
<code>S_IXGRP</code>	00010	Execute by group
<code>S_IRWXO</code>	00007	Read, write or execute (search) by others
<code>S_IROTH</code>	00004	Read by others
<code>S_IWOTH</code>	00002	Write by others
<code>S_IXOTH</code>	00001	Execute by others

Other modes are constructed by a bit-wise OR combination of the file mode bits.

The effective user ID of the process must match the owner of the file or the process must have the appropriate privilege to change the mode of a file.

If neither the process nor a member of the supplementary group list is privileged, and if the effective group ID of the process does not match the group ID of the file, the mode bit 02000 (set group ID on execution) is cleared.

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may affect future calls to `open()`, `creat()`, `read()` and `write()` on this file.

If the process is not a privileged process and the file is not a directory, the mode bit 01000 (save text segment after execution) is deleted.

If a directory can be written to and the sticky bit is set, files in this directory can only be deleted or renamed if at least one of the following is true (see `unlink()` and `rename()`):

- the file belongs to the user
- the directory belongs to the user
- the user has right permission for the file
- the user is a privileged user

On successful completion, `fchmod()` marks the `st_ctime` field of the file for update.

#### Return value

- 0 if successful.
- 1 if an error occurs. The file mode is not changed. `errno` is set to indicate the error.

#### Errors

`fchmod()` will fail if:

- `EBADF` *fildes* is not an open file descriptor.
- `EINVAL` An attempt was made to access a BS2000 file, or the value of *mode* is invalid.
- `EIO` An I/O error occurred while reading from or writing to the file system.
- `EINTR` A signal was caught during execution of the `fchmod` system call.
- `EPERM` The user ID does not match that of the file owner, and the process does not have the appropriate privileges.
- `EROFS` The file referred to by *fildes* resides on a read-only file system.

#### Notes

`fchmod()` is executed only for POSIX files.

#### See also

`chmod()`, `chown()`, `creat()`, `fcntl()`, `fstatvfs()`, `mknod()`, `open()`, `read()`, `rename()`, `stat()`, `unlink()`, `write()`, `sys/stat.h`, `sys/types.h`.



## fchown - change owner or group of file

**Syntax**        `#include <unistd.h>`  
`int fchown(int fildes, uid_t owner, gid_t group);`

**Description** Like `chown()`, `fchown()` changes the user ID and the group ID of the addressed file, except that the file is not identified by the pathname but by the file descriptor *fildes*. The user ID is set to *owner* and the group ID is set to *group*. If *owner* or *group* is specified as -1, the corresponding ID is not changed.

If `fchown()` is called by a process without appropriate privileges, the bits set-user-ID on execution and set-group-ID on execution, i.e. `S_ISUID` and `S_ISGID`, are cleared (see `chmod()`).

The effective user ID of the process must match the owner of the file or the process must have the appropriate privilege to change ownership of a file.

On successful completion, `fchown()` marks the `st_ctime` field of the file for update.

**Return value**

0	if successful. The user ID and group ID of the specified file are set as required.
-1	if an error occurs. The user ID and group ID of the file are not changed, and <code>errno</code> is set to indicate the error.

**Errors**        `fchown()` will fail if:

EABDF	<i>fildes</i> does not point to an open file.
EINTR	A signal was caught during the system call.
EINVAL	An attempt was made to access a BS2000 file. <i>group</i> or <i>owner</i> is not in the permissible range.
EIO	An I/O error occurred while reading from or writing to the file system.
EPERM	The user ID does not match the owner of the file, or the process does not have the appropriate privileges.
EROFS	The file resides on a read-only file system.

**Notes**        `fchown()` is executed only for POSIX files

**See also**     `chmod()`, `chown()`, `unistd.h`.

## fclose - close stream

Syntax `#include <stdio.h>`  
`int fclose(FILE *stream);`

Description `fclose()` causes the buffer of the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream is written to the file; any unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated. The `fclose()` function will perform a `close()` on the file descriptor that is associated with the stream pointed to by *stream*.

After the call to `fclose()`, the behavior of *stream* is undefined.

### Return value

0 if successful  
EOF if an error occurs; `errno` is set to indicate the error.

### Errors

`fclose()` will fail if:

EAGAIN The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

EBADF The file descriptor underlying *stream* is not valid.

#### *Extension*

The BS2000 file is not accessible in the process. □

EFBIG An attempt was made to write a file that exceeds the maximum file size or the process file size limit (see also `ulimit()`).

EINTR `fclose()` was interrupted by a signal.

EIO An I/O error occurred. The process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking the `SIGTTOU` signal, and the process group of the process is orphaned.

ENOSPC There was no free space remaining on the device containing the file.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

**EPIPE** An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the process. If threads are used, then the function affects the process or a thread in the following manner: If an `EPIPE` error occurs, the `SIGPIPE` signal is not sent to the process, but is sent to the calling thread instead.

**Notes** Whenever a program is terminated normally or with `exit()`, an `fclose()` is automatically executed for every open file. In other words, `fclose()` need not be called explicitly except in cases when a file needs to be closed before program termination, e.g. to avoid exceeding the limit for open files (=2048).

The program environment determines whether `fclose()` is executed for a BS2000 or POSIX file.

*BS2000*

If *stream* does not point to a `FILE` structure, the program is aborted.

Since no data is buffered for record I/O, there is no internal call to the `fflush()` function. □

**See also** `close()`, `exit()`, `fflush()`, `fopen()`, `setbuf()`, `stdio.h`.

## fcntl - control open file

**Name**        **fcntl**

**Syntax**      `#include <fcntl.h>`

*Optional*

`#include <sys/types.h>`

`#include <unistd.h>`

`int fcntl(int fil-des, int cmd, ... / * arg */);`

**Description** `fcntl()` provides for control over open files.

*fil-des* is a file descriptor of an open file.

`fcntl()` can take a third argument, whose data type and value depend upon the value of the passed command *cmd*. The command *cmd* specifies the operation to be performed by `fcntl()` and may be one of the following:

- |                      |   |
|----------------------|---|
| <code>F_DUPFD</code> | Returns a new file descriptor with the following characteristics: <ul style="list-style-type: none"><li>– Lowest numbered available (i.e. open) file descriptor greater than or equal to the integer value passed as the third argument (<i>arg</i>).</li><li>– Same open file (or pipe) as the original file.</li><li>– Same file position indicator as the original file (i.e. both file descriptors share one file position indicator).</li><li>– Same access mode (read, write, or read/write) as the original file.</li><li>– Same file status bits as the original file.</li><li>– The close-on-exec flag (see <code>F_GETFD</code>) associated with the new file descriptor is set to remain open across <code>exec</code> system calls.</li></ul> |
| <code>F_GETFD</code> | Gets the close-on-exec flag associated with file descriptor <i>fil-des</i> . If the low-order bit is 0, the file will remain open across <code>exec</code> . Otherwise, the file will be closed upon execution of <code>exec</code> .   |
| <code>F_SETFD</code> | Sets the close-on-exec flag associated with <i>fil-des</i> to the low-order bit of the integer value given as the third argument (0 or 1 as above).   |
| <code>F_GETFL</code> | Gets the <i>fil-des</i> status flag.  |
| <code>F_SETFL</code> | Sets the <i>fil-des</i> status flag to the integer value given as the third argument. Only certain flags can be set (see <code>fcntl()</code> ).  |

*Extension*

**F\_FREESP** Frees storage space associated with a section of the ordinary file *fil**des*. The section is specified by a variable of data type `struct flock` pointed to by the third argument *arg*. The data type `struct flock` is defined in the `fcntl.h` header file (see `fcntl()`) and contains the following members:

- `l_whence` is 0, 1 or 2 to indicate that the relative offset `l_start` will be measured from the start of the file, the current position, or the end of the file, respectively.
- `l_start` is the offset from the position specified in `l_whence`. `l_len` is the size of the section. An `l_len` of 0 frees up to the end of the file; in this case, the end of file (i.e., file size) is set to the beginning of the section freed. Any data previously written into this section is no longer accessible.

The following commands are used for file and record-locking. Locks may be placed on an entire file or on segments of a file.

**F\_SETLK** Set or clear a file segment lock according to the `flock` structure that *arg* points to (see `fcntl()`). The *cmd* `F_SETLK` is used to establish read (`F_RDLCK`) and write (`F_WRLCK`) locks, as well as remove either type of lock (`F_UNLCK`). If a read or write lock cannot be set, `fcntl()` will return immediately with an error value of -1.

**F\_SETLKW** This *cmd* is the same as `F_SETLK` except that if a read or write lock request is blocked by other locks, the process will wait until the segment is free to be locked.

**F\_GETLK** If the lock request described by the `flock` structure that *arg* points to could be created, then the structure is passed back unchanged except that the lock type is set to `F_UNLCK`, and the `l_whence` field will be set to `SEEK_SET`. If a lock is found that would prevent this lock from being created, then the structure is overwritten with a description of the first lock that is preventing such a lock from being created.

This command never creates a lock; it simply tests whether a particular lock could be created.

**F\_RSETLK**, **F\_RSETLKW**, **F\_RGETLK**

These commands are used by the network daemon `lockd` to lock NFS files with the NFS server.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock and no read locks may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The `flock` structure describes the type (`l_type`), starting offset (`l_whence`), relative offset (`l_start`), size (`l_len`), process ID (`l_pid`), and system ID (`l_sysid`) of the relevant segment of the file.

The value of `l_whence` is `SEEK_SET`, `SEEK_CUR` or `SEEK_END` to indicate that the relative offset `l_start` bytes will be measured from the start of the file, current position or end of the file, respectively. The value of `l_len` is the number of consecutive bytes to be locked. The value of `l_len` 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, i.e. one in which a lock was found, the value of `l_whence` will be `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 must not be negative relative to the beginning of the file. A lock will be set to extend to the largest possible value of the file offset for that file by setting `l_len` to 0. If such a lock also has `l_start` set to 0 and `l_whence` is set to `SEEK_SET`, the whole file will be locked.

There will be at most one type of lock set for each byte in the file. If the calling process already has existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region will be replaced by the new lock type before a successful return from an `F_SETLK` or an `F_SETLKW` request. As specified above under the descriptions of shared locks and exclusive locks, an `F_SETLK` or an `F_SETLKW` request will (respectively) fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

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 created using the `fork()` function.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process's locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, the `fcntl()` function will fail with an `EDEADLK` error.

When mandatory file and record locking is active on a file (see `chmod()`, `open()`, `read()` and `write()` system calls issued on the file will be affected by the record locks in effect.

The following additional value can be used when creating `oflag`:

`O_LARGEFILE` If this value is set, the offset maximum specified in the internal description of the open file is the highest value that can be properly represented in an object of type `off64_t`.

The `O_LARGEFILE` flag can be enabled and disabled with `F_SETFL`.

The response of the following values is the same as the response for `F_GETLK`, `F_SETLK`, `F_SETLKW` and `F_FREESP` except that an argument of type `struct flock64` must be passed instead of an argument of type `struct flock`:

`F_GETLK64`, `F_SETLK64`, `F_SETLKW64` and `F_FREESP64`

The `flock64` structure is defined like the `flock` structure (see `<fcntl( )>`) except for: `off64_t l_start` and `off64_t l_len`.

If threads are used, then the function affects the process or a thread in the following manner: When the `F_SETLKW` command is called, the thread waits until the request can be fulfilled.

## Return value

A new file descriptor

upon successful completion of the command `F_DUPFD`.

Value of process status flags, as defined in `fcntl.h`

upon successful completion of the command `F_GETFD`.

The return value will not be negative.

Value other than -1

upon successful completion of the commands `F_SETFD`, `F_SETFL`, `F_GETLK`, `F_SETLK` and `F_SETLKW`.

0

upon successful completion of the command `F_FREESP`

Value of file status flags and access modes

upon successful completion of the command `F_GETFL`.

The return value will not be negative.

-1

if an error occurs. `errno` is set to indicate the error.

## Errors

`fcntl( )` will fail if:

`EACCES`

`cmd` is `F_SETLK`, the type of lock (`l_type`) is a read lock (`F_RDLCK`), and the segment of a file to be locked is already write-locked by another process.

The type is a write lock (`F_WRLCK`) and the segment of a file to be locked is already read or write locked by another process.

EAGAIN	<i>cmd</i> is F_FREESP, the file exists, mandatory file/record locking is set, and there are outstanding record locks on the file.
<i>Extension</i>	
EAGAIN	<i>cmd</i> is F_SETLK or F_SETLKW, and the file is currently being mapped to virtual memory using <code>mmap()</code> (see <code>mmap()</code> ).
EBADF	<i>fildev</i> is not a valid open file descriptor.  <i>cmd</i> is F_SETLK or F_SETLKW, the type of lock ( <i>l_type</i> ) is a write lock (F_WRLCK), and <i>fildev</i> is not a valid file descriptor open for reading.  <i>cmd</i> is F_SETLK or F_SETLKW, the type of lock ( <i>l_type</i> ) is a write lock (F_WRLCK), and <i>fildev</i> is not a valid file descriptor open for writing.  <i>cmd</i> is F_FREESP, and <i>fildev</i> is not a valid file descriptor open for writing.
<i>Extension</i>	
EDEADLK	<i>cmd</i> is F_FREESP, mandatory record locking is enabled, O_NDELAY and O_NONBLOCK are clear, and a deadlock condition was detected.
EDEADLK	<i>cmd</i> is F_SETLKW, the lock is blocked by a lock from another process, and putting the calling process to sleep (i.e. in a wait state) until that lock becomes free would cause a deadlock situation.
<i>Extension</i>	
EFAULT	<i>cmd</i> is F_FREESP, and the value pointed to by <i>arg</i> is located at an address outside the address space used by the process.  <i>cmd</i> is F_GETLK, F_SETLK or F_SETLKW, and the value pointed to by <i>arg</i> is located at an address outside the address space used by the process.
EINTR	A signal was caught during the <code>fcntl()</code> system call.
EINVAL	<i>cmd</i> is F_DUPFD, and <i>arg</i> is either negative or greater than or equal to the value for the maximum number of open file descriptors permitted for each user.  <i>cmd</i> is not a valid value.  <i>cmd</i> is F_GETLK, F_SETLK or F_SETLKW, and <i>arg</i> or the data it points to is not valid, or <i>fildev</i> refers to a file that does not support locking.  An attempt was made to access a BS2000 file.
<i>Extension</i>	
EIO	An I/O error occurred while reading from or writing to the file system.
EMFILE	<i>cmd</i> is F_DUPFD, and the number of file descriptors currently open in the calling process is the configured value for the maximum number of open file descriptors allowed each user.



- ENOLCK        *cmd* is F\_SETLK or F\_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.
- ENOLINK      *files* is on a remote computer and the connection to this computer is not active or *cmd* is F\_FREESP, the file on a remote computer and the connection to it are not active.
- E\_OVERFLOW    One of the values returned cannot be represented correctly.

**Notes**        `fcntl()` is executed only for POSIX files

**See also**     `close()`, `creat()`, `dup()`, `exec()`, `fork()`, `open()`, `sigaction()`, `pipe()`, `fcntl.h`, `sys/type.h`, `unistd.h`.

## fcvt - convert floating-point number to string

Syntax        `#include <stdlib.h>`  
              `char *fcvt(double value, int ndigit, int *decpt, int *sign);`

Description See `ecvt()`.

## FD\_CLR, FD\_ISSET, FD\_SET, FD\_ZERO - macros for synchronous I/O multiplexing

Syntax        `#include <sys/time.h>`  
              `FD_CLR (int fd, fd_set *fdset);`  
              `FD_ISSET (int fd, fd_set *fdset);`  
              `FD_SET (int fd, fd_set *fdset);`  
              `FD_ZERO (fd_set *fdset);`

Description See `select()`.

## fdelrec - delete record in ISAM file *(BS2000)*

**Syntax**        `#include <stdio.h>`  
`int fdelrec(FILE *stream, void *key);`

**Description** `fdelrec()` deletes the record with the key value *key* from an ISAM file with record I/O. `FILE *stream` is the file pointer of an ISAM file that was opened in the mode `type=record, forg=key` (see also `fopen()`, `freopen()`). `void *key` is a pointer to an area which contains the key value of the record to be deleted in its complete length or null. If *key* is equal to null, the last record read is deleted. The record must be read immediately before the `fdelrec` call.

**Return value**

0	if successful. The record with the specified key was deleted.
> 0	The record to be deleted does not exist.
EOF	if an error occurs.

**Notes**

If the call was error-free (return values 0 or > 0) the EOF flag of the file is reset.

If the specified key value is not present in the file (return value > 0) the current position of the file position indicator remains unchanged. Sole exception: if, at the time of the `fdelrec` call, the file is positioned on the second or higher key of a group of records with identical keys, then `fdelrec()` positions the file on the first record after this group.

In ISAM files with key duplication, `fdelrec()` deletes the first record with the specified key. The file is then positioned on the next record (with the same key or the next higher key).

**See also**        `flocate()`, `fopen()`, `freopen()`, `stdio.h`.

## fdetach - cancel assignment to STREAMS file

Syntax      `#include <stropts.h>`  
`int fdetach(const char *path);`

Description The `fdetach()` function cancels the assignment of a file descriptor under STREAMS to a name in the file system. *path* is the pathname of the object (file or directory) in the name space of the file system to which the file descriptor was previously assigned with `fattach()`. The user must be the owner of the file or a user with special permissions.

A successful `fdetach()` call has the following effects: all pathnames that have identified the assigned STREAMS file then identify again the original object to which the STREAMS file was assigned. All subsequent operations on *path* work with the node in the file system and not with the STREAMS file.

The access permissions and the node status are restored as they were before the assignment.

All open file descriptors established while the STREAMS file was assigned to the file identified by *path* continue to refer to the STREAMS file after the `fdetach()` has taken effect.

If there are no open file descriptors or other references to the STREAMS file, a successful `fdetach()` has the effect on the assigned file of a final `close()` call on this file.

### Return value

0            if successful.  
-1           if an error occurs. `errno` is set to indicate the error.

### Errors

`fdetach()` will fail if:

EACCES      Search permission is denied for a component of the path.  
EPERM      The effective user ID of the process is not that of the owner of the file identified by *path* and the process does not have the appropriate access permissions.  
ENOTDIR    A component of the pathname prefix is not a directory.  
ENOENT     A component of the pathname does not exist, or *path* points to an empty string.  
EINVAL     *path* is not assigned to a STREAMS file.

ENAMETOOLONG

The length of *path* exceeds `{PATH_MAX}`, or a component of the pathname is longer than `{NAME_MAX}`, while `{_POSIX_NO_TRUNC}` is active.

ELOOP

Too many symbolic links were encountered in resolving *path*.

**See also** `close()`, `fattach()`, `stropts.h`.

## fdopen - associate stream with file descriptor

Syntax `#include <stdio.h>`  
`FILE *fdopen(int filides, const char *mode);`

Description `fdopen()` associates a stream with a file descriptor.  
*mode* a character string having one of the following values:

<code>r</code> or <code>rb</code>	open a file for reading
<code>w</code> or <code>wb</code>	open a file for writing
<code>a</code> or <code>ab</code>	open a file for writing at end of file
<code>r+</code> , <code>r+b</code> or <code>rb+</code>	open a file for update (reading and writing)
<code>w+</code> , <code>w+b</code> or <code>wb+</code>	open a file for update (reading and writing)
<code>a+</code> , <code>a+b</code> or <code>ab+</code>	open a file for update (reading and writing) at end of file

The meaning of these flags is exactly as specified in `fopen()`, except that *mode* arguments beginning with `w` do not cause the file to be truncated to length 0 (see `fopen()`).

The *mode* argument for the stream must only include the access modes that were originally defined for the file, i.e. `fdopen()` cannot be used to change the file access mode. The file position indicator associated with the stream is set to the same position as the file position indicator associated with the file descriptor.

The error and end-of-file indicators for the stream are cleared. The `fdopen()` function may cause the `st_atime` field of the underlying file to be marked for an update.

### BS2000

The `st_atime` field is ignored for BS2000 files. The file retains its original access mode. □

For automatic conversion, the `b` for binary must not be specified in *mode*. Furthermore, the environment variable `IO_CONVERSION` must not be present or must have the value `YES`.

Return value  
 Pointer to a stream

if successful.

Null pointer if an error occurs; `errno` is set to indicate the error.

Errors `fdopen()` will fail if:

- EBADF *fdes* is not a valid file descriptor.
- EINVAL For POSIX files: *mode* is not a valid mode.
- EMFILE {FOPEN\_MAX} streams are already open in the calling process.  
{STREAM\_MAX} streams are already open in the calling process.
- ENOMEM There is not enough memory to allocate a buffer.

*BS2000*

If errors occur, e.g. due to an invalid file descriptor, `fdopen()` will return neither a defined result nor an error message. The program does not abort in this case. □

Notes {STREAM\_MAX} is the number of streams that one process can have open at one time. If defined, it has the same value as {FOPEN\_MAX}, i.e. 2048.

File descriptors are obtained from calls like `open()`, `dup()`, `creat()` or `pipe()`.

The program environment determines whether `fdopen()` is executed for a BS2000 or POSIX file.

See also `fclose()`, `fopen()`, `open()`, `stdio.h`, section "File processing" on page 74.

## feof - test end-of-file indicator on stream

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

Description `feof()` tests the end-of-file indicator for the stream pointed to by *stream*.

Return value

<code>≠ 0</code>	EOF is set for <i>stream</i> ; the end of file was reached.
<code>0</code>	EOF is not set.

Notes        `feof()` is normally used after access functions that do not report end of file (`fread()`).  
If the file has been repositioned (e.g. with `fseek()`, `fsetpos()`, `rewind()`) after EOF has been reached, or if the `clearerr()` function has been called, `feof()` returns a value of 0.  
The program environment determines whether `feof()` is executed for a BS2000 or POSIX file.  
*BS2000*  
`feof()` is implemented both as a macro and as a function.  
`feof()` can also be used unchanged on files with record I/O. □

See also     `clearerr()`, `ferror()`, `fopen()`, `fseek()`, `fsetpos()`, `stdio.h`.



## ferror - test error indicator on stream

Syntax        `#include <stdio.h>`  
              `int ferror(FILE *stream);`

Description `ferror()` tests the error indicator for the stream pointed to by *stream*.

Return value

<code>≠ 0</code>	if the error indicator is set for <i>stream</i> .
<code>0</code>	if the error indicator is not set for <i>stream</i> .

Notes        The error indicator remains set until the associated file pointer is released (e.g. by a `rewind()`, `fclose()` or program termination) or until the `clearerr()` function is called.

The program environment determines whether `ferror()` is executed for a BS2000 or POSIX file.

### *BS2000*

`ferror()` is implemented both as a macro and as a function.

`ferror()` should always be used when reading from a file or writing to it.

`ferror()` can also be used unchanged on files with record I/O. □

See also     `clearerr()`, `feof()`, `fopen()`, `stdio.h`.

## fflush - flush stream

Syntax `#include <stdio.h>`

```
int fflush(FILE *stream);
```

Description If *stream* points to an output stream or an update stream in which the most recent operation was not input, `fflush()` causes any buffered data for that stream to be written to the file. If *stream* is a null pointer, the flushing action is performed on all open files.

Return value

0 if successful. The buffer was flushed.

EOF if an error occurs. The buffer was not flushed. `errno` is set to indicate the error.

*BS2000*

Alternatively, the buffer did not need to be flushed, since it does not exist (because no write function has been executed on the file), or the file is an input or INCORE file. □

*stream* is not associated with any file (since the file is already closed, for example) or the buffered data could not be transferred.

Errors `fflush()` will fail if:

EAGAIN The `O_NONBLOCK` flag is set for the file descriptor underlying *stream*, and the process would be delayed in the write operation.

EBADF The file descriptor underlying *stream* is not valid.

EFBIG An attempt was made to write a file that exceeds the maximum file size or the process file size limit (see also `ulimit()`).

EINTR `fflush()` was interrupted by a signal.

EIO An I/O error occurred.

The process is a member of a background process group attempting to write to its controlling terminal; `TOSTOP` is set; the process is neither ignoring nor blocking `SIGTTOU`, and the process group of the process is orphaned.

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. A `SIGPIPE` signal will also be sent to the process.

If threads are used, then the function affects the process or a thread in the following manner: If an `EPIPE` error occurs, the `SIGPIPE` signal is not sent to the process, but is sent to the calling thread instead.

Notes The program environment determines whether `fflush()` is executed for a BS2000 or POSIX file.

#### *BS2000*

All standard I/O functions that write data to a BS2000 file (`printf()`, `putc()`, `fwrite()`, etc.) store this data temporarily in a buffer and only write it to the file when one of the following events occurs:

- A newline character (`\n`) is detected (only for text files).
- The maximum record length of a disk file is reached.
- For terminals: when output to the terminal is followed by input from the terminal.
- The functions `fseek()`, `fsetpos()`, `rewind()` or `fflush()` are called.
- The file is closed.

In addition, for ANSI functionality only:

If reading from any text file makes data transfer necessary from the external file to the buffer, the data of all ISAM files still stored in buffers is automatically written out to the files.

Buffering does not take place in the case of outputs to strings (`sprintf()`) and to INCORE files.

`fflush()` causes a line change in a text file even if the data in the buffer does not end with a newline character. Data that follows is written to a new line (or a new record).

Exception for ANSI functionality:

If the data of an ISAM file in the buffer does not end in a newline character, `fflush()` does not cause a change of line (or change of record). Subsequent data extends the record in the file. Consequently, when an ISAM file is read, only those newline characters explicitly written by the program are read in.

`fflush()` is automatically executed internally when a file is closed (`fclose()`, `close()`) or when a program ends normally or is terminated by means of an `exit()`.

`fflush()` can be used to control the output of data during program execution, e.g. to concatenate various inputs into a single output and print them together at a user-defined point in time.

In the case of record I/O, calls to the `fflush()` function are not rejected with an error, but have no effect. No data is buffered for files with record I/O. □

See also `exit()`, `close()`, `fclose()`, `stdio.h`.

## ffs - seek first set bit

**Syntax**      `#include <strings.h>`  
                 `int ffs(int i);`

**Description** `ffs()` searches for the first set bit in the transferred argument, beginning with the least-significant bit, and returns the position of this bit. The numbering of the bits begins with 1, starting with the least-significant bit.

**Return value**  
                 Position of the first set bit  
                                 if  $i \neq 0$ .  
  
                 0                    if  $i = 0$ .

**See also**      `strings.h`.

## fgetc - get byte from stream

**Syntax**        `#include <stdio.h>`  
`int fgetc(FILE *stream);`

**Description** `fgetc()` reads the next existing byte of type `unsigned char` from the input stream pointed to by *stream*, converts it to an `int`, and advances the associated file position indicator for the stream (if defined).

`fgetc()` can mark the structure component `st_atime` for the file to which *stream* is assigned for changing (see `sys/stat.h`). The structure component `st_atime` is updated as soon as `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` are called successfully for *stream* and return data which is not was not provided by a preceding call to `ungetc()` or `ungetwc()`.

### Return value

Next byte from the input stream pointed to by *stream* upon successful completion.

EOF            if the stream is at end-of-file. The end-of-file indicator for the stream is set.

EOF            if a read error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

**Errors**        `fgetc()` will fail if:

EAGAIN        The `O_NONBLOCK` flag is set for the file descriptor underlying *stream*, and the process would be delayed in the `fgetc()` operation.

EBADF         The file descriptor underlying *stream* is not a valid file descriptor open for reading.

EINTR         The read operation was terminated due to the receipt of a signal, and no data was transferred.

EIO            A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group is orphaned.

## Notes

If the integer value returned by `fgetc()` is stored into a variable of type `char` and then compared against the integer constant `EOF`, the comparison may never succeed, because sign-extension of a variable of type `char` on widening to integer is machine-dependent. Portable applications should therefore always use an `int` variable for the result of `fgetc()`.

The `ferror()` or `feof()` functions must be used to distinguish between an error condition and an end-of-file condition.

If a comparison such as:

```
while((c = fgetc(dz)) != EOF)
```

is used in a program, the variable `c` must always be declared as an `int` value.

Otherwise, if `c` were defined as a `char`, the `EOF` condition would never be satisfied for the following reason: `-1` is converted to the `char` value `0xFF` (i.e. `+255`); however, `EOF` is defined as `-1`.

If `fgetc()` is reading from the standard input `stdin` in the POSIX environment, and `EOF` is the end criterion for reading, the `EOF` condition can be achieved by the following actions:

- ▶ on a block-special terminal: by entering the key sequence `@ @ d`
- ▶ on a character-special terminal: by entering `CTRL+D`

#### *BS2000*

If `fgetc()` is reading from the standard input `stdin` in the BS2000 environment, and `EOF` is the end criterion for reading, the `EOF` condition can be achieved by means of the following actions at the terminal:

1. by pressing the `K2` key.
2. by entering the system commands `EOF` and `RESUME-PROGRAM`. □

The program environment determines whether `fgetc()` is executed for a BS2000 or POSIX file.

See also `feof()`, `ferror()`, `fopen()`, `getchar`, `getc()`, `stdio.h`, `sys/stat.h`.

## fgetpos - get current value of file position indicator in stream

**Name** fgetpos, fgetpos64

**Syntax** #include <stdio.h>

```
int fgetpos(FILE *stream, fpos_t *pos);
int fgetpos64(FILE *stream, fpos64_t *pos);
```

**Description** fgetpos() stores the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored contains information usable by fsetpos() for repositioning the stream to its position at the time of the call to fgetpos(). There is no difference in functionality between fgetpos() and fgetpos64() except that fgetpos64() uses a fpos64\_t data type.

**Return value**

0 if successful.

≠ 0 if an error occurs; errno is set to indicate the error.

*BS2000*  
errno is set to EBADF.

**Errors** fgetpos() will fail if:

EBADF The file descriptor underlying *stream* is not valid.

ESPIPE The file descriptor underlying *stream* is associated with a pipe or FIFO.

**Notes** The program environment determines whether fgetpos() is executed for a BS2000 or POSIX file.

*BS2000*

fgetpos() can be used on binary files (SAM in binary mode, PAM, INCORE) and text files (SAM in text mode, ISAM). fgetpos() cannot be used on system files (SYSDTA, SYSLST, SYSOUT).

For ISAM files, the function pair fgetpos()/fsetpos() is far more effective than the comparable function pair ftell()/fseek().

For record I/O, fgetpos() returns the position after the last record to be read, written or deleted or the position reached by an immediately preceding positioning operation.

For ISAM files with key duplication, fgetpos() always returns the position after the last record of a group with identical keys if one of these records has previously been read, written or deleted.

**See also** fseek(), fseek64(), lseek(), lseek64(), fsetpos(), fsetpos64(), ftell(), ftell64(), ungetc(), stdio.h.

## fgets - get string from stream

**Syntax**      `#include <stdio.h>`  
`char *fgets(char *s, int n, FILE *stream);`

**Description** `fgets()` reads at most  $n-1$  bytes from the stream pointed to by *stream* until a newline character or an end-of-file condition is encountered. The string is read into the array pointed to by *s* and terminated with a null byte.

`fgets()` can mark the structure component `st_atime` for the file to which *stream* is assigned for changing (see `sys/stat.h`). The structure component `st_atime` is updated as soon as `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` are called successfully for *stream* and return data which is not was not provided by a preceding call to `ungetc()` or `ungetwc()`.

### Return value

Pointer to the result string  
upon successful completion.

Null pointer      if the stream is at end-of-file. The end-of-file indicator for the stream is set.

Null pointer      if a read error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

**Errors**      See `fgetc()`.

**Notes**      The area in which `fgets()` is to store the string that is read must be supplied explicitly. In contrast to `gets()`, `fgets()` also enters a newline character (if read) into the result string.  
The program environment determines whether `fgets()` is executed for a BS2000 or POSIX file.

**Example**      See `fputs()`.

**See also**      `fgetc()`, `fopen()`, `fputs()`, `fread()`, `gets()`, `stdio.h`, `sys/stat.h`.



## fgetwc - get wide character string from stream

Syntax `#include <wchar.h>`

*Optional*

`#include <stdio.h>` □

`wint_t fgetwc(FILE *stream);`

Description `fgetwc()` reads the next character (if present) from the input stream pointed to by *stream*, converts that to the corresponding wide character code and advances the file position indicator for the stream (if defined).

If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

`fgetwc()` can mark the structure component `st_atime` for the file to which *stream* is assigned for changing (see `sys/stat.h`). The structure component `st_atime` is updated as soon as `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` are called successfully for *stream* and return data which is not was not provided by a preceding call to `ungetc()` or `ungetwc()`.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`).

Return value

Wide character code of type `wint_t` upon successful completion.

WEOF if the stream is at end-of-file. The end-of-file indicator for the stream is set.

WEOF if a read error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

Errors

`fgetwc()` will fail if:

EAGAIN The `O_NONBLOCK` flag is set for the file descriptor underlying *stream*, and the process would be delayed in the `fgetc()` operation.

EBADF The file descriptor underlying *stream* is not a valid file descriptor open for reading.

EINTR The read operation was terminated due to the receipt of a signal, and no data was transferred.

*Extension*

EINVAL An attempt was made to access a BS2000 file □

EIO The process is a member in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group is orphaned.

**Notes** In this version of the runtime system the wide character functions are only supported for UFS files.

`ferror()` or `feof()` must be used to distinguish between an error condition and an end-of-file condition.

**See also** `feof()`, `ferror()`, `fgetc()`, `fopen()`, `stdio.h`, `wchar.h`.

## fgetws - get wide character string from stream

Syntax `#include <wchar.h>`

*Optional*

`#include <stdio.h>` □

`wchar_t *fgetws(wchar_t *ws, int n, FILE *stream);`

Description `fgetws()` reads characters from *stream*, converts these to the corresponding wide character codes, and places them in the `wchar_t` array pointed to by *ws*, until *n-1* characters are read, or a newline character is read, converted and transferred to *ws*, or an end-of-file condition is encountered. The wide character string, *ws*, is then terminated with a null wide-character code.

If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

`fgetws()` can mark the structure component `st_atime` for the file to which *stream* is assigned for changing (see `sys/stat.h`). The structure component `st_atime` is updated as soon as `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` are called successfully for *stream* and return data which is not was not provided by a preceding call to `ungetc()` or `ungetwc()`.

Return value

*ws* upon successful completion.

Null pointer if the stream is at end-of-file. The end-of-file indicator for the stream is set.

Null pointer if a read error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

Errors See `fgetwc()`.

See also `fgetwc()`, `fopen()`, `fread()`, `stdio.h`, `wchar.h`.

## \_\_FILE\_\_ - macro for source file names

Syntax \_\_FILE\_\_

Description This macro generates the file name of the source program as a string in the form:

```
"name\0"
```

Notes This macro does not need to be defined in a header file. Its name is recognized and replaced by the compiler.

## fileno - get file descriptor

Syntax 

```
#include <stdio.h>
int fileno(FILE *stream);
```

Description `fileno()` returns the integer file descriptor associated with the stream pointed to by *stream*.

Return value

`int value` if successful. Value of the file descriptor associated with *stream*.  
`-1` if an error occurs; `errno` is set to indicate the error.

Errors `fileno()` will fail if:

`EABDF` *stream* is not a valid stream.

Notes The program environment determines whether `fileno()` is executed for a BS2000 or POSIX file.

See also `fdopen()`, `fopen()`, `stdin()`, `stdio.h`, section "Interaction of file descriptors and streams" on page 79.

**flocate - set file position indicator in ISAM file** (BS2000)

Syntax `#include <stdio.h>`

```
int flocate(FILE *stream, void *key, size_t keylen, int option);
```

Description `flocate()` is used to explicitly position an ISAM file with record I/O. `flocate()` sets the file position indicator of the file pointed to by `stream` according to the following specifications:  
 the key value `key`,  
 the key length `keylen` and  
 the specified `option` (`_KEY_FIRST`, `_KEY_LAST`, `_KEY_EQ`, `_KEY_GE`).

`FILE *stream` is the file pointer of an ISAM file opened in the mode `type=record, forg=key` (see `fopen()`, `freopen()`).

`void *key` is the pointer to an area containing the key value.

`size_t keylen` is the length of the key value. The value must not be zero.

If `keylen` is less than the key length of the file, then `flocate()` internally pads the key value with binary zeros to the key length of the file and uses this generated key as the basis for positioning.

If `keylen` is greater than the key length of the file, `flocate()` internally truncates the key value from the right to the key length of the file and uses this shortened key as the basis for positioning.

`int option` may contain the following values defined in `stdio.h`:

<code>_KEY_FIRST</code>	Sets the file position indicator to beginning of file. The <code>key</code> and <code>keylen</code> parameters are ignored. Positioning works even if the file is empty.
<code>_KEY_LAST</code>	Sets the file position indicator to end of file. The <code>key</code> and <code>keylen</code> parameters are ignored. Positioning works even if the file is empty.
<code>_KEY_EQ</code>	Sets the file position indicator to the first record with the specified <code>key</code> .
<code>_KEY_GE</code>	Sets the file position indicator to the first record with a key value greater than or equal to the specified <code>key</code> .

Return value

0	if successful. The record with the specified key exists.
> 0	The record does not exist.
EOF	if an error occurs.

- Notes**      If the call was error-free (return values 0 or > 0), the EOF flag of the file is reset.
- If the specified key value is not present in the file (return value > 0), the current setting of the file position indicator remains unchanged. Sole exception: if at the time of the `flocate` call the file is positioned on the second or higher key of a group of records with identical keys, then `flocate()` positions the file on the first record after this group.
- In ISAM files with key duplication, `flocate()` cannot be used to position on the second or higher record of a group with identical keys. This can only be done by sequential reading or deleting.
- `flocate()` can only be used to position on the first record or after the last record of such a group.
- See also**    `fdelrec()`, `fgetpos()`, `fsetpos()`, `fopen()`, `freopen()`, `stdio.h`.

## flockfile, ftrylockfile, funlockfile - functions for locking standard input/output

Syntax

```
#include <stdio.h>

void flockfile(FILE *file);

int ftrylockfile(FILE *file);

void funlockfile(FILE *file);
```

Description The `flockfile()` and `ftrylockfile()` functions allow for the explicit locking of (FILE\*) objects at the application level. The lock can be eliminated with `funlockfile()`. These functions can be used by a thread to represent a series of I/O statements that are to be executed as a unit.

The `flockfile()` function is used by a thread to obtain access permission for a (FILE\*) object.

The `ftrylockfile()` function is used by a thread to obtain access permission for a (FILE\*) object if the object is available; `ftrylockfile()` is a version of `flockfile()` that does not block the object.

The `funlockfile()` function is used by a thread to give up the access permission it obtained. `funlockfile()` is ignored if the calling thread is not the owner of the (FILE\*) object.

It is logical to assign every (FILE\*) object a lock counter. This counter is implicitly initialized to 0 when the (FILE\*) object is created. The lock for the (FILE\*) object is removed when the counter has the value 0.

When the counter value is positive, then a single thread is the owner of the (FILE\*) object. If the `flockfile()` function is called when the counter is 0 or contains a positive value and the caller is the owner of the (FILE\*) object then the counter is incremented. Otherwise the calling thread is interrupted and waits until the counter is 0 again. Every `funlockfile()` call decrements the counter. This allows for nested `flockfile()` calls [or successful `ftrylockfile()` calls] and `funlockfile()` calls.

All functions that point to (FILE\*) objects behave as if they used `flockfile()` and `funlockfile()` to obtain access permission for these (FILE\*) objects.

### Return value

`flockfile()` and `funlockfile()`:  
no return value

`ftrylock()`:

0 if successful.

≠0 if no lock can be activated.

**Notes** In real time applications the use of FILE locks can result in the reversal of priorities. This problem arises when a thread with higher priority “locks” a FILE object that was just “unlocked” by a thread of lower priority, but the thread of lower priority is prematurely stopped by a thread of medium priority. This situation leads to a reversal of the priorities; a thread of higher priority is blocked by a thread of lower priority for an indefinite amount of time.

Developers of real time applications must take the possibility of such reversals of priority into account when designing a system. They could take a series of actions to counteract such situations by having critical sections of code that are protected by FILE locks execute with a higher priority so that a thread cannot be stopped prematurely while executing a critical sections of code.

**See also** `getc_unlocked()`, `pthread_intro()`, `stdio()`.



## floor, floorf, floorl- round off floating point number

**Syntax**        `#include <math.h>`  
                 `double floor(double x);`  
                 `float floorf(float x)`  
                 `long double floorl(long double)`

**Description** `floor()` rounds down the floating-point number  $x$  to an integer.

**Return value**  
If successful, largest integer of type `double` not greater than  $x$ .  
`-HUGE_VAL`        if an overflow occurs.  
                     `errno` is set to indicate the error.

**Errors**        `floor()`, `floorf()`, and `floorl()` will fail if:  
`ERANGE`        Overflow; the result is too large.

**Notes**        The integral value returned by `floor()`, `floorf()`, or `floorl()` as a `double`, `float` or `long` might not be expressible as an `int` or `long int`. The return value should be tested before assigning it to an integer type to avoid the undefined results of an integer overflow.

**See also**     `ceil()`, `ceilf()`, `ceill()`, `fabs()`, `math.h`.

## fmod - compute floating-point remainder value function

**Syntax**      `#include <math.h>`  
`double fmod(double x, double y);`

**Description** `fmod()` computes the remainder of the division  $x/y$ . The remainder has the same sign as the dividend  $x$ , and its absolute value is always less than the divisor  $y$ .

**Return value**  
Remainder of the division  $x/y$   
if successful.  
0              if  $y = 0$ .

**Notes**        An application should verify that  $y$  is non-zero before calling `fmod()`.

**See also**     `ceil()`, `ceilf()`, `ceil11()`, `fabs()`, `floor()`, `math.h`.

## fmtmsg - output message to stderr and/or system console

**Syntax**      `#include <fmtmsg.h>`

```
int fmtmsg(long classification, const char *label, int severity, const char *text,
           const char *action, const char *tag);
```

**Description** Building on the classification component of a message, `fmtmsg()` writes a formatted message to `stderr`, the system console or both.

`fmtmsg()` can be used instead of the usual `printf()` interface to output messages via `stderr`. In conjunction with `gettext()`, `fmtmsg()` provides a simple interface for the creation of language-independent application programs.

A formatted message consists of up to five standard components which are defined below. The *classification* component is not part of the standard message that is shown to the user; instead, it defines the message source and controls the display of the formatted message.

### *classification*

contains identifiers from the following groups of main and secondary classifications. Every identifier of a subclass can be used with a single identifier of a different subclass via inclusive OR. With the exception of the display classification, two or more identifiers from the same subclass should not be used together. Both identifiers of the display classification can be used such that the messages appear on both `stderr` and the system console.

### Major classifications

identify the origin of a status. The identifiers are: `MM_HARD` (hardware), `MM_SOFT` (software) and `MM_FIRM` (firmware).

### Message source subclassifications

identify the type of software in which the problem occurred. The identifiers are: `MM_APPL` (application), `MM_UTIL` (utility routine) and `MM_OPSYS` (operating system).

### Display subclassifications

identify where the message is to be displayed. The identifiers are `MM_PRINT` for outputting the message to standard error output, and `MM_CONSOLE` for outputting the message to the system console. You can use one or both identifiers or you can omit the specification (in the latter case, nothing is output).

### Status subclassifications

indicate whether the application program can recover after the status. Identifiers are: `MM_RECOVER` (recoverable) and `MM_NRECOV` (non-recoverable).

	Additional identifier <code>MM_NULLMC</code> indicates that no classification component is specified for the message.
<i>label</i>	defines the origin of the message. The format of this component consists of two fields separated by a colon. The first field is up to 10 characters long; the second is up to 14 characters long.  It is advisable to mark the package and the program or the application name with <i>label</i> . For example, the content <code>UX:cat</code> for <i>label</i> indicates that the package UNIX system V and the application <code>cat</code> are meant.
<i>severity</i>	indicates the severity level of the status. Identifiers for the severity levels are:  <code>MM_HALT</code> indicates that the application has come across a critical error and processing is being halted. The string "HALT" is output.  <code>MM_ERROR</code> indicates that the application has detected an error. The string "ERROR" is output.  <code>MM_WARNING</code> indicates that an unusual state has arisen which could involve a problem that needs monitoring. The string "WARNING" is issued.  <code>MM_INFO</code> provides information on a state which does not represent an error. The string "INFO" is output.  <code>MM_NOSEV</code> indicates that no severity level exists for the message.
<i>text</i>	describes the cause of the message. The <i>text</i> string is not limited to a particular length. If the string is empty, the text that is output is undefined.
<i>action</i>	describes the first action to be executed in the error recovery process. <code>fmtmsg()</code> writes the prefix "TO FIX:" before this string. The <i>action</i> string is not limited to a particular length.
<i>tag</i>	An identifier that refers to the online documentation for the message. It is recommended that <i>tag</i> contain the origin of the message addressed via <i>label</i> and a unique number. An example of <i>tag</i> is <code>UX:cat:146</code> .

## Environment variables

There are two environment variables which influence the behavior of `fmtmsg()`: `MSGVERB` and `SEV_LEVEL`.

`MSGVERB` informs `fmtmsg()` which message components are to be selected when writing the messages to `stderr`. The value of `MSGVERB` consists of a list of optional keywords separated by colons. `MSGVERB` can be set as follows:

```
MSGVERB=[keyword[:keyword[:...]]]
export MSGVERB
```

Valid *keywords* are: `label`, `severity`, `text`, `action` and `tag`.

If `MSGVERB` contains a keyword for a component and this component does not have the null value assigned to it (see below), `fmtmsg()` outputs this component to `stderr` at message output. If `MSGVERB` does not contain the keyword for a message component, this component is not output. The keywords can be specified in any order. If `MSGVERB` is not defined, if this identifier contains a null string, if the value is not specified in the correct format, or if invalid keywords are specified, `fmtmsg()` selects all components.

At the first call of `fmtmsg()` the `MSGVERB` environment variable is verified so that the message components can be selected if a message is generated via the standard error output `stderr`. The values accepted at the first call are saved for the subsequent calls.

`MSGVERB` influences only the selection of the components that are to be displayed via the standard error output. In the case of output to the console, all messages are selected.

`SEV_LEVEL` defines the severity levels and assigns the strings to be output that are to be used by `fmtmsg()`. The standard severity levels given below cannot be changed. Additional severity levels can be defined, modified and deleted via the `addseverity()` function (see `addseverity(3C)`). If the same severity level is defined by `SEV_LEVEL` and `addseverity()`, the `addseverity()` definition takes precedence.

```
0  (no severity level used)
1  HALT
2  ERROR
3  WARNING
4  INFO
```

`SEV_LEVEL` can be set as follows:

```
SEV_LEVEL=[description[:description[:...]]]
export SEV_LEVEL
```

*description* contains a list with three fields, each separated by a comma:

```
description=severity_keyword,level,printstrin
```

*severity\_keyword* is a string that is used as the keyword for the option `-s severity` of the `fmtmsg` command. This field is not used by the `fmtmsg()` function.

*level* is a string containing a positive integer (not 0, 1, 2, 3 or 4 because these values are reserved for the standard severity levels). If the keyword *severity\_keyword* is used, *level* represents the severity level of the value that was passed to the `fmtmsg()` function.

*printstring* is a string that is used by `fmtmsg()` for the standard message format when the severity level *level* is specified.

If in the list *description* does not represent a list of three fields separated by commas, or if the second field of a list is not an integer, *description* is ignored in the list.

When `fmtmsg()` is called for the first time, the `SEV_LEVEL` environment variable is checked to see whether, in addition to the five standard severity levels and those defined via `addseverity()`, any other severity levels were defined. The values established at the first call are saved for later calls.

#### Return value

<code>MM_OK</code>	if successful.
<code>MM_NOTOK</code>	The function has completely failed.
<code>MM_NOMSG</code>	The function could not generate a message via the standard error output, but was otherwise successful.
<code>MM_NOCON</code>	The function could not generate a message via the system console, but was otherwise successful.

#### Notes

One or more message components can be systematically omitted from the message if the null value of the respective components is specified.

The following table shows the null values and identifiers for the arguments of `fmtmsg()`.

Argument	Type	Null value	Identifier
<i>label</i>	char*	(char*) NULL	MM_NULLLBL
<i>severity</i>	int	0	MM_NULLSEV
<i>class</i>	long	0L	MM_NULLMC
<i>text</i>	char*	(char*) NULL	MM_NULLTXT
<i>action</i>	char*	(char*) NULL	MM_NULLACT
<i>tag</i>	char*	(char*) NULL	MM_NULLTAG

A further means of systematic omission of a component consists of leaving out the keywords of the component when defining the `MSGVERB` environment variable.

**Example 1** `fmtmsg(MM_PRINT, "UX:cat", MM_ERROR, "Incorrect syntax",  
"See manual", "UX:cat:001")`

returns a complete message with the standard message format:

```
UX:cat: ERROR: Incorrect syntax TO FIX: See manual UX:cat:001
```

**Example 2** If the `MSGVERB` environment variable is set as follows:

```
MSGVERB=severity:text:action
```

and example 1 is then used, `fmtmsg()` generates:

```
ERROR: Incorrect syntax TO FIX: See manual
```

**Example 3** If the `SEV_LEVEL` environment variable is set as follows:

```
SEV_LEVEL=note,5,NOTE
```

the following `fmtmsg()` call

```
fmtmsg(MM_UTIL | MM_PRINT, "UX:cat", 5, "Incorrect syntax",  
"See manual", "UX:cat:001")
```

returns the following output:

```
UX:cat: NOTE: Incorrect syntax TO FIX: See manual UX:cat:001
```

**See also** `printf()`. `fmtmsg.h`.

## fopen - open stream

**Name**        **fopen, fopen64**

**Syntax**      `#include <stdio.h>`

```
FILE *fopen(const char *filename, const char *mode);
FILE *fopen64(const char *filename, const char *mode);
```

**Description** `fopen()` opens the file whose pathname is the string pointed to by *filename*, and associates a stream with it.

*filename* can be:

- a valid POSIX file name
- a valid BS2000 file name:
  - `link=linkname`  
*linkname* designates a BS2000 link name.
  - (SYSDTA), (SYSOUT), (SYSLST), the corresponding system file
  - (SYSTEM), terminal I/O
  - (INCORE), temporary binary file that is created in virtual memory only.

*mode* is a string that specifies the desired access mode. It can have one of the following values:

r	Open text file for reading. The file must already exist.
w	Open text file for writing. If the file exists, the old contents are deleted. If the file does not exist, it is created.
a	Open text file for appending to the end of the file. If the file exists, it is positioned to end of file, i.e. the old contents are preserved and the new data is appended to the end of the file. If the file does not exist, it is created.
rb	Open binary file for reading. The file must already exist.
wb	Open binary file for writing. If the file exists, the old contents are deleted. If the file does not exist, it is created.
ab	Open binary file for appending to the end of the file. If the file exists, it is positioned to end of file, i.e. the old contents are preserved and the new data is appended to the end of the file. If the file does not exist, it is created.
r+w, r+	Open text file for reading and writing. The file must already exist. The old contents are preserved.



<code>w+r, w+</code>	Open text file for writing and reading. If the file exists, the old contents are deleted. If the file does not exist, it is created.
<code>a+r, a+</code>	Open text file for appending to the end of the file and for reading. If the file exists, it is positioned to end of file, i.e. the old contents are preserved, and new data is appended to the end of the file. For KR functionality (only available with C/C++ versions lower than V3), existing files are positioned to end of file when opened; for ANSI functionality, to the beginning of the file. If the file does not exist, it is created.
<code>r+b, rb+</code>	Open binary file for reading and writing. The file must already exist. The old contents are preserved.
<code>w+b, wb+</code>	Open binary file for writing and reading. If the file exists, the old contents are deleted. If the file does not exist, it is created.
<code>a+b, ab+</code>	Open binary file for appending to the end of the file and for reading. If the file exists, it is positioned to end of file, i.e. the old contents are preserved and the new data is appended to the end of the file. For KR functionality (only available with C/C++ versions lower than V3), existing files are positioned to end of file when opened; for ANSI functionality, to the beginning of the file. If the file does not exist, it is created.

The character `b` in the above access modes is ignored. Opening a file with read mode (i.e. with `r` as the first character in the *mode* argument) fails if the file does not exist or cannot be read.

Opening a file with append mode (i.e. with `a` as the first character in the *mode* argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to `fseek()`.

When a file is opened with update mode (i.e. with `+` as the second character in the *mode* argument), both input and output may be performed on the associated stream. However, output must not be directly followed by input without an intervening call to `fflush()` or to a file positioning function (`fseek()`, `fsetpos()` or `rewind()`), and input must not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device such as the terminal. The error and end-of-file indicators for the stream are cleared.

For automatic conversion, the `b` for binary must not be specified in *mode*. Furthermore, the environment variable `IO_CONVERSION` must not be present or must have the value `YES`.

*BS2000*

The following must be noted when executing BS2000 files:

Besides the access mode, an optional entry to control handling of the tab character (`\t`) may be specified in *mode*. This is relevant only for text files with the SAM and ISAM access methods.

```
" . . . , tabexp=yes "
```

The tab character is expanded into the appropriate number of blanks. This is the default setting for KR functionality (only available with C/C++ versions lower than V3).

```
" . . . , tabexp=no "
```

The tab character is not expanded. This is the default setting for ANSI functionality.

There is no difference in functionality between `fopen` and `fopen64` except that `fopen64` returns a pointer that can point past the 2GB limit. `fopen64()` sets the `O_LARGEFILE` bit in the File status flag.

## Return value

File pointer      if successful.

Null pointer      if *filename* cannot be accessed, *mode* is invalid, or the file cannot be opened.  
`errno` is set to indicate the error.

## Errors

`fopen()` will fail if:

`EACCES`          Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

`EINTR`            A signal was caught during the `fopen()` system call.

`EINVAL`           The value of the *mode* argument is invalid.

`EISDIR`           The named file is a directory and *mode* requires write access.

`EMFILE`           { `OPEN_MAX` } file descriptors are already open for the calling process.  
                      { `FOPEN_MAX` } streams are already open for the calling process.  
                      { `STREAM_MAX` } streams are already open for the calling process.

`ENAMETOOLONG`    The length of *filename* exceeds { `PATH_MAX` } or a pathname component is longer than { `NAME_MAX` } .

`ENFILE`           The maximum allowable number of files is currently open in the system.

`ENOENT`           The named file does not exist or *filename* points to an empty string.

`ENOMEM`           There is not enough memory available.

ENOSPC	The file does not exist, and the directory in which the new file is to be created cannot be expanded.
ENOTDIR	A component of the path is not a directory.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist.
EROFS	The named file resides on a read-only file system and <i>mode</i> requires write access.
ETXTBSY	The file is a pure procedure file (shared text file) that is currently executing and write protection is required for <i>mode</i> .
EOVERFLOW	The file named is a regular file and the size of the file cannot be represented correctly in an object of type <code>off_t</code> .

**Notes** `{STREAM_MAX}` is the number of streams that one process can have open at one time. If defined, it has the same value as `{FOPEN_MAX}`, i.e. 2048.

The program environment determines whether `fopen()` is executed for a BS2000 or POSIX file.

*BS2000*

The BS2000 file name or link name may be entered in lowercase and uppercase letters. It is automatically converted to uppercase letters. Specifying a `b` as the second character in the *mode* parameter causes the file to be opened as a binary file. This is relevant only for SAM files, since only SAM files can be processed in both binary and text modes.

System files and ISAM files are always processed as text files. Specifying binary mode for these files leads to an error on opening.

(INCORE) and PAM files are always processed as binary files. For compatibility reasons, files may be opened as binary files without explicitly specifying the binary mode.

When a new file is created it is given the following attributes by default:

	Binary file	Text file
Access method	SAM	SAM (KR functionality, only available with C/C++ versions lower than V3) ISAM (ANSI functionality)
Record format	F	V

The following file attributes can be changed by using a link name with the `SET-FILE-LINK` command: access method, record length, record format, block length and block format.

Whenever the old contents of an existing file are deleted (i.e. when a file is opened for rewriting or for rewriting and reading), the catalog attributes of that file are preserved.

When a file is opened for an update, reading and writing can be performed via the same file pointer. All the same, an output should not be immediately followed by an input without a preceding positioning operation (with `fseek()`, `fsetpos()` or `rewind()`) or an `fflush` call. This also applies to an output that follows an input.

### Set the file position indicator in append mode

(INCORE) files can only be opened for writing (`w`), for writing and reading (`w+r`) or for reading (`r`). Data must first be written. The following options are available to read in the written data: if the file was opened only for writing, it can be opened for reading with the function `freopen()`. If it was opened for writing and reading, the file position indicator can be set to the beginning of the file with `rewind()`.

A file may be opened for different access modes simultaneously, provided these modes are compatible with one another within the BS2000 data management system.

When a program begins, the following three file pointers are assigned to it automatically:

<code>stdin</code>	file pointer for standard input (terminal)
<code>stdout</code>	file pointer for standard output (terminal)
<code>stderr</code>	file pointer for standard error output (terminal)

A maximum of `_NFILE` files may be open simultaneously. `_NFILE` is defined as 2048 in `stdio.h`.

For opening files with record I/O, the *mode* parameter has two additional options. These follow the access mode in the string (see above), each separated by a comma:

```
"...,type=record [,forg={seq/key}]"
```

<code>type=record</code>	The file is opened for record I/O. If this option is omitted, the file is opened for stream I/O.
<code>forg=seq</code>	The file is organized sequentially. Sequential files may be SAM or PAM files.
<code>forg=key</code>	The file is organized index-sequentially. Index-sequential files are ISAM files.

If `forg()` is omitted, the file organization depends on the FCB type (FCBTYP) of the file: The FCB type is defined by the catalog entry of an existing file or by a `SET-FILE-LINK` command. Sequential organization is assumed for SAM and PAM files, index-sequential organization for ISAM files.

If `forg()` is omitted and the FCB type is not defined (file does not exist, no `SET-FILE-LINK` command), sequential file organization is assumed, and a SAM file is created.

The following restrictions apply to record I/O. If these restrictions are ignored, the file is not opened, and an error value is returned:

The file must be opened in binary mode (b specified in the access mode).

type=record is permitted for SAM, PAM and ISAM files.

forg=seq is permitted for SAM and PAM files; forg=key for ISAM files.

The append mode a is invalid for ISAM files. The position is determined by the key in the record.

**See also** `creat()`, `fclose()`, `fdopen()`, `ferror()`, `freopen()`, `open()`, `stdio.h`.

## fork - create new process

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h> □`

`pid_t fork(void);`

Description `fork()` creates a new process. The new process (child process) is an exact copy of the calling process (parent process) in all of the following points:

- real and effective user and group IDs
- environment
- close-on-exec bit (see `exec()`)
- signal actions (`SIG_DFL`, `SIG_IGN`, address of the signal handling function)
- supplementary group IDs
- set-user-ID mode bit
- set-group-ID mode bit
- nice value (see `nice()`)
- all attached shared memory segments (see `shmop()`)
- process group ID
- session ID (see `exit()`)
- current working directory
- root directory
- file mode creation mask (see `umask()`)
- resource limits (see `getrlimit()`)
- controlling terminal

The child process differs from the parent process in the following points:

- The child process has a unique process ID. The child process ID also does not match any active process group ID.
- The child process also has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors. All the child's file descriptors share the same file description as the corresponding file descriptor of the parent.
- The child process has its own copy of the parent's directory streams. All directory streams in the child process may share the file position indicator with the corresponding directory stream of the parent.
- The child process may have its own copy of the parent's message catalog descriptors.
- The child process values for the `tms` structure components `tms_utime`, `tms_stime`, `tms_cutime` and `tms_cstime` are set to 0 (see `times()`).

- The time left until an alarm clock signal is reset to 0 (see `alarm()`).
- All `semadj` values are deleted (see `semop()`).
- File locks set by the parent process are not inherited by the child process (see also `fcntl()`).
- The set of signals pending for the child process is initialized to the empty set.

If threads are used, then the function affects the process or a thread in the following manner:

- A process is created with a single thread. If a "multi-threaded" process calls `fork()`, the new process contains a copy of the calling thread and its entire address space, including the state of Mutex objects and other resources. Fork handlers can be set up with the `pthread_atfork()` function.

#### *BS2000*

- BS2000 files, with the exception of memory pools, are not inherited with `fork()`. The `fork()` function cannot be called in the signal-handling function either. The following BS2000 resources are also not inherited:
  - Opened BS2000 files do not remain open
  - AID breakpoints
  - Task File Table (TFT)
  - SYSFILE assignments
  - Registered STXIT and contingency routines □

#### Return value

- |    |   |
|----|---|
| 0  | upon successful completion. <code>fork()</code> returns 0 to the child process and returns the process ID of the child process to the parent process. |
| -1 | if an error occurs. -1 is returned to the parent process, no child process is created, and <code>errno</code> is set to indicate the error.           |

#### Errors

`fork()` will fail if:

- |        |   |
|--------|---|
| EAGAIN | The system lacks the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user <code>{CHILD_MAX}</code> would be exceeded, or if DIV or FASTRAM areas are stored in the parent process. |
|--------|---|

#### *Extension*

- |        |                               |
|--------|-------------------------------|
| ENOMEM | The swap area is too small. □ |
|--------|-------------------------------|

#### Notes

As of this version, `fork()` can also be used in signal handling and contingency routines.

#### See also

`alarm()`, `exec`, `fcntl()`, `semop()`, `signal()`, `times()`, `sys/types.h`, `unistd.h`.

## fpathconf - get value of pathname variable

Syntax      #include <unistd.h>  
            long int fpathconf(int *files*, int *name*);

Description See `pathconf()`.



## fprintf, printf, sprintf - write formatted output on output stream

**Syntax**      `#include <stdio.h>`

```
int fprintf(FILE *stream, const char *format [, arglist]);
int printf (const char *format [, arglist]);
int sprintf (char *s, const char *format [, arglist]);
```

**Description** `fprintf()` writes formatted output on the output stream pointed to by *stream*.

`printf()` writes formatted output on the standard output stream `stdout`.

`sprintf()` writes formatted output, followed by the null byte, in consecutive bytes starting at the address *s*. The user must ensure that sufficient space is available.

Each of these functions converts the arguments in *arglist* and outputs them under the control of the *format*.

*format* is a character string, beginning and ending in its initial shift state, if defined. It is composed of zero or more directives and may include the following three types of characters:

- characters of type `char`, which are simply copied to the output stream (1: 1).
- white-space characters, starting with a backslash (`\`) (see `isspace()`).
- conversion specifications beginning with the percent character (`%`), each of which is associated with zero or more arguments in *arglist*. The results are undefined if fewer arguments are passed in *arglist* than are defined in *format*. If the number of arguments defined in *format* is greater than the arguments passed in *arglist*, the excess arguments are ignored.

### Characters

The following applies to the current version of the C runtime system:  
Only characters from the EBCDIC character set are permitted.

## White-space characters

Character	Meaning	Format control action
\b	backspace character	The output is shifted to 1 character before the current position, unless the current position is the start of a line In this version of the C runtime system, \b is evaluated only for BS2000 output, not for output to the POSIX subsystem.
\f	form-feed character	The output is shifted to the start of the next logical page. In this version of the C runtime system, \f is evaluated only for BS2000 output, not for output to the POSIX subsystem..
\n	newline character	The output is shifted to the start of the next line.
\r	carriage return	The output is shifted to the start of the current line. All output that was already written on the stream of this line is discarded.
\t	horizontal tab	The output is shifted to 8 characters after the current position.
\v	vertical tab	The output is shifted to the next vertical tab position. In this version of the C runtime system, \v is evaluated only for BS2000 output, not for output to the POSIX subsystem.

## Conversion specifications

Conversions can be applied to the  $n$ -th argument after the format in the argument list, rather than to the next unused argument. In this case, the conversion character % is replaced by the sequence % $n$ \$, where  $n$  is a decimal integer in the range [1, {NL\_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages.

In format strings containing the % $n$ \$ form of conversion specifications, elements in the argument list *arglist* can be referenced from the format string *format* as many times as required ( $n$ -times). In format strings containing the % form of conversion specifications, each argument in the argument list is evaluated exactly once.

All forms of `fprintf()` allow for the insertion of a language-dependent radix character in the output string. The radix character is defined in the program's locale (category LC\_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

Each conversion specification is introduced by the % character or by the character sequence % $n$ \$, after which the following appear in sequence:

- Zero or more **flags**, which modify the meaning of the conversion specification.
- An optional decimal number that specifies a minimum **field width**. If the converted value has fewer bytes than the field width, it will be padded to the field width with spaces on the left (or padded on the right if the left-adjustment flag "-" was specified).

- A **precision** that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x` and `X` conversions; the number of digits to appear after the radix character for the `e`, `E` and `f` conversions; the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of bytes to be printed from a string in `s` conversion. The precision takes the form of a period (`.`), followed by a decimal digit string, where a null digit string is treated as 0.
- An optional `h` specifying that a following `d`, `i`, `o`, `u`, `x` or `X` conversion character applies to a type `short int` or type `unsigned short int` argument (the argument will have been promoted according to the integral promotions, and its value will be converted to type `short int` or `unsigned short int` before printing);  
an optional `h` specifying that a following `n` conversion character applies to a pointer to a type `short int` argument;  
an optional `l` (`ell`) specifying that a following `d`, `i`, `o`, `u`, `x` or `X` conversion character applies to a type `long int` or `unsigned long int` argument;  
an optional `l` (`ell`) specifying that a following `n` conversion character applies to a pointer to a type `long int` argument;  
or an optional `L` specifying that a following `e`, `E`, `f`, `g` or `G` conversion character applies to a type `long double` argument.

If an `h`, `l` or `L` appears with any other conversion character, the behavior is undefined.

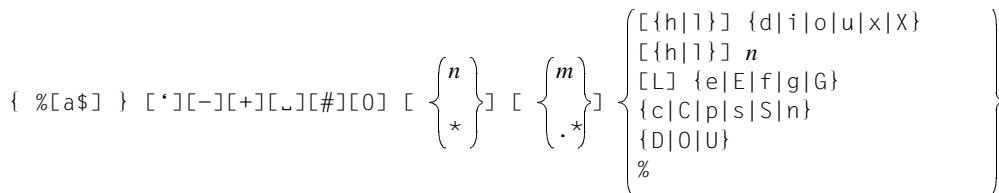
- A **conversion character** that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (`*`). In this case an argument of type `int` supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a `-` flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format strings containing the `%n$` form of a conversion specification, a field width or precision may be indicated by the sequence `*m$`, where `m` is a decimal integer in the range `[1, {NL_ARGMAX}]` giving the position in the argument list of an integer argument containing the field width or precision, for example:

```
printf ("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

*format* can contain either numbered argument specifications (that is, `%n$` and `*m$`), or unnumbered argument specifications (that is, `%` and `*`), but normally not both. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*-1)th, are specified in the format string.

Conversion specifications can be given in XPG4 Version 2-conformant environments as shown below:



- 
- 1.
  - 2.
  - 3.
  - 4.
  - 5.
1. Start of a conversion specification
  2. Flags
  3. Field width
  4. Precision
  5. Characters that define the actual conversion

### Flags

- ' The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %g or %G) will be formatted with thousands' grouping characters. For other conversions, the behavior is undefined. The non-monetary grouping character is used.
- The result of the conversion will be left-justified within the field.
- + The result of a signed conversion will always begin with a sign (+ or -).
- \_ If the first character of a signed conversion is not a sign, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.
- # This flag specifies that the value is to be converted to an alternative form. This flag has no effect for c, d, i, s and u. For o conversion, it increases the precision to force the first digit of the result to be 0. For x or X conversions, a non-zero result will have the string "0x" (or "0X") prefixed to it. For e, E, f, g or G conversions, the result will always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will not be removed from the result as they normally are.

0 For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g` and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and `-` flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x` and `X` conversions, if a precision is specified, the `0` flag will be ignored. For other conversions, the behavior is undefined.

### Conversion characters

`d`, `i` The `int` argument is converted to a signed decimal in the style `[-]ddd`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

`o` The `unsigned int` argument is converted to unsigned octal format in the style `ddd`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

`u` The `unsigned int` argument is converted to unsigned decimal format in the style `ddd`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

`x` The `unsigned int` argument is converted to unsigned hexadecimal format in the style `ddd`; the letters `abcdef` are used in addition to the digits. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

`X` Behaves the same as the `x` conversion character except that letters `ABCDEF` are used.

`f` The `double` argument is converted to decimal notation in the style `[-]ddd.d`, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly 0 and no `#` flag is present, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

`e`, `E` The `double` argument is converted in the style `[-]d.ddde+-dd`, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is 0 and no `#` flag is

- present, no radix character appears. The value is rounded to the appropriate number of digits. The E conversion character will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0.
- g, G The `double` argument is converted in the style `f` or `e` (or in the style `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted; style `e` (or `E`) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit.
- c The `int` argument is converted to an `unsigned char`, and the resulting byte is written.
- s The argument must be a pointer to an array of `char`. Bytes from the array are written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array must contain a null byte.
- p The argument must be a pointer to `void`. The value of the pointer is converted to a sequence of printable characters; in the POSIX subsystem, this is the hexadecimal representation of the address.
- n The argument must be a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the `printf` functions. No argument is converted.
- C `wchar_t` is converted to an array of bytes representing a character, and the resulting character is written. If the precision is specified, the effect is undefined. The conversion is the same as that expected from `wctomb()`. This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). This conversion character has therefore no effect.
- S The argument must be a pointer an array of type `wchar_t`. Wide character codes from the array, up to but not including any terminating null wide-character code are converted to a sequence of bytes, and the resulting bytes are written. If the precision is specified, no more than that many bytes are written, and only complete characters are written. If the precision is not specified, or is greater than the size of the array of converted bytes, the array of wide characters must be terminated by a null wide character. The conversion is the same as that expected from `wcstombs()`.

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). This conversion character has therefore no effect.

`%` The `%` character is output; no argument is converted.

If the character that follows `%` or the character sequence `%a$` is not a valid conversion character, the result of the conversion is undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `printf()` and `fprintf()` are printed as if `putc()` had been called.

The structure components `st_ctime` and `st_mtime` of the file are marked for changing between successful execution of `fprintf()` or `printf()` and the next successful completion of a call to `fflush()` or `fclose()` for the same data stream or a call to `exit()` or `abort()` (see `sys/stat.h`).

*BS2000*

The conversion specifications for output to `STDOUT` depend on whether KR (only available with C/C++ versions lower than V3) or ANSI functionality is to be supported. The appropriate specifications for both functionalities are detailed below.

**Conversion specifications (KR functionality)**

(only available with C/C++ versions lower than V3)

The conversion specifications may be entered in the following format:

$$\% \left[ [-] [ + ] [ 0 ] \left[ \left. \begin{matrix} n \\ * \end{matrix} \right\} \right] \left[ \left. \begin{matrix} m \\ . * \end{matrix} \right\} \right] \left\{ \begin{matrix} [ ] \{ d | o | u | x \} \\ \{ D | O | U | X \} \\ \{ e | f | g \} \\ \{ c | s \} \\ \% \end{matrix} \right\}$$

- 
- 1.
  - 2.
  - 3.

1. Every conversion specification must begin with a percent character (`%`).
2. Flags (i.e. formatting characters) to control the output of a sign, left or right justification, width of the output field, etc.
3. Characters that define the actual conversion.

Meanings of flags for KR functionality (only available with C/C++ versions lower than V3):

- Left-justified alignment of the output field. Default: right-justified alignment.
- + The result of a signed conversion will always be output with a sign.  
Default: only a negative sign, if present, is output.
- 0 Zero padding. The output field will be padded with zero for all conversions.  
Default: The output field is padded with blanks. Zero padding will also be used with left-justified alignment (flag –).
- n* Minimum field width (including radix character). If more positions are required for the conversion of a number, this specification has no effect. If the output is shorter than the specified field width, it is padded with blanks or zeros up to the field width (see flags – and 0).
- \* The total field width (see *n*) is defined by an argument instead of a conversion specification. The current (integral) value must immediately precede the argument to be converted or the value of the precision specification (flag *.m*) in the argument list (delimited by a comma).
- .m* Precision specification.  
e, f, g conversions: exact number of positions after the radix character  
Default: 6 positions.  
s conversion: maximum number of characters to be output. Default: all characters up to the terminating null byte.  
The precision specification is ignored for all other conversions.
- .\** The precision (see *.m*) is defined by an argument instead of a conversion specification. The current (integral) value must immediately precede the argument to be converted in the argument list (delimited by a comma).

Meanings of conversion characters for KR functionality (only available with C/C++ versions lower than V3):

- l l before d, o, u, x:  
conversion of an argument of type `long`.  
This specification is identical to be uppercase letters D, O, U, X.
- d, o, u, x Representation of an integer (`int`) as  
signed decimal number (d),  
unsigned octal number (o),  
unsigned decimal number (u),  
signed hexadecimal number (x).
- f Representation of a floating-point number (`float` or `double`) in the form `[-]ddd.ddd`.  
The radix character is determined by the locale (category `LC_NUMERIC`).



- The default is a period. The number of positions after the radix character depends on the precision specified in *.m*; the default is 6 positions. If the precision is set to 0, the output will have no radix character.
- e Representation of a floating-point number (float or double) in the form: `[-].ddd{+|-}dd`. The radix character is determined by the locale (category LC\_NUMERIC). The default is a period. The number of positions after the radix character depends on the precision specified in *.m*; the default is 6 positions. If the precision is set to 0, the output will include the radix character, but with no following digits.
  - g Representation of a floating-point number (float or double) in f or e form. The number of positions after the radix character depends on the precision specified in *.m*; the representation requiring the least space while maintaining the precision is selected.
  - c Format for the output of a single character (char). The null byte is ignored.
  - s Format for the output of strings. The printf functions write the same number of characters of the string as are specified in the precision *.m*. Default: all characters up to (but not including) a terminating null byte are written by printf() functions.
  - % Output of the character %, without conversion.

**Conversion specifications ANSI functionality)**

The conversion specifications may be entered in the following format :

$$\% \left[ [-][+][_][\#][0] \right] \left[ \left. \begin{matrix} n \\ * \end{matrix} \right\} \right] \left[ \left. \begin{matrix} m \\ . \\ * \end{matrix} \right\} \right] \left. \begin{matrix} \left[ \{h|l\} \right] \{d|i|o|u|x|X\} \\ \left[ \{h|l\} \right] \quad n \\ \left[ L \right] \{e|E|f|g|G\} \\ \{c|p|s\} \\ \{D|O|U\} \\ \% \end{matrix} \right\}$$

- 
- 1.
  - 2.
  - 3.

1. Every conversion specification must begin with a percent character (%).
2. Flags (i.e. formatting characters) to control the output of a sign, left or right justification, width of the output field, etc.
3. Characters that define the actual conversion.

## Meanings of flags (for ANSI functionality):

- Left-justified alignment of the output field. Default: right-justified alignment.
- + The result of a signed conversion will always be output with a sign. Default: only a negative sign, if present, is output.
- \_ If the first character of a signed string to be converted is not a sign, the result is prefixed by a blank. The flag \_ is ignored if + is specified at the same time.
- # Conversion of the result to an alternative form.  
For o conversion, the precision is increased to force the first digit of the result to be 0.  
For x or X conversions, a non-zero result will have the string 0x or 0 prefixed to it.  
For e, E, f, g or G conversions, the result will always contain a radix character, even if no digits follow the radix character (a radix character normally appears in the result of these conversions only if a digit follows it). Furthermore, for g and G conversions, trailing zeros will not be removed from the result.  
The flag # has no effect with c, s, d, i and u conversions.
- 0 Zero padding. The output field is padded with zeros on converting integers (d, i, o, u, x, X) and floating-point numbers (e, E, f, g, G). By default, the output field is padded with blanks. 0 is ignored if the flag o or a precision .m is specified when converting integers. The 0 flag has no effect with c, p and s conversions.
- n Minimum total field width (including radix character). If more positions are required for the conversion of a number, this specification has no effect. If the output is shorter than the specified field width, it is padded with blanks or zeros up to the field width (see flags – and 0).
- \* The total field width (see n) is defined by an argument instead of a conversion specification. The current (integral) value must immediately precede the argument to be converted or the value of the precision specification (flag .m) in the argument list (delimited by a comma).
- .m Precision specification.  
d, i, o, u, x or X conversions: minimum number of digits to be output. Default: 1.  
e, E, and f conversions: exact number of positions after the radix character (max 20). Default: 6 positions.  
g or G conversions: maximum number of significant positions.  
s conversions: maximum number of characters to be output. Default: all characters up to the terminating null byte (\0).

**.\*** The precision (see *.m*) is defined by an argument instead of a conversion specification. The current (integral) value must immediately precede the argument to be converted in the argument list (delimited by a comma).

#### Meanings of conversion characters (for ANSI functionality):

**h** h before d, i, o, u, x, X:  
conversion of an argument of type `short`.

h before n:  
The argument is of type pointer to `short int` (no conversion).

**l** l before d, i, o, u, x, X:  
conversion of an argument of type `long`.

l before d, o, u is equivalent to the uppercase letters D, O, U.

l before n:  
The argument is of type pointer to `long int` (no conversion).

**ll** ll before d, i, o, u, x, X:  
conversion of an argument of type `long long int` or `unsigned long long int`.

ll before n:  
The argument is of type pointer to `long long int`.

**L** L before e, E, f, g, G:  
conversion of an argument of type `long double`.

**d, i, o, u, x, X** Representation of an integer (`int`) as signed decimal number (d, i), unsigned octal number (o), unsigned decimal number (u), unsigned hexadecimal number (x, X).  
The lowercase letters `abcdef` are used with `x`, and the uppercase letters `ABCDEF` are used with `X`. The precision specification *.m* defines the minimum number of digits to be output. If the value can be represented using fewer digits, the result will be padded with leading zeros. A precision of 1 is set by default. The result of converting the value 0 with precision 0 is no output.

**f** Representation of a floating-point number (`float` or `double`) in the form `[-]ddd.ddd`. The radix character is determined by the locale (category `LC_NUMERIC`). The default is a period. The number of positions after the radix character depends on the precision specified in *.m*; the default is 6 positions. If the precision is set to 0, the output will have no radix character.

e, E	Representation of a floating-point number ( <code>float</code> or <code>double</code> ) in the form <code>[-]d.ddde{+ -}dd</code> . The radix character is determined by the locale (category <code>LC_NUMERIC</code> ). The default is a period. For E conversions, the exponent is prefixed by the uppercase letter E. The number of positions after the radix character depends on the precision specified in <code>.m</code> ; the default is 6 positions. If the precision is set to 0, the output will have no radix character.
g, G	Representation of a floating-point number ( <code>float</code> or <code>double</code> ) in <code>f</code> or <code>e</code> form (or for G conversions, in <code>E</code> form). The number of significant positions depends on the precision specified in <code>.m</code> . The <code>e</code> or <code>E</code> form is used only if the exponent of the conversion result is less than -4 or greater than the specified precision.
c	Format for the output of a single character ( <code>char</code> ). The null byte is ignored.
p	Conversion of an argument of type pointer to <code>void</code> . The output occurs as an 8-digit hexadecimal number (analogous to the entry <code>%08.8x</code> ).
s	Format for the output of strings. The <code>printf</code> functions write the same number of characters of the string as are specified in the precision <code>.m</code> . Default: all characters up to (but not including) a terminating null byte are written by <code>printf()</code> functions.
n	No conversion and output of the argument occurs. The argument is of type pointer to <code>int</code> . This integer variable is assigned the number of bytes that were generated for output by the <code>printf</code> functions up to that point.
%	Output of the character <code>%</code> , without conversion. □

## Return value

Number of bytes transferred (excluding null bytes for `sprintf()`) upon successful completion.

## Negative value

if an error occurs. `errno` is set to indicate the error.

Errors	fprintf() and printf() will fail if:	
EAGAIN	The <code>O_NONBLOCK</code> flag is set for the file descriptor underlying <i>stream</i> and the process would be delayed in the write operation.	
EBADF	The file descriptor underlying <i>stream</i> is not a valid file descriptor for writing.	
EFBIG	An attempt was made to write a file that exceeds the maximum file size or the process file size limit (see <code>ulimit()</code> ).	
EINTR	The write operation was terminated due to the receipt of a signal, and no data was transferred.	
EIO	The process is a member of a background process group attempting to write to its controlling terminal, <code>TOSTOP</code> is set, the process is neither ignoring nor blocking <code>SIGTOU</code> , and the process group of the process is orphaned.	
ENOSPC	No free space is available on the device containing the file.	
EPIPE	An attempt was made to write a pipe or FIFO that is not open for reading by any process. A <code>SIGPIPE</code> signal will also be sent to the process.	

Notes When floating-point numbers are converted, they are rounded to the specified precision by the `printf` functions.

The `printf` functions do not perform conversions from one data type to another. Values that are not to be output in accordance with their types must be converted explicitly (e.g. with the `cast` operator).

The characters are not written to the external file immediately, but are temporarily stored in an internal C buffer (see section "Buffering streams" on page 77).

The program environment determines whether `fprintf()` is executed for a BS2000 or POSIX file.

#### *BS2000*

Maximum number of characters to be output:

for KR functionality (only available with C/C++ versions lower than V3), a maximum of 1400 characters per `fprintf` call;

for ANSI functionality, a maximum of 1400 characters per conversion element (e.g. `%s`). □

Attempts to output uninitialized variables or to output variables in non-compliance with their data types may lead to undefined results.

If the percent character (`%`) in a conversion specification is followed by an undefined flag or conversion character, the behavior is undefined.

See also `fputc()`, `fscanf()`, `setlocale()`, `stdio.h`, section "Locale" on page 53.

## fputc - put byte on stream

**Syntax**      `#include <stdio.h>`  
`int fputc(int c, FILE *stream);`

**Description** `fputc()` converts the byte specified by *c* to an unsigned char and writes it to the output stream pointed to by *stream* at the position indicated by the associated file position indicator for the stream, if defined. The file position indicator is then advanced appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the byte is appended to the output stream.

The structure components `st_ctime` and `st_mtime` of the file are marked for changing between successful execution of `fputc()` and the next successful completion of a call to `fflush()` or `fclose()` for the same data stream or a call to `exit()` or `abort()` (see `sys/stat.h`).

### Return value

The written value

if successful.

EOF            if an error occurs, e.g. because *stream* was not opened for writing or the output file could not be extended. The error indicator for the stream is set, and `errno` is set to indicate the error.

### Errors

`fputc()` will fail if:

EAGAIN        The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

EBADF         The file descriptor underlying *stream* is not a valid file descriptor open for writing.

EFBIG         An attempt was made to write to a file that exceeds the maximum file size or the process file size limit (see `ulimit()`).

EINTR         The write operation was terminated due to the receipt of a signal, and no data was transferred.

EIO            The process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU` and the process group of the process is orphaned.

ENOSPC        There was no free space remaining on the device containing the file.

**EPIPE** An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the process.

If threads are used, then the function affects the process or a thread in the following manner: If an `EPIPE` error occurs, the `SIGPIPE` signal is not sent to the process, but is sent to the calling thread instead.

**Notes** The characters are not written immediately to the external file, but are stored in an internal C buffer (see section "Buffering streams" on page 77).

On output to text files, control characters for white space (`\n`, `\t`, etc.) are converted to their appropriate effect in accordance with the type of text file (see section "White-space characters" on page 84).

`fputc()` does not execute as fast as `putc()`, but requires less memory per call.

The program environment determines whether `fputc()` is executed for a BS2000 or POSIX file.

**See also** `ferror()`, `fopen()`, `putc()`, `puts()`, `setbuf()`, `stdio.h`, `sys/stat.h`.

## fputs - put string on stream

**Syntax**      `#include <stdio.h>`  
`int fputs(const char *s, FILE *stream);`

**Description** `fputs()` writes the null-terminated string pointed to by `s` to the stream pointed to by `stream`. The terminating null byte is not written.

The structure components `st_ctime` and `st_mtime` of the file are marked for changing between successful execution of `fputs()` and the next successful completion of a call to `fflush()` or `fclose()` for the same data stream or a call to `exit()` or `abort()` (see `sys/stat.h`).

**Return value**  
Non-negative number  
    if successful.  
  
*BS2000*  
0           if successful. □  
  
EOF         if an error occurs; `errno` is set to indicate the error.

**Errors**      See `fputc()`.

**Notes**       `puts()` appends a newline character while `fputs()` does not.  
  
On output to text files, control characters for white space (`\n`, `\t`, etc.) are converted to their appropriate effect in accordance with the type of text file (see section "White-space characters" on page 84).  
  
The program environment determines whether `fputs()` is executed for a BS2000 or POSIX file.

**See also**     `fopen()`, `fputc()`, `putc()`, `puts()`, `stdio.h`, `sys/stat.h`.



## fputc - put wide-character code on stream

**Syntax**      `#include <wchar.h>`

*Optional*

`#include <stdio.h>` □

`wint_t fputc(wint_t wc, FILE *stream);`

**Description** `fputc()` writes the character corresponding to the wide-character code *wc* to the output stream pointed to by *stream*, at the position indicated by the associated file-position indicator for the stream (if defined), and then advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream. If an error occurs when writing the character, the shift state of the output file is left in an undefined state.

The structure components `st_ctime` and `st_mtime` of the file are marked for changing upon successful execution of `fputc()` and the next successful completion of a call to `fflush()` or `fclose()` for the same data stream or a call to `exit()` or `abort()` (see `sys/stat.h`).

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**Return value**

*wc*                      upon successful completion.

WEOF                    if an error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

**Errors**                `fputc()` will fail if either the stream is unbuffered or data in the stream's buffer needs to be written, and:

EAGAIN                The `O_NONBLOCK` flag is set for the file descriptor underlying *stream*, and the process would be delayed in the write operation.

EBADF                 The file descriptor underlying the stream is not a valid file descriptor open for a write operation.

EFBIG                 An attempt was made to write to a file that exceeds the maximum file size or the process file size limit (see `ulimit()`).

EINTR                 The write operation was terminated due to the receipt of a signal, and no data was transferred.

*Extension*

EINVAL	An attempt was made to access a BS2000 file. □
EIO	The process is a member of a background process group attempting to write to its controlling terminal, <code>TOSTOP</code> is set, the process is neither ignoring nor blocking <code>SIGTTOU</code> and the process group of the process is orphaned.
ENOSPC	There was no free space remaining on the device containing the file.
EPIPE	An attempt is made to write to a pipe or <code>FIFO</code> that is not open for reading by any process. A <code>SIGPIPE</code> signal will also be sent to the process.  If threads are used, then the function affects the process or a thread in the following manner: If an <code>EPIPE</code> error occurs, the <code>SIGPIPE</code> signal is not sent to the process, but is sent to the calling thread instead.

**See also** `ferror()`, `fopen()`, `setbuf()`, `stdio.h`, `sys/stat.h`, `wchar.h`.

## fputws - put wide character string on stream

**Syntax**      `#include <wchar.h>`

*Optional*

`#include <stdio.h>` □

`int fputws(const wchar_t *ws, FILE *stream);`

**Description** `fputws()` writes a character string corresponding to the (null-terminated) wide character string pointed to by `ws` to the stream pointed to by `stream`. No character corresponding to the terminating null wide-character code is written.

The structure components `st_ctime` and `st_mtime` of the file are marked for changing between successful execution of `fputws()` and the next successful completion of a call to `fflush()` or `fclose()` for the same data stream or a call to `exit()` or `abort()` (see `sys/stat.h`).

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**Return value**

Non-negative number  
upon successful completion.

-1  
if an error occurs; e.g. because the stream is unbuffered or data in the stream's buffer needs to be written. The error indicator for the stream is set, and `errno` is set to indicate the error.

**Errors**      See `fputc()`.

**Notes**      `fputws()` does not append a newline character.

**See also**    `fopen()`, `fputc()`, `stdio.h`, `sys/stat.h`, `wchar.h`.

## fread - read binary data

**Syntax**      `#include <stdio.h>`  
                  `size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);`

**Description** `fread()` reads, into the array pointed to by *ptr*, up to *nitems* elements whose size is specified by *size* in bytes, from the stream pointed to by *stream*. The file position indicator for the stream (if defined) is advanced by the number of bytes successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial member is read, its value is indeterminate.

`fread()` can mark the structure component `st_atime` for the file to which *stream* is assigned for changing (see `sys/stat.h`). The structure component `st_atime` is updated as soon as `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` are called successfully for *stream* and return data which is not was not provided by a preceding call to `ungetc()` or `ungetwc()`.

*BS2000*

### Record I/O

`fread()` reads a record (or block) from the current file position.

Number of bytes to be read:

In the following, *n* is the total number of bytes to be read, i.e.

$$n = \text{size} * \text{nitems}$$

If *n* is greater than the current record length, only the current record will be read.

If *n* is less than the current record length, only the first *n* bytes of the record will be read, and the next read operation will access the data of the next record.

`fread()` returns the same value as for stream I/O, i.e. the number of elements read in their entirety. For record I/O, it is best to use only element length 1, since the return value will then correspond to the length of the record read (without any record length field). □

### Return value

- Number of elements successfully read upon successful completion. The return value is less than *nitems* only if a read error or end-of-file is encountered.
- 0 if *size* or *nitems* id equal to 0. The contents of the array pointed to by *ptr* and the state of the stream remain unchanged. `errno` is not set.
- if a read error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

**Errors**      See `fgetc()`.

- Notes** `ferror()` or `feof()` must be used to distinguish between an error condition and an end-of-file condition.
- The array to which *ptr* points must be large enough to hold the data elements read.
- To ensure that *size* specifies the correct number of bytes for a data element, the `sizeof()` function should be used for the size of the data unit to which *ptr* points.
- `fread()` reads beyond the newline (`\n`) character and is therefore specially suitable for reading binary files.
- The program environment determines whether `fread()` is executed for a BS2000 or POSIX file.
- See also** `feof()`, `ferror()`, `fgetc()`, `fopen()`, `getc()`, `gets()`, `scanf()`, `stdio.h`, `sys/stat.h`.

## free - free allocated memory

- Syntax** `#include <stdlib.h>`
- ```
void free(void *ptr);
```
- Description** `free()` releases memory space that was previously reserved using `malloc()`, `calloc()` or `realloc()`.
- `free()` is part of a C-specific memory management package with its own free memory management facility. Memory deallocated with `free()` is not returned to the operating system but is handled by the free memory management facility.
- ptr* is the pointer to the memory area to be released. *ptr* must be the result of a previous `malloc()`, `calloc()`, or `realloc()` call. Otherwise, the result is undefined.
- See also** `calloc()`, `malloc()`, `realloc()`, `sdtlib.h`.

## freopen - flush and reopen stream

**Name**        **freopen, freopen64**

**Syntax**      `#include <stdio.h>`

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);  
FILE *freopen64(const char *filename, const char *mode, FILE *stream);
```

**Description** `freopen()` first attempts to flush the stream and close any file descriptor associated with *stream*. Failure to flush or close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

`freopen()` then opens the file whose pathname is the string pointed to by *filename* and associates the stream pointed to by *stream* with it. The *mode* argument is used just as in `fopen()` (see `fopen()`).

The original stream is closed regardless of whether the subsequent open succeeds.

For automatic conversion, the `b` for binary must not be specified in *mode*. Furthermore, the environment variable `IO_CONVERSION` must not be present or must have the value `YES`.

There is no difference in functionality between `freopen` and `freopen64` except that `freopen64` returns a pointer that can point past the 2GB limit `freopen64()` sets the `O_LARGEFILE` bit in the File status flag.

*BS2000*

See `fopen()`, `fopen64()`.

*Restriction*

If *stream* references a BS2000 file and *filename* refers to a POSIX file, the POSIX file can be opened with `freopen()` only if *stream* refers to `stdin`, `stdout` or `stderr`. If this is not the case, only the BS2000 file is closed, and 0 is returned.

If *stream* references a POSIX file and *filename* refers to a BS2000 file, the BS2000 file can be opened with `freopen()` only if *stream* refers to `stdin`, `stdout` or `stderr`. If this is not the case, only the POSIX file is closed, and 0 is returned. This applies regardless of the current assignments for the standard streams.

**Return value**

Value of *stream*  
if successful.

Null pointer    if an error occurs; `errno` is set to indicate the error.

|              |                                                                                                                                                                                                                                                   |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Errors       | freopen() will fail if:                                                                                                                                                                                                                           |
| EACCES       | Search permission is denied on a component of the path, or the file exists and the permissions specified by <i>mode</i> are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created. |
| EINTR        | A signal was caught during the freopen() system call.                                                                                                                                                                                             |
| EISDIR       | The named file is a directory and <i>mode</i> requires write access.                                                                                                                                                                              |
| ELOOP        | Too many symbolic links were found when resolving the path.                                                                                                                                                                                       |
| EMFILE       | {OPEN_MAX} file descriptors are currently open in the calling process.                                                                                                                                                                            |
| ENAMETOOLONG | The length of the path argument exceeds {PATH_MAX} or a component of the path is longer than {NAME_MAX}.                                                                                                                                          |
| ENFILE       | The maximum allowable number of files is currently open in the system.                                                                                                                                                                            |
| ENOENT       | The specified file does not exist or <i>filename</i> points to an empty string.                                                                                                                                                                   |
| ENOSPC       | The file does not exist, and the directory or file system in which a new file was to be created cannot be expanded.                                                                                                                               |
| ENOTDIR      | A component of the pathname is not a directory.                                                                                                                                                                                                   |
| ENXIO        | The specified file is a character-oriented or block-oriented device file and the device assigned to this file does not exist.                                                                                                                     |
| EOVERFLOW    | The specified file is a regular file, but its size cannot be represented correctly in an object of type <i>off_t</i> .                                                                                                                            |
| EROFS        | The named file resides on a read-only file system and <i>mode</i> requires write access.                                                                                                                                                          |
| ETXTBSY      | The file is a pure procedure file (shared text file) that is currently executing and write protection is required for <i>mode</i> .                                                                                                               |

**Notes** freopen() is normally used to reassign the file pointers *stdin*, *stdout* and *stderr* to files other than the default files opened. *stderr* is not buffered by default, but can be buffered or line-buffered by using freopen().

The program environment determines whether freopen() is executed for a BS2000 or POSIX file.

*BS2000*

See fopen().

**See also** creat(), fclose(), fopen(), fdopen(), *stdio.h*.

## frexp - extract mantissa and exponent from double precision number

**Syntax**      `#include <math.h>`

```
double frexp(double num, int *exp);
```

**Description** `frexp()` splits a floating-point value *num* into the mantissa *x* and the exponent *exp* using the formula:

$$num = x * 2^{exp}$$

*|x|* is in the interval [0.5, 1.0]

*exp* is a pointer to an integer that specifies the exponent to the base 2t.

`frexp()` is the inverse function of `ldexp()`.

**Return value**

**Mantissa *x***      a floating-point number of type `double` that lies in the interval [0.5, 1.0] and satisfies the equation:  $num = x * 2^{exp}$ . The exponent is stored in *exp*.

**0**                      if *num* is equal to 0 (in which case the exponent is also equal to 0).

**Notes**                An application wishing to check for error situations should set `errno` to 0 before calling `frexp()`. If `errno` is set on return, an error has occurred.

**See also**             `ldexp()`, `modf()`, `math.h`.



## fscanf, scanf, sscanf - read formatted input

**Syntax**      `#include <stdio.h>`

```
int fscanf(FILE *stream, const char *format [, arglist]);
int scanf(const char *format [, arglist]);
int sscanf(const char *s, const char *format [, arglist]);
```

**Description** `scanf()` reads bytes from the standard input stream `stdin` according to a specified format. `fscanf()` reads bytes from the stream pointed to by `stream` according to a specified format. `sscanf()` reads bytes from the string `s` according to a specified format.

Each of these functions reads bytes, interprets them according to the directives given in the control string `format`, and stores the results in the areas specified by the arguments in `arglist`, if any.

`format` is a character string, beginning and ending in its initial shift state, if defined. It is composed of zero or more directives and may include the following three types of characters:

- characters of type `char`, which are simply copied to the output stream (1: 1).
- white-space characters, starting with a backslash (`\`) (see `isspace()`).
- conversion specifications beginning with the percent character (`%`), each of which is associated with zero or more arguments in `arglist`. The results are undefined if fewer arguments are passed in `arglist` than are defined in `format`. If the number of arguments defined in `format` is greater than the arguments passed in `arglist`, the excess arguments are ignored.

### Characters

The following applies to the current version of the C runtime system:  
Only characters from the EBCDIC character set are permitted.

The `scanf` functions read each input character, but do not convert it or store it in a variable. If the input character does not match the character specified in `format`, input processing is aborted.

### White-space characters

The control string `format` may include zero or more characters producing white space. These characters have no control function.

White-space characters in the input are treated as delimiters between input fields; they are not converted (see `%c` and `%[]` for exceptions). Leading white space in the input is ignored.

Depending on which functionality is to be supported by the `scanf` functions, a different number of white-space characters are recognized (shown in the table below):

| Character       | Meaning                  | Valid for following functionality |               |             |
|-----------------|--------------------------|-----------------------------------|---------------|-------------|
|                 |                          | XPG4                              | ANSI (BS2000) | KR (BS2000) |
| <code>␣</code>  | <b>blank</b>             | x                                 | x             | x           |
| <code>\n</code> | <b>newline character</b> | x                                 | x             | x           |
| <code>\t</code> | <b>horizontal tab</b>    | x                                 | x             | x           |
| <code>\f</code> | <b>form-feed</b>         | x                                 | x             | -           |
| <code>\v</code> | <b>vertical tab</b>      | -                                 | x             | -           |
| <code>\r</code> | <b>carriage return</b>   | -                                 | x             | -           |

### Conversion specifications

Conversions can be applied to the  $n$ -th argument after the format in the argument list *arglist*, rather than to the next unused argument. In this case, the conversion character `%` is replaced by the sequence `%n$`, where  $n$  is a decimal integer in the range `[1, {NL_ARGMAX}]`, giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the `%n$` form of conversion specifications, it is unspecified whether numbered elements in the argument list *arglist* can be referenced from the format string *format* more than once.

*format* can contain either form of a conversion specification, that is, `%` or `%n$`, but the two forms cannot normally be mixed within a single format string. The only exception to this is that `%%` or `%*` can be mixed with the `%n$` form.

All forms of `fscanf()` allow for the insertion of a language-dependent radix character in the input string. The radix character is defined in the program's locale (category `LC_NUMERIC`). In the `POSIX` locale, or in a locale where the radix character is not defined, the radix character defaults to a period (`.`).

Each conversion specification is introduced by the `%` character or by the character sequence `%n$`, after which the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional non-zero decimal integer that specifies the maximum **field width**.
- An optional size modifier `h`, `l`, `ll` or `L` indicating the size of the receiving object. The conversion characters `d`, `i` and `n` must be preceded by `h` if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, or by `l` (ell) if it is a pointer to `long int`. Similarly, `o`, `u` and `x` must be preceded by `h` if the corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`, or by `l` (ell) if it is a pointer to `unsigned long int` or an `ll` if it is a pointer to a `long long int`.

Finally, e, f and g must be preceded by l (ell) if the corresponding argument is a pointer to double rather than a pointer to float.

If an h, l (ell) or L appears with any other conversion character, the behavior is undefined.

- A **conversion character** that specifies the type of conversion to be applied.

fscanf() executes each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters is executed by reading input until no more valid input bytes can be read, or up to the first byte which is not a white-space character (which remains unread).

A directive that is an ordinary character is executed as follows. The next byte is read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent bytes remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification is executed in the following steps:

Input white-space characters are skipped, unless the conversion specification includes a [ or one of the conversion characters c or n.

An item is read from the input, unless the conversion specification includes an n conversion character. An input item is defined as the longest sequence of input bytes (up to any specified maximum field width) which is an initial subsequence of a matching sequence. The first byte, if any, after the input item remains unread. If the length of the input item is 0, the execution of the conversion specification fails; this condition is a matching failure, unless an error prevented input, in which case it is an input failure.

Except in the case of a % conversion character, the input item (or, in the case of a %n conversion specification, the count of input bytes) is converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a \*, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

Conversion specifications can be given in XPG4-conformant environments as shown below:

$$\{ \%[a\$] \} [ \left. \begin{array}{c} n \\ * \end{array} \right\} \left\{ \begin{array}{l} [ \{h|l\} ] \{d|i|o|u|x|X\} \\ [ \{h|l\} ] \ n \\ [ | | L ] \{e|E|f|g|G\} \\ \{c|C|p|s|S\} \\ \{ [ \dots ] | [ ^ \dots ] \} \\ \% \end{array} \right\}$$

### Conversion characters

- d Matches an optionally signed decimal integer, whose format is the same as expected for `strtod()` with the value 10 for *base*. The corresponding argument must be of type pointer to `int`.
- i Matches an optionally signed decimal integer, whose format is the same as expected for `strtol()` with the value 0 for *base*. The corresponding argument must be of type pointer to `int`.
- o Matches an optionally signed octal integer, whose format is the same as expected for `strtol()` with the value 8 for *base*. The corresponding argument must be of type pointer to `unsigned`.
- u Matches an optionally signed decimal integer, whose format is the same as expected for `strtol()` with the value 10 for *base*. The corresponding argument must be of type pointer to `unsigned`.
- x, X Matches an optionally signed hexadecimal integer, whose format is the same as expected for `strtol()` with the value 16 for *base*. The corresponding argument must be of type pointer to `unsigned`.
- e, E, f, g, G These conversion characters match an optionally signed floating-point number, whose format is the same as expected for `strtod()`. The corresponding argument must be of type pointer to `float`.
- s Matches a sequence of bytes that are not white-space characters. The corresponding argument must be a pointer to the initial byte of a `char` array that is large enough to accept the sequence and a terminating null character byte, which will be added automatically.
- S Matches a sequence of characters that are not white space. The sequence is converted to a sequence of wide character codes in the same manner as `mbstowcs()`. The corresponding argument must be a pointer to the first byte of an array of type `wchar_t`, which must be large enough to accept the

sequence and a terminating null byte, which will be added automatically. If the field width is specified, it determines the maximum number of characters accepted.

This conversion character is only recognized in XPG4 mode.

- [ Matches a non-empty sequence of bytes from a set of expected bytes (the scanset). The corresponding argument must be a pointer to the initial byte of a `char` array that is large enough to accept the sequence and a terminating null byte, which is added automatically. The conversion specification includes all subsequent bytes in the *format* string up to and including the matching right square bracket (`]`). The bytes between the square brackets (the scanlist) comprise the scanset, unless the byte after the left square bracket is a circumflex (`^`), in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right square bracket. As a special case, if the conversion specification begins with `[ ]` or `[ ^ ]`, the right square bracket is included in the scanlist, and the next right square bracket is the matching right square bracket that ends the conversion specification. If a `-` is in the scanlist and is not the first character, nor the second where the first character is a `^`, nor the last character, the behavior is undefined.
- c Matches a sequence of bytes of the number specified by the field width (or 1 if no field width is present). The corresponding argument must be a pointer to the initial byte of a `char` array that is large enough to accept the sequence. No terminating null byte is added. The normal skip over white-space characters is suppressed in this case; `%1s` should be used to read the next byte that is not a white-space character.
- C Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The sequence is converted to a sequence of wide character codes in the same manner as `mbstowcs()`. The corresponding argument must be a pointer to the first byte of an array of type `wchar_t` large enough to accept the sequence which is the result of the conversion. No null wide character code is added.  
If the matched sequence begins with the initial shift state, the conversion is the same as expected for the `mbstowcs()` function; otherwise, the behavior of the conversion is undefined. The normal skip over white-space characters is suppressed in this case.
- p Matches a set of sequences, which must be the same as the set of sequences that is produced by the `%p` conversion of the `printf` functions. `p` must match the implementation for `printf` functions. The corresponding argument must be a pointer to a pointer to `void`. The interpretation of the input item is implementation-dependent; if the input item is not a value that

was converted earlier during the same program execution, the behavior of the %p conversion is undefined. This is specially true for pointer outputs generated by other systems.

- n No input is processed. The corresponding argument must be a pointer to the integer into which the number of input bytes read thus far by this call are to be entered. Execution of a %n conversion specification does not increment the assignment count returned at the completion of execution of the function.
- % Matches a single %; no conversion or assignment occurs. The complete conversion specification must be %%.

If a conversion specification is invalid, the behavior of `scanf()` is undefined.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any bytes matching the current conversion specification have been read (other than leading white-space characters, where permitted), execution of the current conversion specification terminates with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) is terminated with an input failure.

Reaching the end of the string in a `sscanf` call is equivalent to encountering the end-of-file indicator during an `fscanf` call.

If conversion terminates on a conflicting input, the offending input byte is left unread in the input stream. Any trailing white space (including newline characters) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments cannot be directly determined, except via the %n conversion specification.

*BS2000*

### **Conversion specifications (KR functionality)**

(only available with C/C++ versions lower than V3)

Conversion specifications contain directives that specify how input fields are to be interpreted and converted. They may be entered in the following format:

$$\% \left[ \begin{array}{c} n \\ * \end{array} \right] \left\{ \begin{array}{l} [\{h|l\}] \{d|o|x\} \\ [l] \{e|f\} \\ [D|E|F|O|X] \\ \{c|s\} \\ \{[...]|[^{...}]\} \\ \% \end{array} \right\}$$

Every conversion specification must begin with a percent character (%). The remaining characters are interpreted as follows:

- \*** Skip an assignment.  
The next input field is read and converted, but not stored in a variable.
- n** Maximum length of the input field to be converted.  
If a white-space character or a character that does not match the type specified in the conversion specification appears before this entry, the length is truncated accordingly.
- l** **l** before d, o, x:  
conversion of an argument of type pointer to long int (d) or unsigned long int (o, x). The specification is identical to the uppercase letters D, O, X.  
  
**l** before e, f:  
conversion of an argument of type pointer to double.  
The specification is identical to the uppercase letters E, F.
- h** **h** before d, o, x:  
conversion of an argument of type pointer to short int (d) or unsigned short int (o, x).
- d** A decimal integer value is expected. The corresponding argument must be a pointer to int.
- o** An octal integer value is expected. The corresponding argument may be a pointer to unsigned int or int. The value is internally represented as unsigned.
- x** A hexadecimal integer value is expected. The corresponding argument may be a pointer to unsigned int or int. The value is internally represented as unsigned.
- e, f** A floating-point number is expected. The corresponding argument must be a pointer to float. The floating-point number can contain a sign as well as an exponent (E or e, followed by an integer value). The radix character is determined by the locale (category LC\_NUMERIC). The default is a period.
- c** A character is expected. The corresponding argument should be a pointer to char. In this case scanf() will also read blanks. %1s should be used to read the next non-blank character. c is suitable for reading strings that include blanks; to do so, a pointer to a char array must be passed as an argument, and a field length of n must be specified (e.g. %10c). The scanf() function does not automatically terminate the string with the null byte in this case.

- s A string is expected. The corresponding argument must be a pointer to a char array that is large enough to accept the string and a terminating null byte. scanf() automatically terminates the string with the null byte. Leading white-space characters in the input are ignored; a trailing white-space character is interpreted as a delimiter (end of the string).
- [ ] A string is expected. The corresponding argument must be a pointer to a char array that is large enough to accept the string (including the automatically appended null byte). In this specification, as opposed to %s, blanks do not automatically function as delimiters.
- [... ] In this specification, characters are read in until the first character not listed in the square brackets appears. Thus, the string may only consist of the characters appearing within [ ]; any characters not specified are treated as delimiters.
- [^... ] In this specification, characters are read in until one of the characters listed in the square brackets after ^ is encountered. Only the characters specified within the [ ] are treated as delimiters.
- % Input of the % character, without conversion. □

*BS2000***Conversion specifications (ANSI functionality)**

Conversion specifications contain directives that specify how input fields are to be interpreted and converted. They may be entered in the following format:

$$\% \left[ \begin{array}{c} n \\ * \end{array} \right] \left\{ \begin{array}{l} [\{h|l\}] \{d|i|o|u|x|X\} \\ [\{h|l\}] n \\ [l|L] \{e|E|f|g|G\} \\ \{c|p|s\} \\ \{[...]|[^...]\} \\ \% \end{array} \right\}$$

Leading white-space characters in the input are ignored.

Every conversion specification must begin with a percent character (%). The remaining characters are interpreted as follows:

- \* Skip an assignment. The next input field is read and converted, but not stored in a variable.



|          |                                                                                                                                                                                                                                                                                                             |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>n</i> | Maximum length of the input field to be converted. If a white-space character or a character that does not match the type specified in the conversion specification appears before this entry, the length is truncated accordingly.                                                                         |
| l        | l before d, i, o, u, x, X:<br>conversion of an argument of type pointer to long int (d, i) or unsigned long int (o, u, x, X).<br><br>l before e, E, f, g, G:<br>conversion of an argument of type pointer to double.<br><br>l before n:<br>The argument is of the type pointer to long int (no conversion). |
| ll       | ll before d, i, o, u, x, X:<br>conversion of an argument of type long long int or unsigned long long int.<br><br>ll before n:<br>The argument is of type pointer to long long int.                                                                                                                          |
| h        | h before d, i, o, u, x, X:<br>conversion of an argument of type pointer to short int (d, i) or unsigned short int (o, u, x, X).<br><br>h before n: The argument is of the type pointer to short int (no conversion).                                                                                        |
| L        | L before e, E, f, g, G: conversion of an argument of type pointer to long double.                                                                                                                                                                                                                           |
| d        | A decimal integer value is expected. The corresponding argument must be a pointer to int.                                                                                                                                                                                                                   |
| i        | An integer value is expected. The base (hexadecimal, octal, decimal) is determined from the structure of the input field. Leading 0x or 0X: hexadecimal; leading 0: octal; otherwise: decimal. The corresponding argument must be a pointer to int.                                                         |
| o        | An octal integer value is expected. The corresponding argument may be a pointer to unsigned int or int. The value is internally represented as unsigned.                                                                                                                                                    |
| u        | A decimal integer value is expected. The corresponding argument must be a pointer to unsigned int.                                                                                                                                                                                                          |
| x, X     | A hexadecimal integer value is expected. The corresponding argument may be a pointer to unsigned int or int. The value is internally represented as unsigned.                                                                                                                                               |

- e, E, f, g, G A floating-point number is expected. The corresponding argument must be a pointer to `float`. The floating-point number can contain a sign as well as an exponent (E or e, followed by an integer value). The radix character is determined by the locale (category `LC_NUMERIC`). The default is a period.
- c A character is expected. The corresponding argument should be a pointer to `char`. In this case `scanf()` will also read blanks. `%1s` should be used to read the next non-blank character. `c` is suitable for reading strings that include blanks; to do so, a pointer to a `char` array must be passed as an argument, and a field length of `n` must be specified (e.g. `%10c`). The `scanf()` function does not automatically terminate the string with the null byte in this case.
- p An 8-digit pointer value is expected, analogous to the format `%08.8x`. The corresponding argument must be of type pointer to `void`.
- s A string is expected. The corresponding argument must be a pointer to a `char` array that is large enough to accept the string and a terminating null byte. `scanf()` automatically terminates the string with the null byte. Leading white-space characters in the input are ignored; a trailing white-space character is interpreted as a delimiter (end of the string).
- [ ] A string is expected. The corresponding argument must be a pointer to a `char` array that is large enough to accept the string (including the automatically appended null byte). In this specification, as opposed to `%s`, blanks do not automatically function as delimiters.
- [ . . . ] In this specification, characters are read until the first character not listed in the square brackets appears. Thus, the string may only consist of the characters appearing within [ ]; characters not specified therein are treated as delimiters. The closing bracket ] can be included in the list of characters to be read by specifying it as the first character immediately after the opening bracket: [ ] . . . ].
- [ ^ . . . ] In this specification, characters are read until one of the characters listed in the square brackets after ^ is encountered. Only the characters specified within the [ ] are treated as delimiters. The closing bracket ] can be included in the list of delimiters by specifying it as the first character immediately after the ^ character: [ ^ ] . . . ].
- n No characters are read from the input field. The argument is of type pointer to `int`. This integer variable is assigned the number of characters processed thus far by `scanf()`.
- % Input of the % character, without conversion. □

`fscanf()` and `scanf()` can mark the structure component `st_atime` for the file to which *stream* is assigned for changing (see `sys/stat.h`). The structure component `st_atime` is updated as soon as `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` are called successfully for *stream* and return data which is not was not provided by a preceding call to `ungetc()` or `ungetwc()`.

### Return value

number of successfully matched and assigned input items upon successful completion.

0 if an input character that does not match the format string is found at the outset.

EOF if the input ends before the first conflicting input or conversion. In contrast to XPG4, `errno` is not set.

### Notes

If the application calling `fprintf()` has any objects of type `wchar_t`, it must also include either `sys/types.h` or `stddef.h` to have `wchar_t` defined.

In format strings containing the `%` form of conversion specifications, each argument in the argument list is used exactly once. In format strings containing the `%n$` form of conversion specifications, each numbered argument of the argument list may be used as often as required.

When integer values are converted to `unsigned int` (`o`, `u`, `x`, `X`) the two's complement is formed from a value with a negative sign. For example, format `%u` for input `-1` returns `X'FFFFFFFF'`.

The return value of a `scanf` call should always be checked to ensure that no error has occurred!

The next `scanf` call starts reading immediately after the character last processed by the previous call.

If an input character does not correspond to the format specified, it is written back to the input buffer. It must be fetched from there with `getc()`; otherwise, the next `scanf` call will receive the same character again.

The program environment determines whether `fscanf()` is executed for a BS2000 or POSIX file.

**See also** `getc()`, `printf()`, `setlocale()`, `strtod()`, `strtol()`, `langinfo.h`, `stdio.h`.

## fseek - reposition file position indicator in stream

**Name**        **fseek, fseek64, fseeko, fseeko64**

**Syntax**      `#include <stdio.h>`

```
int fseek(FILE *stream, long int offset, int whence);
int fseek64(FILE *stream, long long int offset, int whence);
int fseeko(FILE *stream, off_t offset, int whence);
int fseeko64(FILE *stream, off64_t offset, int whence);
```

**Description** When POSIX files are executed, the function behaves in conformance with XPG as described below:

`fseek()` sets the file position indicator for the stream pointed to by *stream*.

The new position, measured in bytes from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*. The specified point is the beginning of the file for `SEEK_SET`, the current value of the file position indicator for `SEEK_CUR`, or end-of-file for `SEEK_END`.

If the *stream* is to be used with wide character input/output functions, *offset* must either be 0 or a value returned by an earlier call to `ftell()` on the same stream and *whence* must be `SEEK_SET`.

A successful call to `fseek()` clears the end-of-file indicator for the stream and undoes any effects of `ungetc()` and `ungetwc()` on the same stream. After an `fseek()` call, the next operation on a stream opened for an update may be either input or output.

If the most recent operation, other than `ftell()`, on a given stream is `fflush()`, the file offset in the underlying open file description will be adjusted to reflect the location specified by `fseek()`.

`fseek()` allows the file position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return null bytes until data is actually written into the gap.

If the stream was opened for writing and buffered data has not yet been written to the underlying file, `fseek()` will cause the unwritten data to be written to the file and mark the `st_ctime` and `st_mtime` fields of the file for update.

The `fseek64()` function behaves like `fseek()` except that the offset type `long long` is used for `fseek64()`.

There is no difference in functionality between `fseeko()` and `fseeko64()` except that `fseeko64()` uses the `off64_t` structure.

The `fseeko()` function is the same as the modified `fseek()` function except that the offset argument is of type `off_t` and that the `EOverflow` error has changed.

*BS2000*

The following must be noted when executing BS2000 files:

`fseek()` sets the file position indicator for the file associated with *stream* in accordance with the specifications in *offset* and *whence*. This allows files to be processed non-sequentially.

Text files (SAM in text mode, ISAM) can be positioned absolutely to the beginning or end of the file as well as to any position previously marked with `ftell()`.

Binary files (SAM in binary mode, PAM, INCORE) can be positioned absolutely (see above) or relatively, i.e. relative to beginning of file, end of file, or current position (by a desired number of bytes).

The significance, combination options, and effects of the *offset* and *whence* parameters differ for text and binary files and are therefore discussed individually below:

**Text files (SAM in text mode, ISAM)**

Possible values:

|               |                                                         |
|---------------|---------------------------------------------------------|
| <i>offset</i> | 0L or value determined by a previous <i>ftell</i> call. |
| <i>whence</i> | SEEK_SET (beginning of file)<br>SEEK_END (end of file)  |

Meaningful combinations and their effects:

| <i>offset</i>            | <i>whence</i> | Effect                                                  |
|--------------------------|---------------|---------------------------------------------------------|
| <code>ftell</code> value | SEEK_SET      | Position to the location marked by <code>ftell()</code> |
| 0L                       | SEEK_SET      | Position to the beginning of the file                   |
| 0L                       | SEEK_END      | Position to the end of the                              |

**Binary files (SAM in binary mode, PAM, INCORE)**

Possible values:

|               |                                                                                                                                                                                                                                                                                                     |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>offset</i> | Number of bytes by which the current file position indicator is to be shifted.<br>This number may be<br><br>positive: position forwards toward the end of the file<br>negative: position backwards toward the beginning of the file<br>0L: absolute positioning to the beginning or end of the file |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

*whence* For absolute positioning to the beginning or end of the file, the location at which the file position indicator is to be set.

For relative positioning, the reference point from which the file position indicator is to be moved by *offset* bytes:

SEEK\_SET (beginning of file)

SEEK\_CUR (current position)

SEEK\_END (end of file)

Meaningful combinations and their effects:

| <i>offset</i>      | <i>whence</i>                    | <b>Effect</b>                                                                                                 |
|--------------------|----------------------------------|---------------------------------------------------------------------------------------------------------------|
| 0L                 | SEEK_SET                         | Position to the beginning of the file.                                                                        |
| 0L                 | SEEK_END                         | Position to the end of the file.                                                                              |
| positive number    | SEEK_SET<br>SEEK_CUR<br>SEEK_END | Forward positioning from beginning of file, from current position, from end of file (beyond the end of file). |
| negative number    | SEEK_CUR<br>SEEK_END             | Backward positioning from current position, from end of file.                                                 |
| <i>ftell</i> value | SEEK_SET                         | Position to the location marked by an <i>ftell</i> call.                                                      |

Return value

0 if successful.

-1 if the specified file cannot be positioned. *errno* is set to indicate the error.

An improper seek can be, for example, an *fseek()* done on a file that has not been opened via *fopen()*; in particular, *fseek()* may not be used on a terminal or on a file opened via *popen()*. After a stream is closed, no further operations are defined on that stream.

*fseek()* and *fseeko()* will fail if, either the stream is unbuffered or the stream's buffer needed to be flushed, and the call to *fseek()* or *fseeko()* causes an underlying *lseek()* or *write()* to be invoked:

EAGAIN The *O\_NONBLOCK* flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

EBADF The file descriptor underlying *stream* is not open for writing or the stream's buffer needed to be flushed and the file is not open.

|           |                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EFBIG     | An attempt was made to write a file that exceeds the maximum file size or the process file size limit (see also <code>ulimit()</code> ).                                                                                                                                                                                                                                                                         |
| EINTR     | The write operation was terminated due to the receipt of a signal, and no data was transferred.                                                                                                                                                                                                                                                                                                                  |
| EINVAL    | The <i>whence</i> argument is invalid. The resulting file-position indicator would be set to a negative value.                                                                                                                                                                                                                                                                                                   |
| EIO       | An I/O error occurred.<br>The process is a member of a background process group attempting to write to its controlling terminal, <code>TOSTOP</code> is set, the process is neither ignoring nor blocking the <code>SIGTTOU</code> signal, and the process group of the process is orphaned.                                                                                                                     |
| ENOSPC    | There was no free space remaining on the device containing the file.                                                                                                                                                                                                                                                                                                                                             |
| EPIPE     | An attempt was made to write to a pipe or FIFO that is not open for reading by any process; a <code>SIGPIPE</code> signal will also be sent to the process.<br><br>If threads are used, then the function affects the process or a thread in the following manner: If an <code>EPIPE</code> error occurs, the <code>SIGPIPE</code> signal is not sent to the process, but is sent to the calling thread instead. |
| ENXIO     | The device does not exist or it cannot be accessed.                                                                                                                                                                                                                                                                                                                                                              |
| EOVERFLOW | For <code>fseek()</code> : The resulting file offset value cannot be represented correctly in an object of type <code>long</code> .                                                                                                                                                                                                                                                                              |
| EOVERFLOW | For <code>fseeko()</code> : The resulting file offset value cannot be represented correctly in an object of type <code>off_t</code> .                                                                                                                                                                                                                                                                            |

**Notes** For POSIX files, the file offset returned by `ftell()` is measured in bytes, and a seek to a position relative to that file offset is permissible; however, portability to other systems requires that a direct file offset (i.e. the value returned by `ftell()`) be used by `fseek()`. Arithmetic operations cannot always be meaningfully performed on any other file offset which may not necessarily be measured in bytes.

The program environment determines whether `fseek()` is executed for a BS2000 or POSIX file.

#### *BS2000*

The call `fseek(stream,0L,SEEK_SET)` is equivalent to the call `rewind(stream)`.

If new records are written to a text file that was opened in the write or append mode and an `fseek` call is issued, any data that may still be in the buffer is first written to the file and terminated with a newline character (`\n`).

Exception for ANSI functionality:

If the data of an ISAM file in the buffer does not end in a newline character, `fseek()` does not insert a change of line (or record). In other words, the data is not automatically terminated with a newline character when it is written from the buffer. Subsequent data extends the record in the file. Consequently, when an ISAM file is read, only the newline characters that were explicitly written by the program are read in.

If a binary file is positioned past the end of file, a gap appears between the last physically stored data and the newly written data. Reading from this gap returns binary zeros.

It is not possible to position to system files (SYSDDTA, SYSLST, SYSOUT).

A successful `fseek()` call deletes the EOF flag of the file and cancels all the effects of the preceding `ungetc` calls for that file.

In the case of record I/O, `fseek()` can be only be used for positioning to the beginning or end of the file.

`fseek(stream,0L,SEEK_SET)` positions on the first record of the file.

`fseek(stream,0L,SEEK_END)` positions after the last record of the file.

If `fseek()` is called with any other arguments, it will return EOF.

**See also** `fopen()`, `fsetpos()`, `ftell()`, `lseek()`, `rewind()`, `tell()`, `ungetc()`, `stdio.h`.



## fsetpos - set file position indicator for stream to current value

**Name**        **fsetpos, fsetpos64**

**Syntax**      `#include <stdio.h>`

```
int fsetpos(FILE *stream, const fpos_t *pos);
int fsetpos64(FILE *stream, const fpos64_t *pos);
```

**Description** `fsetpos()` sets the file position indicator for the stream pointed to by *stream* to a position *pos*, obtained from an earlier call to `fgetpos()`.

`fsetpos()` clears the end-of-file indicator for the stream and undoes any effects of `ungetc()` on the same stream. After an `fsetpos()` call, the next operation on an update stream may be either input or output.

There is no difference in the functionality between `fsetpos()` and `fsetpos64()` except that `fsetpos64()` uses an `fpos64_t` type.

**Return value**

0                if successful.

≠ 0             if an error occurs.

*BS2000*  
errno is set to EBADF.

**Notes**        The program environment determines whether `fsetpos()` is executed for a BS2000 or POSIX file.

*BS2000*

`fsetpos()` can be used on binary files (SAM in binary mode, PAM, INCORE) and text files (SAM in text mode, ISAM). `fsetpos()` cannot be used on system files (SYSDDTA, SYSLST, SYSOUT).

A successful call to the `fsetpos()` function deletes the EOF flag of the file and cancels all the effects of the preceding `ungetc` calls for that file.

If new records are written to a text file that was opened in the write or append mode and an `fsetpos` call is issued, any data that may still be in the buffer is first written to the file and terminated with a newline character (`\n`).

Exception for ANSI functionality:

If the data of an ISAM file in the buffer does not end in a newline character, `fsetpos()` does not insert a change of line (or record). In other words, the data is not automatically terminated with a newline character when it is written from the buffer. Subsequent data extends the record in the file. Consequently, when an ISAM file is read, only the newline characters that were explicitly written by the program are read in.

After positioning, the next operation may be either a read or write operation.

For ISAM files, the function pair `fgetpos()/fsetpos()` is far more efficient than the comparable function pair `ftell()/fseek()`.

In the case of record I/O in ISAM files with key duplication, `fsetpos()` cannot be used to position on the second or higher record of a group with identical keys. This can only be done by sequential reading or deletion. `fsetpos()` can only be used to position on the first record or after the last record of such a group.

**See also** `fgetpos()`, `fseek()`, `ftell()`, `open()`, `rewind()`, `ungetc()`, `stdio.h`.

## fstat - get file status of open file

**Name**        **fstat, fstat64**

**Syntax**      `#include <sys.stat.h>`

*Optional*

`#include <sys/types.h>`

`int fstat(int fildev, struct stat *buf);`

`int fstat64(int fildev, struct stat64 *buf);`

**Description** `fstat()` obtains information on an open file associated with the file descriptor *fildev*, which is returned by a successful `open()`, `creat()`, `dup()`, `fcntl()` or `pipe()` system call.

*buf* is a pointer to a `stat` structure into which information concerning the respective file is placed.

There is no difference in functionality between `fstat()` and `fstat64()` except that `fstat64()` returns the file status in a `stat64` structure.

The contents of the structure pointed to by *buf* include the following members:

```
mode_t    st_mode;    /* File mode (see mknod()) */
ino_t     st_ino;     /* Inode number (i-Node) */
dev_t     st_dev;     /* ID of device containing a
                        directory entry for this file */
dev_t     st_rdev;    /* Device ID, only defined for
                        character special or block special files */
nlink_t   st_nlink;   /* Number of links */
uid_t     st_uid;     /* User ID of the file's owner */
gid_t     st_gid;     /* Group ID of the file's group */
off_t     st_size;    /* File size in bytes */
time_t    st_atime;   /* Time of last access */
time_t    st_mtime;   /* Time of last data modification */
time_t    st_ctime;   /* Time of last file status change
                        The time is measured in seconds since
                        00:00:00 UTC, Jan 1, 1970 */
long      st_blksize; /* Preferred I/O block size */
blkcnt_t  st_blocks;  /* Number of st_blksize blocks allocated */
```

The `stat64` structure is defined like the `stat` structure except for the following components:

```
ino64_t st_ino
off64_t st_size and
blkcnt64_t st_blocks
```

The elements have the following meanings:

|                         |                                                                                                                                                                                                                                                                                                     |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>st_mode</code>    | The mode of the file is defined in the system call <code>mknod()</code> . Apart from the modes defined in <code>mknod()</code> , the mode of a file can be <code>S_IFLNK</code> if the file is a symbolic link, or <code>S_IFSOCK</code> if a socket descriptor is involved.                        |
| <code>st_ino</code>     | Uniquely identifies the file in a given file system. The pair <code>st_ino</code> and <code>st_dev</code> uniquely identifies regular files.                                                                                                                                                        |
| <code>st_dev</code>     | Uniquely identifies the file system that contains the file.                                                                                                                                                                                                                                         |
| <code>st_rdev</code>    | May be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.                                                                                                                        |
| <code>st_nlink</code>   | May be used only by administrative commands.                                                                                                                                                                                                                                                        |
| <code>st_uid</code>     | The user ID of the file's owner.                                                                                                                                                                                                                                                                    |
| <code>st_gid</code>     | The group ID of the file's group.                                                                                                                                                                                                                                                                   |
| <code>st_size</code>    | For regular files, this is the address of the end of the file. It is undefined for block special or character special files.                                                                                                                                                                        |
| <code>st_atime</code>   | Time when file data was last accessed. Modified by the following system calls: <code>creat()</code> , <code>mknod()</code> , <code>pipe()</code> , <code>utime()</code> and <code>read()</code> .                                                                                                   |
| <code>st_mtime</code>   | Time when data was last updated. Modified by the following system calls: <code>creat()</code> , <code>mknod()</code> , <code>pipe()</code> , <code>utime()</code> and <code>write()</code> .                                                                                                        |
| <code>st_ctime</code>   | Time when the file status was last changed. Modified by the following system calls: <code>chmod()</code> , <code>chown()</code> , <code>creat()</code> , <code>link()</code> , <code>mknod()</code> , <code>pipe()</code> , <code>unlink()</code> , <code>utime()</code> and <code>write()</code> . |
| <code>st_blksize</code> | A hint as to the 'best' unit size for I/O operations. This field is not defined for block special or character special files.                                                                                                                                                                       |
| <code>st_blocks</code>  | The total number of physical blocks of size 512 bytes actually allocated on disk. This field is not defined for block special or character special files.                                                                                                                                           |

### *BS2000*

With BS2000 files the following elements of the `stat` structure are set:

|                             |                                                                                                                                                                                                                    |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>mode_t st_mode</code> | File mode containing access permissions and file type.<br><br>Access permissions: Here the Basic ACL is mapped to the file mode bits. The file mode bits are all 0 if the file does not have basic ACL protection. |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                              |                                                                                                                                                                                                                   |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                              | File type: Introduction of a new file type <code>S_IFDVSBS2=X'40000000'</code> . This type is, however, not disjoint to <code>S_IFPOSIXBS2</code> . The <code>S_ISDVSBS2(mode)</code> macro can be used to query. |
| <code>time_t st_atime</code> | Last access time as is usual in BS2000, but in seconds since 1.1.1970 UTC).                                                                                                                                       |
| <code>time_t st_mtime</code> | Last modification time.                                                                                                                                                                                           |
| <code>time_t st_ctime</code> | Creation time.                                                                                                                                                                                                    |
| <code>long st_blksize</code> | Block size, 2K (i.e. 1 PAM page).                                                                                                                                                                                 |
| <code>long st_blocks</code>  | Number of blocks on the disk that are occupied by the file.                                                                                                                                                       |
| <code>dev_t st_dev</code>    | Contains the 4-byte catalog ID.                                                                                                                                                                                   |

The two consecutive fields

`uid_t st_uid` and  
`gid_t st_gid` contain the 8-byte BS2000 user ID.

All other fields are set to 0.

#### Return value

0 if successful.  
-1 if an error occurs; for POSIX files `errno` is set to indicate the error.

#### Errors

`fstat()` will fail if:

`EBADF` *fdes* is not a valid file descriptor.  
`EFAULT` *buf* points to an invalid address.  
`EIO` An I/O error occurred while reading the file system.  
`ENOLINK` *fdes* refers to a remote computer, whereby the connection to this computer is not active anymore.  
`EOVERFLOW` A component is too large and cannot be stored in the structure pointed to by *buf*.  
`EINTR` A signal was caught during the `fstat()` system call.

#### Notes

`fstat()` is now executed for BS2000 files also.

#### See also

`chmod()`, `chown()`, `creat()`, `link()`, `lstat()`, `mknod()`, `stat()`, `unlink()`, `write()`, `sys/stat.h`, `sys/types.h`.

## fstatvfs, statvfs - read file system information

**Name** fstatvfs, fstatvfs64, statvfs, statvfs64

**Syntax** #include <sys/statvfs.h>  
#include <sys/types.h>

```
int fstatvfs (int fildev, struct statvfs *buf);
int statvfs (const char *path, struct statvfs *buf);
int fstatvfs64 (int fildev, struct statvfs64 *buf);
int statvfs64 (const char *path, struct statvfs64 *buf);
```

**Description** fstatvfs() returns information on the file system to which the file identified by *fildev* belongs. *buf* is a pointer to a structure that is described below. The information on the file system is entered in this structure during the system call.

*fildev* identifies an open file descriptor that is the result of a successful open(), creat(), dup(), fcntl() or pipe() system call. The type of the file system containing the file assigned to *fildev* is known to the operating system. Read, write or execute permissions for the specified file are not needed.

There is no difference in functionality between fstatvfs() / statvfs() and fstatvfs64() / statvfs64() except that fstatvfs64() and statvfs64() both return the file status in statvfs64 structure.

The statvfs structure pointed to by *buf* contains the following components:

```
ulong_t f_bsize;           /* Preferred block size of the file system */
ulong_t f_frsize;        /* Basic block size of the file system
                          (if supported) */
fsblkcnt_t f_blocks;     /* Total number of blocks on the file system
                          in units of f_frsize */
fsblkcnt_t f_bfree;      /* Total number of free blocks */
fsblkcnt_t f_bavail;     /* Number of available free blocks for a
                          non-system administrator */
fsfilcnt_t f_files;      /* Total number of files (inodes) */
fsfilcnt_t f_ffree;      /* Total number of free nodes */
fsfilcnt_t f_favail;     /* Number of inodes for a
                          non-system administrator */
ulong_t f_fsid;          /* File system ID (currently dev) */
char f_basetype[FSTYP SZ]; /* Type name of destination file system,
                          null-terminated */
ulong_t f_flag;          /* Bit mask of the options */
ulong_t f_namemax;       /* Maximum length of the file names */
char f_fstr[32];         /* File-system-specific string */
ulong_t f_filler[16];    /* Reserved for future extensions */
```

The `statvfs64` structure differs from the `statvfs` structure by the following components:

```
fsblkcnt64_t f_blocks
fsblkcnt64_t f_bfree
fsblkcnt64_t f_bavail
fsfilcnt64_t f_files
fsfilcnt64_t f_ffree
fsfilcnt64_t f_favail
```

`f_basetype` contains a null-terminated type name of the file system (FST name) above the mounted destination (e.g. `s5` mounted above `rfs` results in `s5`).

The following values can be returned in the `f_flag` component:

```
ST_RDONLY      0x01    /* Write-protected file system */
ST_NOSUID      0x02    /* setuid/setgid semantics are not supported */
ST_NOTRUNC     0x04    /* Does not truncate file name longer than NAME_MAX*/
```

`statvfs()` works in the same way as `fstatvfs()`, except that the file is addressed via the pathname referenced by *path*. Search authorization is required for every directory in the pathname.

#### Return value

```
0                if successful.
-1              if an error occurs. errno is set to indicate the error.
```

#### Errors

`fstatvfs()` and `statvfs()` will fail if:

```
EIO              An I/O error occurs during reading of the file system.
EINTR           A signal was received during execution of the function.
```

`fstatvfs()` will fail if:

```
EBADF           fildev is not an open file descriptor.
EOVERFLOW       One of the values returned cannot be represented correctly in the structure
                 pointed to by buf, .
```

`statvfs()` will fail if:

```
EACCES         No search authorization exists for a component of the path prefix.
ELOOP          Too many symbolic links were encountered in resolving path.
ENAMETOOLONG   The pathname to which path points is longer than {PATH_MAX}, or the
                 length of a component of the pathname exceeds {NAME_MAX}.
```

ENOENT        A component of the pathname does not exist, or *path* points to an empty string.

ENOTDIR       A component of the path prefix of *path* is not a directory.

`statvfs()` will fail if:

ENAMETOOLONG

The resolving of symbolic links in the pathname leads to an interim result whose length exceeds `{PATH_MAX}`.

**Notes**        Not all elements of the `statvfs` structure are used in all file systems.

**See also**     `chmod()`, `chown()`, `creat()`, `dup()`, `exec`, `link()`, `mknod()`, `pipe()`, `read()`, `time()`, `unlink()`, `utime()`, `write()`, `sys/statvfs.h`.



## fsync - synchronize changes to file

**Syntax**      `#include <unistd.h>`  
`int fsync(int fil-des);`

**Description** `fsync()` causes all the modified data and attributes of *fil-des* that are still in the buffer to be written to the physical storage medium.

**Return value**

0                    if successful  
-1                    if an error occurs; `errno` is set to indicate the error.

**Errors**          `fsync()` will fail if:

`EBADF`            *fil-des* is not a valid file descriptor.  
`EINTR`            A signal was caught during the `fsync()` system call.  
`EINVAL`           *fil-des* refers to a file on which this operation is not possible.  
                    An attempt was made to access a BS2000 file.  
`EIO`                An I/O error occurred while reading from or writing to the file system.

**Notes**            `fsync()` should be used by programs which require modifications to a file to be completed before continuing; for example, a program which contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction are recorded.  
`fsync()` is executed only for POSIX files.

**See also**        `unistd.h`.

## ftell - get current value of file position indicator for stream

**Name**        **ftell, ftell64, ftello, ftello64**

**Syntax**      `#include <stdio.h>`

```
long int ftell(FILE *stream);
long long ftell64(FILE *stream);
off_t ftello(FILE *stream);
off64_t ftello64(FILE *stream);
```

**Description** `ftell()` and `ftello()` obtain the current value of the file position indicator for the stream pointed to by `stream`. This value can be used for positioning with `fseek()/fseeko()`.

The `ftello()` function is the same as the modified `ftell()` function except that the offset argument is of type `off_t` and that the `EOverflow` error has changed.

There is no difference in functionality between `ftell()` and `ftell64()` except that `ftell64()` uses the offset type `long long`.

`ftello64()` defined like `ftello()` except that `ftello64()` uses the offset type `off64_t`.

**Return value**

If successful, current value of the file position indicator for the stream, i.e. the number of bytes that offsets the file position indicator from the beginning of the file.

-1L            if an error occurs; `errno` is set to indicate the error.

*BS2000*

For binary files, current value of the file position indicator i.e. the number of bytes that offsets the file position indicator from the beginning of the file, if successful.

For text files, absolute position of the file position indicator if successful.

-1            if an error occurs; `errno` is set to `ERANGE` if the file position cannot be represented in 4 bytes.

**Errors**        `ftell()` and `ftello()` will fail if:

`EBADF`        The file descriptor underlying `stream` is not open for writing or the stream's buffer needed to be flushed and the file is not open.

`ESPIPE`       The file descriptor underlying `stream` is associated with a pipe or FIFO.

`EOverflow`    For `ftell()`: the resulting file offset is a value that cannot be represented correctly in an object of type `long`.

**E\_OVERFLOW** For *ftello()*: the current file offset cannot be represented correctly in an object of type `off_t`.

**Notes** The program environment determines whether `ftell()` / `ftello()` is executed for a BS2000 or POSIX file.

*BS2000*

`ftell()` can be used on both binary files (SAM in binary mode, PAM, INCORE) as well as text files (SAM in text mode, ISAM).

`ftell()` cannot be used for system files (SYSDTA, SYSLST, SYSOUT).

**See also** `fopen()`, `fseek()`, `lseek()`, `stdio.h`.

## ftime - get date and time

**Syntax**      `#include <sys/timeb.h>`  
                  `int ftime(struct timeb *tp);`

**Description** `ftime()` enters in the structure pointed to by `tp` the exact time in milliseconds since January 1, 1970, 00:00:00.

`tp` is a pointer to a structure that is defined in `sys/timeb.h` as follows.

```
struct timeb {
    time_t time;           /* Share of seconds      */
    unsigned short millitim; /* Share of milliseconds */
    short timezone;       /* Not supported         */
    short dstflag;        /* Not supported         */
};
```

The `timezone` and `dstflag` values are always zero. In other words, `ftime()` cannot be used to determine the local time zone and the setting for daylight saving time.

### *BS2000*

`ftime()` returns, in a structure, the same time as `time` (current local time as the number of seconds elapsed since January 1, 1950 00:00:00) and also includes milliseconds.

For portability reasons, additional options for `ftime()` have been included in the structure. However, they are not supplied in the BS2000 environment. □

**Return value**

|    |                     |
|----|---------------------|
| 0  | if successful.      |
| -1 | if an error occurs. |

**Notes**      Portable applications should use the `memcmp()` function instead of `ftime()`.

Depending on the resolution of the system clock, as a rule the value in `millitim` is not accurate to the last millisecond. Applications that depend on a particular level of precision in `millitim` are therefore not portable.

`ftime()` cannot be used together with the external variable `timezone` in a source file.

The variable `_TIMEZONE_STRUCT` must be set by means of a `DEFINE` at compilation.

### *BS2000*

The memory space for the result structure must be supplied explicitly!

The type `time_t` is defined in `sys/types.h`.

From the following structure components, only the `time` and `millitim` components are provided with values in the BS2000 environment. The other components are included in the structure only for portability reasons:

`time`: Time in seconds since January 1, 1950 00:00:00

`millitim`: Specification in milliseconds (0 to 999) to increase the precision of `time`.

`timezone`: Local time zone, measured in minutes west of Greenwich (not supported).

`dstflag`: Flag for daylight saving time (not supported). □

**See also** `ctime()`, `gettimeofday()`, `time()`, `sys/timeb.h`.

## ftok - interprocess communication

Syntax `#include <sys/ipc.h>`

```
key_t ftok(const char *path, int id);
```

Description `ftok()` returns a key which is based on *path* and *id* and can be used in subsequent `msgget()`, `semget()` and `shmget()` system calls. *path* must be the pathname of an existing file which can be accessed by the process. *id* is a character which uniquely identifies a project.

For all *path* pointers with which the same file is addressed, `ftok()` returns the same key if it is called with the same ID *id*.

`ftok()` returns different keys if different IDs *id* are specified or if various files which are located in the same file system at the same time are addressed via *path*. As a rule, `ftok()` does not return the same key if it is called up again with the same *path* and *id* arguments but the file thus identified was in the meantime deleted and then created again with the same name.

Only the 8 least-significant bits of *id* are used. If these bits are zero, the behavior of `ftok()` is undefined.

Return value

Key of type `key_t` if successful.

(`key_t`) -1 if an error occurs. `errno` is set to indicate the error.

Errors `ftok()` will fail if:

`EACCES` No search authorization exists for a component of the path prefix.

`ELOOP` Too many symbolic links were encountered in resolving *path*.

`ENAMETOOLONG`

The pathname pointed to by *path* is longer than `{PATH_MAX}`, or the length of a component of the pathname exceeds `{NAME_MAX}`; or the resolving of symbolic links in the pathname leads to an interim result whose length exceeds `{PATH_MAX}`.

`ENOENT` A component of the pathname does not exist, or *path* points to an empty string.

`ENOTDIR` A component of the path prefix of *path* is not a directory.

Notes To achieve maximum portability, the ID should occupy the least-significant byte in *id*. The remaining bytes should be set to 0.

See also `msgget()`, `semget()`, `shmget()`, `sys/ipc.h`.

## ftruncate, truncate - set file to specified length

**Name** ftruncate, ftruncate64, truncate, truncate64

**Syntax** #include <unistd.h>

```
int ftruncate (int fildes, off_t length);
int ftruncate64 (int fildes, off64_t length);
int truncate (const char *path, off_t length);
int truncate64 (const char *path, off64_t length);
```

**Description** ftruncate() sets the length of a normal file with the file descriptor *fildes* to *length* bytes. truncate() differs from ftruncate() only in that the file is addressed via a pointer *path* which references a pathname.

The effect of ftruncate() and truncate() on other types of file is undefined. If the file was previously longer than *length* bytes, the bytes after the position *length* can no longer be accessed. If the file was previously shorter, the bytes between the EOF mark before the call and the EOF mark after the call are padded with zeros. With ftruncate() the file must be opened for writing; with truncate() the effective user ID of the process must have write permission for the file.

If the request would cause the file size to exceed the current limit defined for the process for the maximum length of a file, the function is not executed and the system sends the SIGXFSZ signal to the process.

These functions do not change the current position in the file. On successful execution, if the file size was changed, these functions update the st\_ctime and st\_mtime fields of the file. The S\_ISUID and S\_ISGID bits of the file mode may be deleted.

There is no difference in functionality between ftruncate()/ truncate() and ftruncate64()/ truncate64() except that for ftruncate64() and truncate64() the length is specified as offset type off64\_t.

**Return value**

0 if successful.  
-1 if an error occurs. errno is set to indicate the error.

**Errors** ftruncate() and truncate() will fail if:

EINTR A signal was received during execution.  
EINVAL The value of *length* is negative.  
EFBIG or EINVAL The value of *length* is greater than the maximum permissible file size.

|                           |                                                                                                                                             |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| EIO                       | An I/O error occurred when reading from or writing to the file system.                                                                      |
| ftruncate() will fail if: |                                                                                                                                             |
| EBADF or EINVAL           | <i>fdes</i> is not a file descriptor that is opened for writing.                                                                            |
| EINVAL                    | <i>fdes</i> identifies a file that was opened for reading only.                                                                             |
| truncate() will fail if:  |                                                                                                                                             |
| EACCES                    | No search authorization exists for a component of the path prefix or no write authorization exists for the file addressed via <i>path</i> . |
| EISDIR                    | The file addressed via <i>path</i> is a directory.                                                                                          |
| ELOOP                     | Too many symbolic links were encountered in resolving <i>path</i> .                                                                         |
| ENAMETOOLONG              | The length of a component of the pathname exceeds {NAME_MAX} bytes, or the length of the pathname exceeds {PATH_MAX} bytes; or              |
| ENOENT                    | Either a component of the path prefix does not exist, or <i>path</i> references an empty string.                                            |
| ENOTDIR                   | A component of the path prefix from <i>path</i> is not a directory.                                                                         |
| EROFS                     | The file addressed via <i>path</i> resides on a read-only file system.                                                                      |
| truncate() will fail if:  |                                                                                                                                             |
| ENAMETOOLONG              | The resolving of symbolic links in the pathname leads to an interim result whose length exceeds {PATH_MAX}.                                 |

See also `open()`, `unistd.h`.

## ftrylockfile - lock standard input/output

Syntax `#include <stdio.h>`  
`int ftrylockfile(FILE *file);`

Description See `flockfile()`.



## ftw - traverse (walk) file tree

Syntax `#include <ftw.h>`

```
int ftw(const char *path, int (*fn)(const char *, const struct stat *ptr, int flag), int ndirs);
```

Description `ftw()` recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, `ftw()` calls the function pointed to by *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a `stat` structure (see also `sys/stat.h`) containing information about the object, and an integer. The possible values of the integer are defined in the `ftw.h` header. These are:

`FTW_F` for a file

`FTW_D` for a directory

`FTW_DNR` for a directory that cannot be read

`FTW_NS` for an object on which `stat()` could not successfully be executed

If the integer is `FTW_DNR`, descendants of that directory will not be processed. If the integer is `FTW_NS`, the `stat` structure will contain undefined values. An example of an object that would cause `FTW_NS` to be passed to the function pointed to by *fn* would be a file in a directory with `read` but without `execute` (search) permission.

`ftw()` visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a non-zero value, or some error is detected within `ftw()`.

*ndirs* specifies the maximum number of directory streams and/or file descriptors or both available for use by `ftw()` while traversing the tree. When `ftw()` returns, it closes any directory streams and file descriptors it uses not counting any opened by the *fn* function of the user.

Return value

0 if successful, i.e. when the file tree is exhausted. `ftw()` returns the result of the function pointed to by *fn*.

-1 if an error occurs; `errno` is set to indicate the error.

If the function pointed to by *fn* returns a non-zero value, `ftw()` stops its tree traversal and returns whatever value was returned by the function pointed to by *fn*. If `ftw()` detects an error, it returns -1 (see above).

If the function pointed to by *fn* detects a system error, `errno` can be set to that error value.

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Errors           | <code>ftw()</code> will fail if:                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| EACCES           | Search permission is denied for any component of <i>path</i> or read permission is denied for <i>path</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <i>Extension</i> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| EBADF            | An attempt was made to access a BS2000 file. □                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| ENAMETOOLONG     | The length of the <i>path</i> argument exceeds <code>{PATH_MAX}</code> or a pathname component is longer than <code>{NAME_MAX}</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| ENOENT           | <i>path</i> points to the name of a file that does not exist or to an empty string.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| ENOTDIR          | A component of <i>path</i> is not a directory.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Notes            | <p>Since <code>ftw()</code> is recursive, it is possible for it to terminate with a memory error when applied to very deep file structures.</p> <p><code>ftw()</code> uses <code>malloc()</code> to allocate dynamic storage during its operation. If <code>ftw()</code> is forcibly terminated, such as by <code>longjmp()</code> or <code>siglongjmp()</code> being executed by the function pointed to by <i>fn</i> or a signal-handling routine, <code>ftw()</code> will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function pointed to by <i>fn</i> return a non-zero value at its next invocation.</p> <p><code>ftw()</code> is executed only for POSIX files.</p> |
| See also         | <code>longjmp()</code> , <code>malloc()</code> , <code>siglongjmp()</code> , <code>stat()</code> , <code>ftw.h</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

## funlockfile - unlock standard input/output

Syntax `#include <stdio.h>`  
`void funlockfile(FILE *file);`

Description See `flockfile()`.

## fwide - specify file orientation

|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                          |                                                                                    |                          |                                                                          |                       |                                                  |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|------------------------------------------------------------------------------------|--------------------------|--------------------------------------------------------------------------|-----------------------|--------------------------------------------------|
| Syntax                   | <pre>#include &lt;stdio.h&gt; #include &lt;wchar.h&gt;  int fwide(FILE *dz, int mode);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |                          |                                                                                    |                          |                                                                          |                       |                                                  |
| Description              | <p><code>fwide()</code> specifies the orientation of the file with the file pointer <code>dz</code> as long as this file does not have an orientation. If the orientation has already been specified, for example by a previous I/O operation, then <code>fwide()</code> does not change this orientation.</p> <p><code>fwide()</code> attempts to set the orientation depending on the <code>mode</code> argument in the following manner:</p> <table><tr><td><code>mode &gt; 0</code></td><td>File is wide character-oriented.</td></tr><tr><td><code>mode &lt; 0</code></td><td>File is byte-oriented.</td></tr><tr><td><code>mode = 0</code></td><td>The file orientation is not changed.</td></tr></table> | <code>mode &gt; 0</code> | File is wide character-oriented.                                                   | <code>mode &lt; 0</code> | File is byte-oriented.                                                   | <code>mode = 0</code> | The file orientation is not changed.             |
| <code>mode &gt; 0</code> | File is wide character-oriented.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                          |                                                                                    |                          |                                                                          |                       |                                                  |
| <code>mode &lt; 0</code> | File is byte-oriented.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                          |                                                                                    |                          |                                                                          |                       |                                                  |
| <code>mode = 0</code>    | The file orientation is not changed.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |                          |                                                                                    |                          |                                                                          |                       |                                                  |
| Return value             | <table><tr><td><code>&gt; 0</code></td><td>if <code>dz</code> is wide character-oriented after calling <code>fwide()</code>.</td></tr><tr><td><code>&lt; 0</code></td><td>if <code>dz</code> is byte-oriented after calling <code>fwide()</code>.</td></tr><tr><td><code>0</code></td><td>if <code>dz</code> does not have an orientation.</td></tr></table>                                                                                                                                                                                                                                                                                                                                                    | <code>&gt; 0</code>      | if <code>dz</code> is wide character-oriented after calling <code>fwide()</code> . | <code>&lt; 0</code>      | if <code>dz</code> is byte-oriented after calling <code>fwide()</code> . | <code>0</code>        | if <code>dz</code> does not have an orientation. |
| <code>&gt; 0</code>      | if <code>dz</code> is wide character-oriented after calling <code>fwide()</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                          |                                                                                    |                          |                                                                          |                       |                                                  |
| <code>&lt; 0</code>      | if <code>dz</code> is byte-oriented after calling <code>fwide()</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                          |                                                                                    |                          |                                                                          |                       |                                                  |
| <code>0</code>           | if <code>dz</code> does not have an orientation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                          |                                                                                    |                          |                                                                          |                       |                                                  |
| Notes                    | In this version of the C runtime system only 1 byte characters are supported as wide characters.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                          |                                                                                    |                          |                                                                          |                       |                                                  |

## fwprintf, swprintf, vfwprintf, vswprintf, vwprintf, wprintf - output formatted wide characters

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

          int fwprintf(FILE *dz, const wchar_t *format [, arglist]);

          #include <stdarg.h>
          #include <wchar.h>

          int vwprintf(const wchar_t *format, va_list arg);

          #include <wchar.h>

          int wprintf(const wchar_t *format [, arglist]);
          int swprintf(wchar_t *s, size_t n, const wchar_t *format [, arglist]);

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

          int vfwprintf(FILE *dz, const wchar_t *format, va_list arg);
          int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

**Description** These functions are used to format output.

`fwprintf()` prepares the arguments in *arglist* according to the specifications in the wide character string *format* and writes them to the file pointed to by the file pointer *dz*. `fwprintf()` returns when the end of *format* is reached.

`vwprintf()` is the same as the `fwprintf()` function with *dz* = `stdout` where the argument list is replaced by an argument of type `va_list` that must have been initialized by the `va_start` macro (possibly followed by `va_arg` calls). The function does not call the `va_end` macro.

`wprintf()` is the same as the `fwprintf()` function with *dz* = `stdout`.

`swprintf()` writes formatted output to the wide character string *s*. Otherwise `swprintf()` is the same as the `fwprintf()` function. A maximum of *n* wide characters are written, including the closing null character that is automatically appended when *n* > 0.

`vfwprintf()` is the same as the `fwprintf()` function where the argument list is replaced by an argument of type `va_list` that must have been initialized by the `va_start` macro (possibly followed by `va_arg` calls). The function does not call the `va_end` macro.

`vsprintf()` is the same as the `swprintf()` function where the argument list is replaced by an argument of type `va_list` that must have been initialized by the `va_start` macro (possibly followed by `va_arg` calls). The function does not call the `va_end` macro.

*format* is a wide character string composed of zero or more directives and wide characters:

- conversion specifications beginning with the percent character (%), each of which is associated with zero or more arguments in *arglist*. The results are undefined if fewer arguments are passed in *arglist* than are defined in *format*. If the number of arguments defined in *format* is greater than the arguments passed in *arglist*, the excess arguments are ignored.  
The arguments assigned to a directive are converted, formatted and written to the output stream according to the directive.
- Characters of type `wchar_t` (but not %) that can be copied directly to the output as is.
- white-space characters (see section "White-space characters" on page 84).

### Conversion specifications

Each conversion specification is introduced by the % character, after which the following appear in sequence:

- Zero or more **flags**, which modify the meaning of the conversion specification.
- An optional non-zero decimal number or an asterisk (\*) that specifies a minimum **field width**. If the converted value has fewer bytes than the field width, it will be padded to the field width with spaces on the left (or padded on the right if the left-adjustment flag "-" was specified).
- An optional **precision** that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x` and `X` conversions; the number of digits to appear after the radix character for the `e`, `E` and `f` conversions; the maximum number of significant digits for the `g` and `G` conversions or the maximum number of bytes to be printed from a string in `s` conversion. The precision takes the form of a period (.), followed by a decimal digit string or an asterisk (\*), where a null digit string (only "." specified) is treated as 0.
- An optional size modifier `h`, `l`, `ll` or `L` preceding a conversion character.
  - `l` before `c` means that an argument of type `wint_t` is to be converted;
  - `l` before `s`: means that an argument of type `wchar_t` (pointer to a wide character) is to be converted;
  - `h` before `d`, `i`, `o`, `u`, `x` or `X`: conversion of an argument of type `short int` or unsigned `short int` (the argument is extended according to the integer extension and its value is converted to a `short int` or unsigned `short int` before output);
  - `h` before `n`: conversion of an argument of type pointer to `short int`;
  - `l` before `d`, `i`, `o`, `u`, `x` or `X`: conversion of an argument of type `long int` or unsigned `long int`;
  - `l` before `n`: conversion of an argument of type pointer to `long int`;
  - `ll` before `d`, `i`, `o`, `u`, `x` or `X`: conversion of an argument of type `long long int` or

unsigned long long int;  
 ll before n: conversion of an argument of type pointer to long long int;  
 L before e, E, f, g or G: conversion of an argument of type long double.

If an h, l (ell) or L appears before any other conversion character, the behavior is undefined.

- A **conversion character** of type `wchar_t` that specifies the type of conversion to be applied (see the list below).

A field width, or precision, or both, may be indicated by an asterisk (\*). In this case the values are obtained from the argument list instead of from the format specification. The (integer) values specifying the field width, precision or both must appear in that order before the argument, if any, to be converted.

A negative field width is taken as a "-" flag followed by a positive field width. A negative precision is taken as if the precision were omitted.

Conversion specifications have the following structure:

$$\% \quad [ - ] [ + ] [ - ] [ \# ] [ 0 ] \left[ \begin{array}{c} n \\ * \end{array} \right] [ . \left[ \begin{array}{c} m \\ * \end{array} \right] \left\{ \begin{array}{l} [ \{ h | l | ll \} \{ d | i | o | u | x | X \} \\ [ \{ h | l | ll \} ] n \\ [ L ] \{ e | E | f | g | G \} \\ [ ] \{ c | s | p \} \\ \{ D | O | U | C | S \} \\ \% \end{array} \right. \right\}$$

1.

2.

3.

4.

5.

1. Start of a conversion specification
2. Flags
3. Field width
4. Precision
5. Characters that define the actual conversion

**Flags**

- The result of the conversion will be left-justified within the array.
- +
- The result of a signed conversion will always begin with a sign (+ or -).
- If the first wide character of a signed conversion is not a sign or the result of a signed conversion is not a wide character, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.
- # This flag specifies that the value is to be converted to an alternative form. This flag has no effect for c, d, i, s and u. For o conversion, it increases the precision to force the first digit of the result to be 0. For x or X conversions, a non-zero result will have the string "0x" (or "0X") prefixed to it. For e, E, f, g or G conversions, the result will always contain a wide radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will not be removed from the result as they normally are. For other conversions, the behavior is undefined.
- 0 For d, i, o, u, x, X, e, E, f, g and G conversions, leading zeros (following any indication of sign or base) are used to pad to the array width; no space padding is performed. If the 0 and – flags both appear, the 0 flag will be ignored.  
For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag will be ignored. For other conversions, the behavior is undefined.

**Conversion characters**

- d, i The `int` argument is converted to a signed decimal in the style `[-]dddd`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1.  
The result of converting 0 with an explicit precision of 0 is no characters.
- o, u The `unsigned int` argument is converted to unsigned octal format (o) or in an unsigned decimal number (u) in the style `dddd`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1.  
The result of converting 0 with an explicit precision of 0 is no characters.
- x, X The `unsigned int` argument is converted to unsigned hexadecimal format in the style `dddd`; the letters `abcdef` (for x) or `ABCDEF` (for X) are used in addition to the digits. The precision specifies the minimum number of digits

- to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1.  
The result of converting 0 with an explicit precision of 0 is no characters.
- f      The `double` argument is converted to decimal notation in the style `[-]ddd.ddd`, where the number of digits after the radix character is equal to the precision specification.  
If the precision is missing, it is taken as 6.  
If the precision is explicitly 0 and no `#` flag is present, no radix character appears.  
If a radix character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.
- e, E    The `double` argument is converted in the style `[-]d.ddde+-dd`, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6.  
If the precision is 0 and no `#` flag is present, no radix character appears. The value is rounded to the appropriate number of digits.  
The `E` conversion character will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0.
- g, G    The `double` argument is converted in the style `f` or `e` (or in the style `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1.  
The style used depends on the value converted; style `e` (or `E`) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit.
- c      If the character “1” precedes it, the argument is converted from type `wint_t` to type `wchar_t`, the resulting character is written.  
If not preceded by a `1`, the argument is converted from type `int` to a wide character, just like for the `btowc()` function call. The resulting character is written.
- s      If not preceded by a `1`, the argument is of type pointer to a `char` array. Characters in the array are converted in the same manner as when the `mbrtowc()` function is called. The conversion status is written to an object of type `mbstate_t` and initialized to 0 before the first multi-byte character is converted. Data is written up to the terminating null character (and only to there).



If preceded by a `l`, the argument is of type pointer to a `wchar_t` array. Wide characters from the array are written up to the terminating null character (and only to there).

If a precision `m` is specified, no more than `m` bytes are written. If the precision is not specified or is greater than the size of the array, the array must contain a wide character null byte (as a terminator).

|   |                                                                                                                                                                                                                 |
|---|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S | Same as <code>ls</code> .                                                                                                                                                                                       |
| C | Same as <code>lc</code> .                                                                                                                                                                                       |
| p | The argument must be a pointer to <code>void</code> . The value is output as an 8-digit hexadecimal number.                                                                                                     |
| n | The argument must be a pointer to <code>int</code> into which is written the number of bytes written to the output so far by this call to one of the <code>fwprintf</code> functions. No argument is converted. |
| % | The wide character <code>%</code> is output; no argument is converted. The complete conversion specification must be of the form <code>%%</code> .                                                              |

If the character that follows `%` is not a valid conversion character, the result of the conversion is undefined.

If an argument is a `UNION` or a pointer to a `UNION`, the result of the conversion is undefined.

The same applies when an argument is an array or a pointer to an array, except for the following three cases:

The argument is an array of type `char` and uses `%s`,

The argument is an array of type `wchar_t` and uses `%ls` or

The argument is a pointer and uses `%p`.

A non-existent array width or a missing array width will never result in the truncation of an array. If the result of a conversion is wider than the array width, the array is simply extended to accept the output.

#### Return value

Number of wide characters output if successful.

Negative value if an error occurs.

#### Notes

In this version of the C runtime system, only 1-byte characters are supported as wide characters.

#### See also

`btowc()`, `fprintf()`, `mbrtowc()`, `printf()`

## fwscanf, swscanf, wscanf - formatted read

Syntax

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

int fwscanf(FILE *dz, const wchar_t *format [, arglist]);

#include <wchar.h>

int swscanf(const wchar_t *s, const wchar_t *format [, arglist]);
int wscanf(const wchar_t *format [, arglist]);
```

Description These functions are used for formatted input. They read the input, convert it according to the specifications in the format string *format* and store the result in the area that was specified in the optional argument list *arglist*.

*fwscanf()* reads formatted input from the file pointed to by *dz*.

*swscanf()* reads formatted input from the wide character string *s*. *swscanf()* is the same as the *fwscanf()* function otherwise. The end of the wide character string is EOF.

*wscanf()* reads formatted input from the standard input *stdin*. *wscanf()* is the same as the *fwscanf()* function with *dz* = *stdin*.

*format* is a character string, beginning and ending in its initial shift state, if defined. It is composed of zero or more directives and may include the following three types of characters:

- characters of type `char` (but no white-space characters or `%`), which are simply copied to the output stream (1: 1).
- white-space characters, starting with a backslash (`\`) (see `iswspace()`).
- conversion specifications beginning with the percent character (`%`), each of which is associated with zero or more arguments in *arglist*. The results are undefined if fewer arguments are passed in *arglist* than are defined in *format*. If the number of arguments defined in *format* is greater than the arguments passed in *arglist*, the excess arguments are ignored.

The *wscanf()* functions read the input characters without converting them at first and stores them in a variable. If the input character does not match the character specified in *format*, input processing is aborted and the function returns. If the conversion is aborted because a wide character does not fit, then this character is left unread in the input stream.

## White-space characters

The control string *format* may include zero or more characters producing white space. These characters have no control function.

White-space characters in the input are treated as delimiters between input fields; they are not converted (see %c, %n and %[] for exceptions). Leading white space in the input is ignored.

## Conversion specifications

All forms of `fwscanf()` allow for the insertion of a language-dependent radix character in the input string. The radix character is defined in the program's locale (category LC\_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

Each conversion specification is introduced by the % character, after which the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional non-zero decimal integer that specifies the maximum **field width**.
- An optional size modifier `h`, `l`, or `L` indicating the size of the receiving object:
  - `l` before the conversion characters `c`, `s` and `[]`: The corresponding argument is a pointer to `wchar_t`.
  - `h` or `l` before `d`, `i` and `n`: The corresponding argument is a pointer to `short int (h)` or `long int (l)`.
  - `h` or `l` before `o`, `u` and `x`: The corresponding argument is a pointer to `unsigned short int (h)` or `unsigned long int (l)`.
  - `ll` before `d`, `i` and `n`: The corresponding argument is a pointer to `long long int`.
  - `ll` before `o`, `u` and `x`: The corresponding argument is a pointer to `unsigned long long int`.
  - `l` or `L` before `e`, `f` and `g`: The corresponding argument is a pointer to `double (l)` or `long double (L)`.
- If `h`, `l` or `L` is before any other conversion character, the behavior is undefined.
- A **conversion character** that specifies the type of conversion to be applied.

`fwscanf()` executes each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters is executed by reading input until no more valid wide characters can be read (EOF), or up to the first byte which is not a white-space character (which remains unread).

A directive that is an ordinary character is executed as follows. The next wide character is read from the input and compared with the wide character that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent wide characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification is executed in the following steps:

Input white-space characters are skipped, unless the conversion specification includes a `[` or one of the conversion characters `c` or `n`.

An item is read from the input, unless the conversion specification includes an `n` conversion character. An input item is defined as the longest sequence of input bytes (up to any specified maximum field width) which is an initial subsequence of a matching sequence. The first byte, if any, after the input item remains unread.

If the length of the input item is 0, the execution of the conversion specification fails; this condition is a matching failure, unless an error prevented input, in which case it is an input failure.

Except in the case of a `%` conversion character, the input item (or, in the case of a `%n` conversion specification, the count of input bytes) is converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure.

Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

Conversion specifications have the following format:

$$\{ \% \} [ \left. \begin{array}{c} m \\ * \end{array} \right] \left. \begin{array}{l} [ \{ h | l | L \} \{ d | i | o | n | u | x | X \} \\ [ ] \{ c | s \} \\ [ ] | L \{ e | E | f | g | G \} \\ \{ p \} \\ [ ] \{ [ \dots ] | [ ^ \dots ] \} \\ \% \end{array} \right\}$$

**Conversion characters**

- d** Matches an optionally signed decimal integer, whose format is the same as expected for `wcstol()` with the value 10 for *base*. The corresponding argument must be of type pointer to `int`.
- i** Matches an optionally signed decimal integer, whose format is the same as expected for `wcstol()` with the value 0 for *base*. The corresponding argument must be of type pointer to `int`.
- o** Matches an optionally signed octal integer, whose format is the same as expected for `wcstoul()` with the value 8 for *base*. The corresponding argument must be of type pointer to unsigned integer.
- u** Matches an optionally signed decimal integer, whose format is the same as expected for `wcstoul()` with the value 10 for *base*. The corresponding argument must be of type pointer to unsigned integer.
- x, X** Matches an optionally signed hexadecimal integer, whose format is the same as expected for `wcstoul()` with the value 16 for *base*. The corresponding argument must be of type pointer to unsigned integer.
- e, E, f, g, G** These conversion characters match an optionally signed floating-point number, whose format is the same as expected for `wcstod()`. The corresponding argument must be of type pointer to `float`.
- s** Matches a sequence of wide characters that are not white-space characters.  
The corresponding argument must be a pointer to the initial byte of a `wchar_t` array that is large enough to accept the sequence and a terminating null character byte, which will be added automatically.
- S** Reads a sequence of wide characters that are not white space characters. If `l` is not specified, the sequence is converted to a sequence of wide character codes in the same manner as `wcrtomb()`. The conversion status is written to an object of type `mbstate_t` and initialized to 0 before the first wide character is converted. Data is written up to the terminating null character. The corresponding argument must be a pointer to the first byte of an array of type `char`, which must be large enough to accept the sequence and a terminating null byte, which will be added automatically.  
If `l` is specified, the corresponding argument must be a pointer to the initial byte of a `wchar_t` array that is large enough to accept the sequence and a terminating null character byte, which will be added automatically.
- [** Matches a non-empty sequence of bytes from a set of expected bytes (the `scanset`).  
If `l` is not specified, the sequence is converted to a sequence of wide character codes in the same manner as `wcrtomb()`. The conversion status

is written to an object of type `mbstate_t` and initialized to 0 before the first wide character is converted. Data is written up to the terminating null character. The corresponding argument must be a pointer to the first byte of an array of type `char`, which must be large enough to accept the sequence and a terminating null byte, which will be added automatically.

If `l` is specified, the corresponding argument must be a pointer to the initial byte of a `wchar_t` array that is large enough to accept the sequence and a terminating null character byte, which will be added automatically.

The conversion specification includes all subsequent wide characters in the *format* string up to and including the matching right square bracket (`]`). The wide characters between the square brackets (the scanlist) comprise the scanset, unless the first wide characters after the left square bracket is a circumflex (`^`), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right square bracket.

As a special case, if the conversion specification begins with `[ ]` or `[ ^ ]`, the right square bracket is included in the scanlist, and the next right square bracket is the matching right square bracket that ends the conversion specification. If a `-` is in the scanlist and is not the first character nor the last character after `[` or `[ ^`, the behavior is undefined.

**c** Matches a sequence of wide characters of the number specified by the field width (or 1 if no field width is present).

If `l` is not specified, the sequence is converted to a sequence of wide character codes in the same manner as `wcrtomb()`. The conversion status is written to an object of type `mbstate_t` and initialized to 0 before the first wide character is converted. Data is written up to the terminating null character. The corresponding argument must be a pointer to the first byte of an array of type `char`, which must be large enough to accept the sequence. No terminating null byte is added.

If `l` is specified, the corresponding argument must be a pointer to the initial byte of a `wchar_t` array that is large enough to accept the sequence. No terminating null byte is added.

The normal skip over white-space characters is suppressed in this case; `%1s` should be used to read the next byte that is not a white-space character.

**p** Matches a set of sequences, which must be the same as the set of sequences that is produced by the `%p` conversion of the `fwprintf` functions. The corresponding argument must be a pointer to a pointer to `void`. The interpretation of the input item is implementation-dependent; if the input item is not a value that was converted earlier during the same program execution, the behavior of the `%p` conversion is undefined. This is specially true for pointer outputs generated by other systems.

- `n` No input is processed. The corresponding argument must be a pointer to an `int` into which the number of input wide characters read thus far by this call are to be entered. Execution of a `%n` conversion specification does not increment the assignment count returned at the completion of execution of the function.
- `%` Matches a single `%`; no conversion or assignment occurs. The complete conversion specification must be `%%`.

If a conversion specification is invalid, the behavior of `fwscanf()` is undefined.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide characters matching the current conversion specification have been read (other than leading white-space characters, where permitted), execution of the current conversion specification terminates with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if different from `%n`) is terminated with an input failure.

Reaching the end of the string in a `swscanf()` call is equivalent to encountering the end-of-file indicator during an `fwscanf()` call.

Any trailing white space (including newline characters) is left unread unless matched by a conversion specification.

The success of literal matches and suppressed assignments cannot be directly determined, except via the `%n` conversion specification.

#### Return value

- Number of input elements read in and successfully assigned if no input error occurred before the first assignment.  
The number is null when a format error occurs in the first input element.
- EOF if an input error occurred before the first assignment.

**Notes** In this version of the C runtime system, only 1-byte characters are supported as wide characters.

**See also** `scanf()`, `sscanf()`, `fscanf()`, `wcstod()`, `wcstol()`, `wcstoul()`, `wcrtomb()`

## fwrite - output binary data

**Syntax**        `#include <stdio.h>`  
`size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);`

**Description** `fwrite()` writes, from the array pointed to by `ptr`, up to `nitems` elements whose size is specified by `size`, to the stream pointed to by `stream`. The file-position indicator for the stream (if defined) is advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is indeterminate.

The structure components `st_ctime` and `st_mtime` of the file are marked for changing between successful execution of `fwrite()` and the next successful completion of a call to `fflush()` or `fclose()` for the same data stream or a call to `exit()` or `abort()` (see `sys/stat.h`).

*BS2000*

### Record I/O

- `fwrite()` writes a record to the file.
- For sequential files (SAM, PAM), the record is written at the current file position .
- For index-sequential files (ISAM), the record is written at the position corresponding to the key value in the record.
- Number of characters to be output:

$n$  is taken to be the total number of characters to be output, i.e.

$$n = \text{size} * \text{nitems}$$

- If  $n$  is greater than the maximum record length, only one record with the maximum record length is written. The remaining data is lost.
- If  $n$  is less than the minimum record length no record is written. The minimum record length is defined only for ISAM files and means that  $n$  must cover at least the area of the key in the record.
- If  $n$  is less than the record length when a record is written to a file with fixed record length, the record is padded with binary zeros at the end.
- When an existing record is updated in a sequential file (SAM, PAM),  $n$  must be equal to the length of the record to be updated. Otherwise, an error occurs. In PAM files, the record length is the length of a logical block.
- When an existing record is updated in an index-sequential file (ISAM),  $n$  need not be equal to the length of the record to be updated. In other words, a record can be shortened or lengthened.



- In ISAM files for which key duplication is permitted, it is not possible to perform a direct update on a record. Whenever a record with an existing key is written, a new record is written. The old record must be explicitly deleted.
- `fwrite()` produces the same return value as for stream I/O, i.e. the number of elements written in their entirety. For record I/O, it is best to use only an element length of 1, since the return value will then correspond to the length of the record written (without any record length field).  
In the case of a fixed record length, however, any required padding with binary zeros is not taken into account in the return value. □

**Return value**

- Number of elements successfully written  
if successful. This number may be less than *nitems* if a write error is encountered.
- 0  
if *size* or *nitems* is 0. The contents of the array and the state of the stream remain unchanged.  
if a write error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

**Errors** See `fputc()`.

**Notes** To ensure that *size* specifies the correct number of bytes for a data element, the `sizeof()` function should be used for the size of the data unit to which *ptr* points.

On output to files with stream I/O, data is not written immediately to the external file, but is stored in an internal C buffer (see section "Buffering streams" on page 77).

On output to text files, control characters for white space (`\n`, `\t`, etc.) are converted to their appropriate effect in accordance with the type of text file (see section "White-space characters" on page 84).

The program environment determines whether `fwrite()` is executed for a BS2000 or POSIX file.

**See also** `ferror()`, `fopen()`, `printf()`, `putc()`, `puts()`, `write()`, `stdio.h`, `sys/stat.h`.

## gamma - compute logarithm of gamma function

**Syntax**     `#include <math.h>`  
               `double gamma(double x);`  
               `extern int signgam;`

**Description** `gamma()` computes the mathematical gamma function for a given floating-point number  $x$ :

$$\int_0^{\infty} e^{-t} t^{x-1} dt$$

The sign of this value is stored as +1 or -1 in the internal C variable `signgam`. The `signgam` variable may not be defined by the user.

`gamma()` is not reentrant.

### Return value

`gamma(x)`     if successful.  
`HUGE_VAL`     if the correct value results in an overflow.  
                   `errno` is set to indicate the error.  
`HUGE_VAL`     if  $x$  is a non-positive integer.  
                   `errno` is set to indicate the error.

**Errors**     `gamma()` will fail if:

`ERANGE`     Overflow; the return value is too large.  
`EDOM`         $x$  is a non-positive integer.

**See also**    `lgamma()`, `math.h`.

## garbcoll - release memory space to system *(BS2000)*

Syntax        `#include <stdlib.h>`  
              `void garbcoll(void);`

Description The `calloc()`, `malloc()`, `realloc()` and `free()` functions comprise the C-specific memory management package. This package essentially consists of an internal free memory management facility.  
Memory released by `free()` is not returned to the system (RELM-SVC), but is acquired by this free memory management facility.  
All the memory request functions (`calloc()`, `malloc()`, `realloc()`) will first attempt to allocate the required memory via the free memory management facility and only then from the operating system (REQM-SVC).  
If no memory is available even from the system, the memory administered by the free memory management facility is returned (page-wise if possible) to the system (garbage collection).  
This garbage collection mechanism is effective in the address space  $\leq 2$  GB and can also be called explicitly with the `garbcoll()` function.

Notes         All memory areas which were previously released with `free()` and which can be combined to form free pages are returned to the system by `garbcoll()`.

See also      `calloc()`, `malloc()`, `realloc()`, `free()`.

## gcvt - convert floating-point number to string

Syntax        `#include <stdlib.h>`  
              `char *gcvt(double value, int ndigit, char *buf);`

Description See `ecvt()`.

## getc - get byte from stream

Syntax `#include <stdio.h>`

```
int getc(FILE *stream);
```

Description The `getc()` function is equivalent to `fgetc()`, except that if it is implemented as a macro it may evaluate `stream` more than once, so the argument should never be an expression with side effects.

`getc()` is defined both as a function and as a macro.

`getc(stdin)` is identical to `getchar()`.

The `getc_unlocked()` function is functionally equivalent to `getc()` except that it is not implemented as a thread-safe function. For this reason, it can only be safely used in a multi-threaded program if the thread that calls it owns the corresponding (FILE \*) object. This is the case after successfully calling the `flockfile()` or `ftrylockfile()` functions.

Return value

See `fgetc()`.

Errors

See `fgetc()`.

Notes

If the integer value returned by `getc()` is stored in a variable of type `char` and then compared against the integer constant `EOF`, the comparison may not succeed, because sign-extension of a variable of type `char` on widening to integer is machine dependent. Portable applications should therefore ensure that the return value of `getc()` is always stored in a variable of type `int`.

If a comparison such as:

```
while((c = fgetc(dz)) != EOF)
```

is used in a program, the variable `c` must always be declared as an `int` value.

Otherwise, if `c` were defined as a `char`, the `EOF` condition would never be satisfied for the following reason: `-1` is converted to the `char` value `0xFF` (i.e. `+255`); however, `EOF` is defined as `-1`.

Since `getc()` may be implemented as a macro, it may treat a `stream` argument with side effects incorrectly. In particular, `getc(*f++)` may not work as expected. The use of `getc()` is therefore not recommended in such situations; `fgetc()` should be used instead.

If `fgetc()` is reading from the standard input `stdin` in the POSIX environment, and `EOF` is the end criterion for reading, the `EOF` condition can be achieved by the following actions:

- ▶ on a block-special terminal: by entering the key sequence `@ @ d`
- ▶ on a character-special terminal: by entering `CTRL+D`

*BS2000*

If `fgetc()` is reading from the standard input `stdin` in the BS2000 environment, and EOF is the end criterion for reading, the EOF condition can be achieved by means of the following actions at the terminal:

1. by pressing the `[K2]` key.
2. by entering the system commands EOF and RESUME-PROGRAM. □

**Notes** The program environment determines whether `getc()` is executed for a BS2000 or POSIX file.

**See also** `fgetc()`, `putc()`, `putchar_unlocked()`, `stdio.h`.

## getc\_unlocked, getchar\_unlocked, putchar\_unlocked, putchar\_unlocked - standard I/O with explicit lock by the client

**Syntax**

```
#include <stdio.h>

int getc_unlocked(FILE *stream);

int getchar_unlocked(void);

int putc_unlocked(int c, FILE *stream);

int putchar_unlocked(int c);
```

**Description** The functions `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()` and `putchar_unlocked()` are functionally equivalent to the original versions `getc()`, `getchar()`, `putc()` and `putchar()` except that it is not implemented as a thread-safe function.

For this reason, it can only be safely used in a multithread program if the thread that calls it owns the corresponding (FILE \*) object. This is the case after successfully calling the `flockfile()` or `ftrylockfile()` functions.

**Return value** See `getc()`, `getchar()` (both in `getc()`), `putc()` and `putchar()` (both in `putc()`).

**See also** `getc()`, `putc()`, `flockfile()`, `pthread_intro()`, `stdio()`.

## getchar - get byte from standard input stream

Syntax `#include <stdio.h>`

```
int getchar(void);
```

Description The function call `getchar(void)` is equivalent to `getc(stdin)`, i.e. `getchar()` reads 1 byte from the standard input stream.

Return value

See `fgetc()`.

Errors

See `fgetc()`.

Notes

If the integer value returned by `getchar()` is stored into a variable of type `char` and then compared against the integer constant `EOF`, the comparison may never succeed, because sign-extension of a variable of type `char` on widening to integer is machine-dependent. Portable applications should therefore ensure that the return value of `getchar()` is always stored in a variable of type `int`.

If a comparison such as:

```
while((c = fgetc(dz)) != EOF)
```

is used in a program, the variable `c` must always be declared as an `int` value.

Otherwise, if `c` were defined as a `char`, the `EOF` condition would never be satisfied for the following reason: `-1` is converted to the `char` value `0xFF` (i.e. `+255`); however, `EOF` is defined as `-1`.

If `fgetc()` is reading from the standard input `stdin` in the POSIX environment, and `EOF` is the end criterion for reading, the `EOF` condition can be achieved by the following actions:

- ▶ on a block-special terminal: by entering the key sequence `@ @ d`
- ▶ on a character-special terminal: by entering `CTRL+D`

*BS2000*

If `fgetc()` is reading from the standard input `stdin` in the BS2000 environment, and `EOF` is the end criterion for reading, the `EOF` condition can be achieved by means of the following actions at the terminal:

1. by pressing the `K2` key.
2. by entering the system commands `EOF` and `RESUME-PROGRAM`. □

The program environment determines whether `getchar()` is executed for a BS2000 or POSIX file.

See also `fgetc()`, `getc()`, `stdio.h`.

## getchar\_unlocked - standard input with explicit lock by the client

Syntax      #include <stdio.h>  
            int getchar\_unlocked(void);

Description See `getc_unlocked()`.



## getcontext, setcontext - display or modify user context

Syntax      `#include <ucontext.h>`  
`int getcontext(ucontext_t *ucp);`  
`int setcontext(const ucontext_t *ucp);`

Description In conjunction with the functions defined in `makecontext()`, these functions serve to implement the change of context at user level between several control flows of a process.

`getcontext()` initializes the structure pointed to by `ucp` as the current user context of the calling process. The `ucontext_t` structure pointed to by `ucp` defines the user context and contains the contents of the machine register, the signal mask and the stack of the calling process.

`setcontext()` restores the user context pointed to by `ucp`. A successful `setcontext()` call does not return; the program execution continues at the position pointed to by the context structure of `setcontext()`. The context structure should be generated by a preceding `getcontext()` call or have been supplied by the system as the third argument to a signal handling routine (see `sigaction()`).

- If the context structure was generated with `getcontext()`, the program execution is resumed as if the corresponding call of `getcontext()` had returned.
- If the context structure was generated with `makecontext()`, the program execution is resumed with the function specified by `makecontext()`. If this function returns, the process is continued, like after a `setcontext()` call, with the `ucp` argument that was also the argument for `makecontext()`.
- If the `ucp` argument is passed to a signal handling routine, the program execution is continued with the next instruction after the one interrupted by the signal.

If the `uc_link` component from the `ucontext_t` structure pointed to by `ucp` has the value 0, this is a basic process and the process terminates when this context is terminated. The use of a `ucp` argument that was generated differently from the description above leads to unpredictable results.

If threads are used, then the function affects the process or a thread in the following manner:

- `getcontext()` gets the current user context of the calling thread.
- `setcontext()` sets the current user context of the calling thread.

## Return value

getcontext():

0               if successful.  
-1               if an error occurs.

setcontext():

does not return if successful.  
-1               if an error occurs.

## Notes

If a signal handling routine is executed, the user context is stored and a new context generated. If the process leaves the signal handling routine via `longjmp()`, the original context will not be restored and subsequent `getcontext()` calls are no longer reliable. Signal handling routines should therefore use `siglongjmp()` or `setcontext()`.

Portable applications should neither access nor modify the `uc_mcontext` component of the `ucontext_t` structure. A portable application cannot assume that `getcontext()` stores static data of the process in `ucp`, not even `errno`. Care must be taken when manipulating contexts.

## See also

`bsd_signal()`, `makecontext()`, `setjmp()`, `sigaction()`, `sigaltstack()`, `sigprocmask()`, `sigsetjmp()`, `ucontext.h`.

## getcwd - get pathname of current working directory

**Syntax**      `#include <unistd.h>`  
`char *getcwd(char *buf, int size);`

**Description** `getcwd()` returns a pointer to the current directory pathname. The value of *size* must be at least one greater than the length of the pathname to be returned.

If *buf* is not null, the pathname will be stored in the space pointed to by *buf*.

If *buf* is a null pointer, `getcwd()` will obtain *size* bytes of space by calling `malloc()`. In this case, the pointer returned by `getcwd()` may be used as the argument in a subsequent call to `free()`.

The current directory will correspond to the home directory so long as no call to `chdir()` is made. The home directory can be checked with `getpwuid()` or `getpwnam()`. Both functions return a structure that includes a pointer to the original working directory.

When a C program is started, the current directory is set to the home directory, as defined in the file `SYSSRPM`. If the environment variable `HOME` is defined for a C program, the home directory is set to that value.

If the directory entered in the file `SYSSRPM` does not exist, a slash (/) is returned.

### *BS2000*

If an SDF-P variable `SYSPSIX.HOME` exists, the `HOME` variable of the C programming environment is initialized with the value of the `SYSPSIX.HOME` variable. □

The current directory can be changed at any time by calling `chdir()`. The effect of a call to `chdir()` extends for the duration of the calling program. The home directory is not changed by the call.

**Return value**  
0                    if *size* is not large enough or an error occurs in a subordinate function. `errno` is set to indicate the error.

**Errors**      `getcwd()` will fail if:

|        |                                                                                                                  |
|--------|------------------------------------------------------------------------------------------------------------------|
| EACCES | The name of a parent directory could not be obtained because the directory could not be read.                    |
| EINVAL | <i>size</i> is equal to 0.                                                                                       |
| ENOMEM | Insufficient storage space is available.                                                                         |
| ERANGE | <i>size</i> is less than 0, or is greater than 0, but smaller than the length of the <code>pathname + 1</code> . |

**Notes**      `getcwd()` is executed only for POSIX files

**See also**    `malloc()`, `unistd.h`.

## getdate - convert time and date to user format

**Syntax**      `#include <time.h>`

`struct tm *getdate (const char *string);`

`extern int getdate_err;`

**Description** `getdate()` converts user-definable date and/or time specifications from *string* into a `tm`-structure. The structure declaration can be found in the `time.h` file (see also `ctime()`).

User-defined templates are used for dismantling and interpreting the input string. These templates are text files, which the user creates; they are specified via the `DATMSK` environment variable. Each line of the template represents an acceptable date and/or time specification, with some of the field descriptors which are also used by the `date` command being used here. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format. If the operation is successful, the `getdate()` function returns a pointer to a structure of type `tm`; otherwise, `NULL` is returned and the global variable `getdate_err` is set.

The following field descriptors are supported:

|                 |                                                                                                  |
|-----------------|--------------------------------------------------------------------------------------------------|
| <code>%%</code> | same as <code>%</code>                                                                           |
| <code>%a</code> | abbreviated weekday name                                                                         |
| <code>%A</code> | weekday name in full                                                                             |
| <code>%b</code> | abbreviated month name                                                                           |
| <code>%B</code> | month name in full                                                                               |
| <code>%c</code> | local date and time representation                                                               |
| <code>%d</code> | day of the month (01 - 31; the leading 0 is optional)                                            |
| <code>%e</code> | same as <code>%d</code>                                                                          |
| <code>%D</code> | date as <code>%m/%d/%y</code>                                                                    |
| <code>%h</code> | abbreviated month name                                                                           |
| <code>%H</code> | hour (00 - 23)                                                                                   |
| <code>%I</code> | hour (01 - 12)                                                                                   |
| <code>%m</code> | month number (01 - 12)                                                                           |
| <code>%M</code> | minute (00 - 59)                                                                                 |
| <code>%n</code> | same as <code>\n</code>                                                                          |
| <code>%p</code> | local equivalent of AM or PM                                                                     |
| <code>%r</code> | time as <code>%I:%M:%S %p</code>                                                                 |
| <code>%R</code> | time as <code>%H:%M</code>                                                                       |
| <code>%S</code> | second (00-61). Leap seconds are allowed, but the effects of using algorithms are unpredictable. |
| <code>%t</code> | insert tab                                                                                       |
| <code>%T</code> | time as <code>%H:%M:%S</code>                                                                    |
| <code>%w</code> | weekday number (0 - 6; Sunday = 0)                                                               |
| <code>%x</code> | local date representation                                                                        |

|    |                                                                                                                                                                                                                                                                                                      |
|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| %X | local time representation                                                                                                                                                                                                                                                                            |
| %y | year in current century (00 - 99)                                                                                                                                                                                                                                                                    |
| %Y | year as <code>ccyy</code> (e.g. 1997)                                                                                                                                                                                                                                                                |
| %Z | name of time zone, or no character if no time zone exists. If the time zone under %Z is not the one expected by <code>getdate()</code> , an input error occurs. <code>getdate()</code> computes an appropriate time zone based on the information passed to the function (e.g. hour, day and month). |

When comparing the template and the input specification, `getdate()` does not distinguish between upper and lowercase.

The month and weekday names can consist of any combination of lowercase and uppercase letters. The user can define that the specification of the input time or the input date is language-dependent by setting the values `LC_TIME` and `LC_CTYPE` in `setlocale()`.

The descriptors for which digits must be specified have at most two positions. Leading zeros are allowed but they can also be omitted. Blanks in the template or in *string* are ignored.

The field descriptors %c, %x and %X are rejected if they contain invalid field descriptors.

The following rules apply to the conversion of input specifications into the internal format:

- If %Z is specified, `getdate()` sets the elements of the `tm` structure to the current time of the specified time zone. Otherwise the formatted time is initialized with the current local time as if `localtime()` had been executed.
- If only the weekday is given, the current day is assumed if the specified weekday is identical to the current day. If the given day is before the current one, the weekday is taken from the next week.
- If only the month is specified, the current month is assumed if the specified month is the same as the current month. If the specified month is earlier than the current month, the next year is assumed if no year is otherwise specified. (The first day of the month is assumed if no day is specified.)
- If the hour, minute and second are not specified, the current hour, minute and second are taken.
- If no date is specified, the current day is assumed if the specified hour is later than the current one. If the specified hour is earlier than the current one, the next day is assumed.

`getdate()` uses the external variable or the `getdate_err` macro to return the error weight.

## Return value

Pointer to a `tm` structure  
if successful.

Null pointer if an error occurs. `getdate_err` is set to indicate the error.

## Errors

`getdate()` will fail if any of the following errors occur. The error weights are returned in `getdate_err`. The contents of `errno` are not significant here.

- 1 The `DATMSK` environment variable is undefined or zero.
- 2 The template file cannot be opened for reading.
- 3 The file status could not be read.
- 4 The template file is not a regular file.
- 5 An error occurred during reading of the template file.
- 6 `malloc()` could not be executed successfully, as there was not enough memory available.
- 7 There is no line in the template file which matches the input.
- 8 The input format is invalid, e.g. February 31, or a time was specified which cannot be represented in a `time_t` type; `time_t` contains the time in seconds since 00:00:00 UTC, which corresponds to January 1, 1970.

## Notes

The following `getdate()` calls modify the contents of `getdate_err`.

The declaration of the external variable `getdate_err` is contained in the header file `time.h`. `getdate_err` should therefore not be explicitly declared in the program; `time.h` should be inserted instead.

Dates before 1970 and after 2037 are invalid.

**Example 1** Possible contents of a template:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

**Example 2** Some examples of valid input specifications for the temple in example 1:

```
getdate("10/1/87 4 PM");
getdate("Friday");
getdate("Friday September 19 1987, 10:30:30");
getdate("24,9,1986 10:30");
getdate("at monday the 1st of december in 1986");
getdate("run job at 3 PM, december 2nd");
```

If the `LC_TIME` environment variable is set or if `LANG` is set to german, the following specification is valid:

```
getdate("Freitag den 10. Oktober 1986 10.30 Uhr");
```

**Example 3** Local time and date specifications are also supported. Example 3 shows how these can be defined in templates.

| Call                                     | Line in template file    |
|------------------------------------------|--------------------------|
| <code>getdate("11/27/86");</code>        | <code>%m/%d/%y</code>    |
| <code>getdate("27.11.86");</code>        | <code>%d.%m.%y</code>    |
| <code>getdate("86-11-27");</code>        | <code>%y-%m-%d</code>    |
| <code>getdate("Friday 12:00:00");</code> | <code>%A %H:%M:%S</code> |



**Example 4** The following examples clarify the above rules. It is assumed that the current date and time are Monday September 22, 1986, 12:19:47 EDT and that the LANG and LC\_TIME environment variables are not set.

| Input       | Line in template file | Date                         |
|-------------|-----------------------|------------------------------|
| Mon         | %a                    | Mon Sep 22 12:19:48 EDT 1986 |
| Sun         | %a                    | Sun Sep 28 12:19:49 EDT 1986 |
| Fri         | %a                    | Fri Sep 26 12:19:49 EDT 1986 |
| September   | %B                    | Mon Sep 1:19:49 EDT 1986     |
| January     | %B                    | Thu Jan 1:19:49 EST 1987     |
| December    | %B                    | Mon Dec 1:19:49 EST 1986     |
| Sep Mon     | %b %a                 | Mon Sep 1:19:50 EDT 1986     |
| Jan Fri     | %b %a                 | Fri Jan 2 12:19:50 EST 1987  |
| Dec Mon     | %b %a                 | Mon Dec 1:19:50 EST 1986     |
| Jan Wed 198 | %b %a %Y              | Wed Jan 4 12:19:51 EST 1989  |
| Fri 9       | %a %H                 | Fri Sep 26 09:00:00 EDT 1986 |
| Feb 10:30   | %b %H:%S              | Sun Feb 1 10:00:30 EST 1987  |
| 10:30       | %H:%M                 | Tue Sep 23 10:30:00 EDT 1986 |
| 13:30       | %H:%M                 | Mon Sep 22 13:30:00 EDT 1986 |

**See also** `ctime()`, `localtime()`, `setlocale()`, `strftime()`, `times()`, `time.h`.

## getdents - convert directory entries

**Name**        **getdents, getdents64**

**Syntax**      `#include <sys/dirent.h>`

```
int getdents(int fildes, struct dirent *buf, size_t nbyte);
int getdents64(int fildes, struct dirent64 *buf, size_t nbyte);
```

**Description** *fildes* is a file descriptor that is returned by a `open()` or `dup()` system call. `getdents()` attempts to read *nbyte* bytes from the directly associated with *fildes* and to place them in the buffer pointed to by *buf* as directory entries independent of the file system. Since the directory entries independent of the file system have different lengths, the actual number of bytes returned is much smaller than *nbyte* in most cases.

You look in `dirent()` (Reference Manual for System Administrators) to calculate the number of bytes.

The file system independent directory entries are specified using the `dirent` structure. You will find a description in `dirent()`.

For devices that can position, `getdents()` starts at the location in the file that is specified by the read/write pointer assigned to *fildes*. After returning from `getdents()`, the read/write pointer is incremented so that it points to the next directory entry. This system call was developed to implement the `readdir()` function (You will find a description in `directory()`) and should therefore not be used for any other purpose.

There is no difference in functionality between `getdents()` and `getdents64()` except that for `getdents64()` *buf* points to a `dirent64` structure.

**Errors**        The following descriptions of the error codes depend on the function. You will find a generally applicable description in `intro_prm2()` and in `errno()`.

`getdents()` and `getdents64()` are unsuccessful if one or more of the following arise:

EBADF            *fildes* is not a open and valid file descriptor for reading.

EFAULT           *buf* points beyond the assigned address space.

EINVAL           *nbyte* is not large enough for a directory entry.

ENOENT           The current read/write pointer for the directory does not point to a valid entry.

ENOLINK          *fildes* points to a remote computer and the connection to this computer is not active anymore.

ENOTDIR          *fildes* is not a directory.

EIO                An I/O error occurred while accessing the file system.

**Return value**

After successful completion, a non-negative integer that specifies the actual number of bytes read is returned. A return value of 0 means that the end of the directory was reached. If the system call failed, then -1 is returned and `errno` is set to indicate the error.

**See also**     `directory()`, `dirent()`.

## getdtablesize - get size of descriptor table

**Syntax**      `#include <unistd.h>`

```
int getdtablesize(void);
```

**Description** `getdtablesize()` is equivalent to the `getrlimit()` function if `RLIMIT_NOFILE` is specified.

`getdtablesize()` is not thread-safe.

**Return value**

Current limit for the number of simultaneously open file descriptors per process if successful.

-1 if an error occurs.

**Notes**      There is no direct relationship between the value returned by `getdtablesize()` and the `{OPEN_MAX}` constant defined in `limits.h`.

**See also**    `close()`, `getrlimit()`, `open()`, `select()`, `setrlimit()`, `limits.h`, `unistd.h`.

## getegid - get effective group ID of process

**Syntax**      `#include <unistd.h>`

*Optional*

```
#include <sys/types.h> □
```

```
gid_t getegid(void);
```

**Description** `getegid()` returns the effective group ID of the calling process.

**Return value**

Effective group ID of the calling process.

The function is always successful.

**See also**    `getgid()`, `setgid()`, `sys/types.h`, `unistd.h`, manual "POSIX Basics (BS2000/OSD)".

## getenv - get value of environment variable

Syntax `#include <stdlib.h>`

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

Description `getenv()` searches the current environment of the process, i.e. the string array pointed to by `environ`, for a string of the form "*name=value*" and returns a pointer to the string containing the *value* for the specified variable *name*.

`getenv()` is not thread-safe.

Return value

Value of *name*

if a corresponding string exists.

Null pointer

if no corresponding string exists,  
or if the application is called with BS2000 functionality (see section "Scope of the supported C library" on page 18).

Notes

The string "*name=value*" cannot be altered, but may be overwritten by subsequent `putenv` calls. Other library functions do not overwrite the string.

*BS2000*

The string array to which `environ` points can be initialized with values from the SDF-P variable `SYSPOSIX.name` on starting the program (see `environ` and the section "Environment variables" on page 71). □

See also

`exec`, `environ`, `putenv()`, `stdlib.h`, section "Scope of the supported C library" on page 18, and section "Environment variables" on page 71.

## geteuid - get effective user ID of process

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h> □`

`uid_t geteuid(void);`

Description `geteuid()` returns the effective user ID of the calling process.

Return value

Effective user ID of the calling process

The function is always successful.

See also `getuid()`, `setuid()`, `sys/types.h`, `unistd.h`, manual "POSIX Basics (BS2000/OSD)".

## getgid - get real group ID of process

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h> □`

`gid_t getgid(void);`

Description `getgid()` returns the real group ID of the calling process.

Return value

Real group ID of the calling process.

The function is always successful.

See also `getegid()`, `getuid()`, `setgid()`, `sys/types.h`, `unistd.h`, manual "POSIX Basics (BS2000/OSD)".

## getgrent - get group file entry

Syntax `#include <grp.h>`

`struct group *getgrent (void);`

Description See `endgrent()`.

## getgrgid - get group file entry for group ID

Syntax `#include <grp.h>`

*Optional*

`#include <sys/types.h> □`

`struct group *getgrgid(gid_t gid);`

Description `getgrgid()` searches the group file for an entry containing a *gr\_gid* component that matches *gid* (see `grp.h` and the manual "POSIX Basics (BS2000/OSD)").

`getgrgid()` is not thread-safe. Use the reentrant function `getgrgid_r()` when needed.

### Return value

Pointer to an object of the structure `group`  
if an entry with a *gr\_gid* component matching *gid* is found.

Null pointer if an error occurs or no entry with a *gr\_gid* component matching *gid* is found.  
`errno` is set to indicate the error.

Errors `getgrgid()` fails if:

EIO An I/O error occurred.

EINTR A signal was caught during the execution of `getgrgid()`.

EMFILE Too many file descriptors are currently open in the calling process.

ENFILE The file table of the system is currently full.

Notes The return value may point to a static area which may be overwritten by a subsequent call to `getgrgid()` or `getgrnam()`.

Since `getgrgid()` calls functions for file processing that may fail, `errno` should be set to 0 before the call to `getgrgid()`. If `errno` is set to some other value on return from the function, an error has occurred.

See also `getgrgid_r()`, `getgrnam()`, `grp.h`, `limits.h`, `sys/types.h`, and the manual "POSIX Basics (BS2000/OSD)".

## getgrgid\_r - get group file entry for group ID (thread-safe)

**Syntax**        `#include <grp.h>`  
`int getgrgid_r(gid_t gid, struct group *grp, char *buffer,  
                  size_t bufsize, struct group **result);`

**Description** The `getgrgid_r()` updates the group structure pointed to by *grp* and stores a pointer to this structure at the address pointed to by *result*. The structure contains the entry from the group file whose `gr_gid` component matches the *gid*. The structure found in the group file is copied to the memory area of length *bufsize* passed in the *buffer* parameter. The maximum size required for this buffer can be determined via the `sysconf()` parameter `{_SC_GETGR_R_SIZE_MAX}`. When an error occurs or when the desired entry could not be found, a null pointer is returned in the data area pointed to by *result*.

**Return value**  
0                    if successful.  
Error number    if an error occurs. `errno` is set to indicate the error.

**Errors**        The `getgrgid_r()` function fails if:  
ERANGE            The memory area pointed to by *buffer* of length *bufsize* is not large enough to hold the data pointed to by the resulting group structure.

**Notes**        Applications in which there are checks for error situations must set `errno` to 0 before calling `getgrgid_r()`. If `errno` is set to a value not equal to null when it returns, then an error occurred.

**See also**     `getgrgid()`, `getgrnam()`, `grp.h`, `limits.h`, `sys/types.h`.



## getgrnam - get group file entry for group name

Syntax `#include <grp.h>`

*Optional*

`#include <sys/types.h> □`

`struct group *getgrnam(const char *name);`

Description The `getgrnam()` function searches the group file for an entry containing a *gr\_name* component that matches *name* (see also `grp.h` and the manual "POSIX Basics (BS2000/OSD)").

`getgrnam()` is not thread-safe. Use the reentrant function `getgrnam_r()` when needed.

### Return value

Pointer to an object of the structure `group` (see `grp.h`) if successful.

Null pointer if an error occurs or no entry with a *gr\_name* component matching *name* is found. `errno` is set to indicate the error.

### Errors

The `getgrnam()` function fails if:

EIO An I/O error occurred.

EINTR A signal was caught during the execution of `getgrnam()`.

EMFILE Too many file descriptors are currently open in the calling process.

ENFILE The system file table is currently full.

### Notes

The return value may point to a static area which may be overwritten by a subsequent call to `getgrgid()` or `getgrnam()`.

To check for error situations, `errno` should be set to 0 before calling `getgrnam()`.

### See also

`getgrnam_r()`, `getgrgid()`, `grp.h`, `limits.h`, `sys/types.h`, "POSIX Commands" (BS2000/OSD) manual.

## getgrnam\_r - get group file entry for group name (thread-safe)

**Syntax**

```
#include <sys/types.h>
#include <grp.h>
int getgrnam_r(const char * name, struct group * grp, char * buffer,
               size_t bufsize, struct group ** result);
```

**Description** The `getgrnam_r()` updates the group structure pointed to by `grp` and stores a pointer to this structure at the address pointed to by `result`. The structure contains the entry from the group file whose `gr_name` component matches the `name`.

The structure found in the group file is copied to the memory area of length `bufsize` passed in the `buffer` parameter. The maximum size required for this buffer can be determined via the `sysconf()` parameter `{_SC_GETGR_R_SIZE_MAX}`.

When an error occurs or when the desired entry could not be found, a null pointer is returned in the data area pointed to by `result`.

**Return value**

0 if successful.

Error number if an error occurs. `errno` is set to indicate the error.

**Errors** The `getgrnam_r()` function fails if:

ERANGE The memory area pointed to by `buffer` of length `bufsize` is not large enough to hold the data pointed to by the resulting group structure.

**See also** `getgrnam()`, `getgrgid_r()`, `grp.h`, `limits.h`, `sys/types.h`, manual "POSIX Basics (BS2000/OSD)".

## getgroups - get supplementary group IDs

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h> □`

`int getgroups(int gidsetsize, gid_t grouplist[ ]);`

Description `getgroups()` determines the current supplementary group IDs of the calling process and stores the result in the array *grouplist*.

*gidsetsize* specifies the number of elements in the array *grouplist* and must be large enough to accept the complete list. This list cannot be greater than `{NGROUPS_MAX}`. The actual number of supplementary group IDs stored in the array is returned. The values of array entries with indices greater than or equal to the value returned are undefined.

If *gidsetsize* is 0, `getgroups()` returns the number of supplementary group IDs associated with the calling process without modifying the array pointed to by *grouplist*.

Return value

Number of supplementary group IDs

if successful. The return value is non-zero or less than the number of group IDs for the calling process.

-1 if unsuccessful. `errno` is set to indicate the error.

Errors

`getgroups()` fails if:

`EINVAL` The value of *gidsetsize* is non-zero or less than *gr\_number* for the calling process.

Notes The effective group ID of the calling process is included in *grouplist*.

See also `getegid()`, `getuid()`, `setgid()`, `sys/types.h`, `unistd.h`, and the manual "POSIX Basics (BS2000/OSD)".

## gethostid - get ID of current host

**Syntax**        `#include <unistd.h>`  
                 `long gethostid(void);`

**Description** `gethostid()` outputs a 32-bit ID for the current host. The ID is formed from the CPU serial number (3 bytes) and the VM ID (1 byte), so that several VMs in a system can be distinguished from each other.

**Return value**  
                 Unique ID for the current host  
                 if successful.

**See also**        `random()`, `unistd.h`.

## gethostname - get name of current host

**Syntax**        `#include <unistd.h>`  
                 `int gethostname(char *name, size_t namelen);`

**Description** `gethostname()` determines the default name of the current host. The *namelen* parameter specifies the size of the file pointed to by *name*. A trailing zero is appended to the name, provided *namelen* is long enough.  
If the host name exceeds the value *namelen*, the name is truncated and it is not guaranteed that a trailing zero will be appended.

**Return value**  
                 0                    if successful.  
                 -1                    otherwise.

**See also**        `gethostid()`, `unistd.h`.

## getitimer, setitimer - read or set

Syntax `#include <sys/time.h>`

```
int getitimer(int which, struct itimerval *value);
```

```
int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

Description The system offers each process three interval timers that are declared in the `sys/time.h` file. The `getitimer()` call stores the current value of the *which* timer in the structure to which *value* points. The `setitimer()` call sets the value of *which* to the value in the structure to which *value* points; if *ovalue* is not zero, the previous value of the timer is stored in the structure to which *ovalue* points.

The setting of a timer is defined via the `itimerval` structure (see `sys/time.h`), which contains at least the following components:

```
struct timeval  it_interval;    /* Clock interval */
struct timeval  it_value;       /* Current value */
```

If `it_value` is not zero, the time until the next expiry of the timer is specified. If `it_interval` is not zero, a value is specified to which `it_value` is set if the timer expires. If `it_value` is set to zero, the timer is deactivated, regardless of the value of `it_interval`. Setting `it_interval` to zero deactivates the timer after its next expiry (provided `it_value` is not zero).

If time values are smaller than the resolution of the system clock, they are rounded to the system clock's resolution.

Each process has three timers at its disposal which can be addressed via the following values for *which*:

`ITIMER_REAL`      decrements in real time. The `SIGALRM` signal is sent when this timer expires.

`ITIMER_VIRTUAL`    decrements in the virtual process time. This timer only runs when the process is executed. The `SIGVTALRM` signal is sent when this timer expires.

`ITIMER_PROF`      decrements in virtual process time, regardless of `ITIMER_VIRTUAL`. Whenever the `ITIMER_PROF` timer expires, the `SIGPROF` signal is sent. Because this signal interrupts system calls of the process, the programs which use this timer must be prepared to repeat the interrupted system calls.

`setitimer()` and `sleep()` or `usleep()` should not be used together, as this may result in undesirable interactions - in particular, a `sleep()` call signs on its own signal handling routine, so the signal handling routine of the user is not activated.

## Return value

- 0 if successful.
- 1 if an error occurs. `errno` is set to indicate the error.

## Errors

`setitimer()` will fail if:

- `EINVAL` The values to which the *value* argument points are invalid. (For the microseconds a non-negative integer lower than 1,000,000 must be specified, for the seconds a non-negative integer.)

`getitimer()` and `setitimer()` will fail if:

- `EINVAL` The *which* parameter was not recognized

## Notes

The field with the microseconds must not contain a value greater than or equal to one second.

## See also

`alarm()`, `sleep()`, `ualarm()`, `usleep()`, `signal.h`, `sys/time.h`.

## getlogin - get login name

Syntax `#include <unistd.h>`

*Optional*

`#include <stdlib.h>`

`char *getlogin(void);`

**Description** `getlogin()` returns a pointer to a string with the user name of the calling process (which corresponds to the login name of the calling process). 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.

`getlogin()` is not thread-safe. Use the reentrant function `getlogin_r()` when needed.

**Return value**

Pointer to the login name

The function is always successful.

Null pointer if unsuccessful, e.g. if `getlogin()` is called from within a process for which the login name cannot be found. `errno` is set to indicate the error.

**Notes**

The return value usually points to static data whose content is overwritten by each call. Portable applications should therefore save the login name elsewhere if it is required after a subsequent call to the same function.

Three names associated with the current process can be determined:

`getpwuid(geteuid())` returns the name associated with the effective user ID of the process; `getlogin()` returns the name associated with the current login activity; and `getpwuid(getuid())` returns the name associated with the real user ID of the process.

**See also**

`getlogin_r()`, `getpwnam()`, `getpwuid()`, `geteuid()`, `getuid()`, `limits.h`, `unistd.h`.

## getlogin\_r - get login name (thread-safe)

**Syntax**      `#include <unistd.h>`  
`int getlogin_r(char * name, size_t namesize);`

**Description** The `getlogin()` function writes the user name of the calling process (which corresponds to the login name of the calling process) in the data area pointed to by *name*. The data area is *namesize* characters long and should be large enough for the name and the terminating null character. The maximum size of the login name is `{ LOGIN_NAME_MAX }`.

If `getlogin()` is successful, then *name* points to the name that the user logged in under, even if there are several login names with the same user ID.

**Return value**  
0                    if successful.  
Error number    otherwise.

**Errors**            The `getlogin_r()` function fails if:  
ERANGE            The value of *namesize* is smaller than the length of the login name found including the terminating null character.

**See also**        `getlogin()`, `getpwnam_r()`, `getpwuid_r()`.



## getmsg - get message from STREAMS file

Syntax `#include <stropts.h>`

```
int getmsg(int fildev, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagsp);
```

```
int getpmsg(int fildev, struct strbuf *ctlptr, struct strbuf *dataptr, int *bandp, int *flagsp);
```

Description `getmsg()` fetches the contents of a message located in the read queue of the stream head of a STREAMS file, and writes them to a buffer specified by the user. The message contains either a data section, a control section or both. The data and control sections of the message are written to separate buffers, as described below. The semantics of the sections are defined via the STREAMS module which generated the message.

The `getpmsg()` function does the same as `getmsg()`, but it performs a more precise check on the priority of the messages received. Unless otherwise indicated, all information concerning `getmsg()` also applies to `getpmsg()`.

*fildev* specifies a file descriptor that points to an open stream.

*ctlptr* and *dataptr* each reference an `strbuf` structure which has the following elements:

```
int maxlen; /* Maximum buffer size */
int len;    /* Length of the data */
char *buf;  /* Pointer to the buffer */
```

*buf* points to a buffer to which the data or control information is to be written. `maxlen` denotes the highest possible number of bytes that this buffer can hold. On return, `len` contains the number of bytes of the data or control information that was actually received, or the value is 0 if the control or data section has a null length, or the value is -1 if a message does not contain any data or control information.

If `getmsg()` is called, *flagsp* should reference an integer which indicates the type of message the user can receive. This is described later.

*ctlptr* is used to receive the control section of the message and *dataptr* is used to receive the data section. If *ctlptr* (or *dataptr*) is zero or the `maxlen` field is -1, the control (or data) section of the message is not processed and remains in the read queue of the stream head. If *ctlptr* (or *dataptr*) is not zero and there is no corresponding control (or data) section of the message in the read queue of the stream head, `len` is set to -1. If the `maxlen` field is set to 0 and there is a control (or data) section with a null length, this null-length section is removed from the read queue and `len` is set to 0. If the `maxlen` field is set to 0 and there are more than 0 bytes of control (or data) information, this information remains in the read queue and `len` is set to 0. If the `maxlen` field in *ctlptr* or *dataptr* is smaller than the control or data section of the message, `maxlen` bytes will be fetched. In this case the remainder of the message is left in the read queue of the stream head and a non-zero return value is supplied (see return value).

By default `getmsg()` processes the first message available in the read queue. If the integer to which *flagsp* points is set to `RS_HIPRI`, the process only receives high-priority messages. In this case, `getmsg()` only processes the next message if it has high priority. If the integer referenced by *flagsp* is 0, `getmsg()` places each available message in the read queue of the stream head. In this case on return, the integer referred to by *flagsp* is set to `RS_HIPRI` if a high-priority message was encountered, otherwise it is set to 0.

The options for `getpmsg()` are different from those for `getmsg()`. *flagsp* references a bit mask with the following options, which are mutually exclusive: `MSG_HIPRI`, `MSG_BAND` and `MSG_ANY`. Like `getmsg()`, `getpmsg()` processes the next message to become available in the read queue of the stream head. The user in turn can choose to receive only high-priority messages by setting the integer referenced by *flagsp* to `MSG_HIPRI` and the integer referenced by *bandp* to 0. In this case, `getpmsg()` only processes the next message if it is high-priority. Similarly, the user can call up a message from a special priority range by setting the integer referenced by *flagsp* to `MSG_BAND`, and the integer referenced by *bandp* to the desired priority range. In this case, `getpmsg()` only processes the next message if it is in a priority range greater than or equal to the integer referenced by *bandp*, or if it is a high-priority message. If a user only wants to call the first message in the queue, the integer referenced by *flagsp* should be set to `MSG_ANY`, and the integer referenced by *bandp* should be set to 0. If the message received was a high-priority one, on return the integer referenced by *flagsp* is set to `MSG_HIPRI`, and the integer referenced by *bandp* is set to 0. With all other messages the integer referenced by *flagsp* is set to `MSG_BAND`, and the integer referenced by *bandp* is set to the priority range of the message.

If `O_NDELAY` and `O_NONBLOCK` were not set, `getmsg()` and `getpmsg()` block until there is a message of the type specified by *flagsp* in the read queue of the stream head. If `O_NDELAY` or `O_NONBLOCK` was set and there is no message of the specified type in the read queue, `getmsg()` and `getpmsg()` are unsuccessful and `errno` is set to `EAGAIN`.

If a stream from which the messages are to be fetched experiences a loss of connection, `getmsg()` and `getpmsg()` continue to work normally, as described above, until the read queue is empty. Afterwards, 0 is returned in the `len` fields of *ctlptr* and *dataptr*.

If a message is not fully read with a `getmsg()` or `getpmsg()` call, the rest of the message can be fetched with subsequent `getmsg()` or `getpmsg()` calls. If, however, a high-priority message arrives in the stream head of the read queue, the next `getmsg()` or `getpmsg()` call gives priority to this message before processing the rest of the partial message received previously.

## Return value

Non-negative value

if successful.

0 if a complete message was read successfully.

MORECTL indicates that there is more control information waiting to be retrieved.

MOREDATA indicates that there is more data waiting to be retrieved.

bit-wise OR of MORECTL and MOREDATA indicates that both types still remain.

## Errors

`getmsg()` or `getpmsg()` will fail if:

EAGAIN `O_NDELAY` or `O_NONBLOCK` is set and there are no messages available.

EBADF *fdes* is not a valid file descriptor open for reading.

EBADMSG The message in the queue that is to be read is not valid for `getmsg()` or `getpmsg()`.

EINTR A signal was caught during the `getmsg()` or `getpmsg()` system call.

EINVAL An invalid value was specified in *flagsp*, or the stream or multiplexer specified by *fdes* is directly or indirectly linked downstream with a multiplexer.

ENOSTR No stream is assigned to the *fdes* file descriptor.

`getmsg()` and `getpmsg()` can also fail if a STREAMS error message was received at the stream head before the `getmsg()` call. In this case, `errno` displays the STREAMS error which occurred before.

See also `poll()`, `putmsg()`, `read()`, `write()`, `stropts.h`.

## getopt, optarg, optind, opterr, optopt - command option parsing

Syntax `#include <unistd.h>`

```
int getopt(int argc, char * const argv[ ], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

Description `getopt()` is a command-line parser that can be used by applications that follow the specific conventions for entering commands defined in the XPG4 specification (see the manual "POSIX Commands (BS2000/OSD)"). The remaining guidelines are the responsibility of the application.

`getopt()` returns the next option character from `argv` that matches a character in `optstring`. `argc` is the argument count, as passed to `main()` (see `exec`).

`argv` points to an array of `argc+1` elements containing `argc` pointers to character strings, followed by a null pointer. It contains the option names, as passed to `main()` (see `exec`).

`optstring` is a string of recognized option characters (see the manual "POSIX Commands (BS2000/OSD)"). If a character in this string is followed by a colon (:), the option is expected to take one or more arguments.

`optind` is an external variable that represents the index of the next element of the `argv[]` vector to be evaluated. It is initialized to 1 by the system, and `getopt()` updates it when it finishes evaluating each element of `argv[]`. If an element of `argv[]` contains multiple option characters, it is unspecified how `getopt()` determines which options have already been processed.

`optarg` is an external variable that is set by `getopt()` when an option takes an argument. This is done as follows.

1. 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`, this indicates a missing option-argument, and `getopt()` reports an error.
2. Otherwise, `optarg` is set to point to the string following the option character, and `optind` is incremented by 1.

`opterr` is an external variable that controls the output of error messages in the event of an error. If it is set to 0, the output of an error message is suppressed.

`optopt` is an external variable containing the option character that caused `getopt()` to fail.

## Return value

- Next option character from the command line upon successful completion.
- :
- if an option-argument is missing and the first character in *optstring* was a colon; `getopt()` sets the variable `optopt` to the option character that caused the error.
- ?
- if an option character that is not contained in *optstring* is found or if an option-argument is missing and the first character in *optstring* was not a colon. In these 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, `getopt()` prints a diagnostic message to `stderr` in the format specified for the `getopt()`s command (see also the manual "POSIX Commands (BS2000/OSD)").
- 1
- if `argv[optind]` is a null pointer, or if `*argv[optind]` is not the character "-", or if `argv[optind]` points to the string "-": `optind` is not changed. In these cases.
- 1
- if `argv[optind]` points to the string "--": `optind` is incremented.

## Notes

`getopt()` does not fully check for mandatory arguments. That is, given an option string `a:b` and the input `-a -b`, `getopt()` will assume that `-b` is the mandatory argument for option `-a` and not that a mandatory argument is missing for `-a`.

Multiple options cannot be combined if the last option requires an argument. For example, if `a` and `b` are normal options and option `o` requires the argument `xxx`, then `cmd -ab -o xxx` should be specified, not `cmd -abo xxx`. Although the latter grouped syntax is still supported by the current implementation, it may not be supported in future releases.

*BS2000*

When a program is started in the BS2000 environment, the program parameters are supplied as is usual for C programs (see the C and C User Guides). □

If the integer value returned by `getchar()` is stored into a variable of type `char` and then compared against the integer constant `E0F`, the comparison may never succeed, since no sign-extension of a variable of type `char` on widening to integer occurs.

## See also

`exec`, `unistd.h`, `getopts` command (see also the manual "POSIX Commands (BS2000/OSD)").

## getpagesize - get current page size

**Syntax**      `#include <unistd.h>`  
`int getpagesize(void);`

**Description** `getpagesize()` returns the number of bytes of a memory page.

A `getpagesize()` call is equivalent to calling `sysconf(_SC_PAGE_SIZE)` or `sysconf(_SC_PAGESIZE)`.

`getpagesize()` is not thread-safe.

**Return value**

Current page size

The function is always successful.

**Notes**      The page size returned by `getpagesize()` does not have to match the size of the memory pages as divided up for the hardware.

Under POSIX, however, this size is the same as that set for the hardware.

This page size need not match the minimum size that can be requested with `malloc()`, nor may an application rely on the fact that an object of this size can be allocated with `malloc()`.

**See also**    `brk()`, `getrlimit()`, `mmap()`, `mprotect()`, `munmap()`, `msync()`, `sysconf()`, `unistd.h`.

## getpass - read string of characters without echo

**Syntax**      `#include <unistd.h>`

```
char *getpass(const char *prompt);
```

**Description** `getpass()` performs the following actions. It

- opens the process' controlling terminal,
- writes the null-terminated string *prompt* to that device,
- disables echoing,
- reads a string of characters up to the next newline character or EOF,
- restores the terminal state, and
- closes the special file for the terminal.

**Return value**

Pointer to a null-terminated string

upon successful completion. The return value consists of at most `{PASS_MAX}` bytes that were read from the terminal device.

Null pointer      if an error occurs. The original state of the terminal is restored, and `errno` is set to indicate the error.

**Errors**      `getpass()` will fail if:

`EINTR`          `getpass()` was interrupted by a signal.

`EIO`             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.

`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.

`ENXIO`          The process does not have a controlling terminal.

**Notes**          The return value points to static data whose content may be overwritten by each call.

`pclose()` is executed only for POSIX files.

`getpass()` is not thread-safe. Will no longer be supported by the X/Open-Standard in future.

**See also**      `limits.h`, `unistd.h`.

## getpgid - get process group ID

**Syntax**      `#include <unistd.h>`

```
pid_t getpgid(pid_t pid);
```

**Description** `getpgid()` returns the process group ID of the process whose process ID is *pid*. If *pid* is 0, the process group ID of the calling process is returned.

**Return value**

Process group ID

if successful.

(pid\_t)-1 if an error occurs. `errno` is set to indicate the error.

**Errors**      `getpgid()` will fail if:

**EPERM**      The process whose process ID is *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of this process from within the calling process.

**ESRCH**      There is no process with a process ID *pid*.

**EINVAL**      The value of *pid* is invalid.

**See also**      `exec`, `fork()`, `getpgrp()`, `getpid()`, `getsid()`, `setpgid()`, `setsid()`, `unistd.h`.

## getpgmname - get program name *(BS2000)*

**Syntax**      `#include <stdlib.h>`

```
char *getpgmname(void);
```

**Description** `getpgmname()` returns the name of the calling program. The result corresponds to `argv[0]` of the `main` function.

**Return value**

Pointer to the program name.

The function is always successful.



## getpgrp - get process group ID

**Syntax**      `#include <unistd.h>`

*Optional*

`#include <sys/types.h> □`

`pid_t getpgrp(void);`

**Description** `getpgrp()` returns the process group ID of the calling process.

**Return value**

Process group ID

The function is always successful.

**See also** `exec`, `fork()`, `getpid()`, `getppid()`, `kill()`, `setpgid()`, `setsid()`, `sys/types.h`, `unistd.h`, and the manual "POSIX Basics (BS2000/OSD)".

## getpid - get process ID

**Syntax**      `#include <unistd.h>`

*Optional*

`#include <sys/types.h> □`

`pid_t getpid(void);`

**Description** `getpid()` returns the process ID of the calling process.

**Return value**

Process ID of the calling process.

The function is always successful.

**See also** `exec`, `fork()`, `getpgrp()`, `getppid()`, `kill()`, `setpgid()`, `setsid()`, `sys/types.h`, `unistd.h`.

## getpmsg - get message from STREAMS file

```
#include <pwd.h>
```

```
int getpmsg(int fildev, struct strbuf *ctlptr, struct strbuf *dataptr, int *bandp, int *flagsp);
```

Description See `getmsg()`.

## getppid - get parent process ID

Syntax `#include <unistd.h>`

*Optional*

```
#include <sys/types.h>  $\square$ 
```

```
pid_t getppid(void);
```

Description `getppid()` returns the parent process ID of the calling process.

Return value

Parent process ID of the calling process.

The function is always successful.

See also `exec`, `fork()`, `getpgrp()`, `getpid()`, `kill()`, `setpgid()`, `setsid()`, `sys/types.h`, `unistd.h`.

## getpriority, setpriority - get or set process priority

**Syntax**      `#include <sys/resource.h>`  
`int getpriority(int which, id_t who);`  
`int setpriority(int which, id_t who, int priority);`

**Description** `getpriority()` retrieves the current scheduling priority of the process, the process group or the user.

`setpriority()` sets the scheduling priority of the process, the process group or the user.

The arguments *which* and *who* define which process is addressed. *which* can take the following values: `PRIO_PROCESS`, `PRIO_PGRP` or `PRIO_USER`. Depending on this, the contents of *who* are interpreted as process ID, process group ID or user ID respectively. A null value for *who* denotes the current process, the current process group or the current user.

`getpriority()` returns the highest priority (the lowest numerical value) that is claimed by one of the specified processes. `setpriority()` sets the priorities of all specified processes to the value specified via *priority*.

The default priority is 0; lower priorities mean improved scheduling. If the priority is below -20, the value -20 is used; if it is over 20, the value 20 is used.

Only users with the appropriate authorization can reduce priorities.

When threads are used, the `getpriority()` and `setpriority()` functions affect the process or a thread in the following manner:

- Query or set the scheduling priority of the process.
- If the process is multithreaded, the scheduling priority affects all threads of the process.

### Return value

`getpriority()`:

$-20 \leq \text{return value} \leq 20$

if successful.

-1 if an error occurs. `errno` is set to indicate the error.

`setpriority()`:

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

`getpriority()` and `setpriority()` will fail if:

ESRCH        No process was found to which the specified values *which* and *who* apply.  
EINVAL      *which* was neither `PRIO_PROCESS`, `PRIO_PGRP` nor `PRIO_USER`, or *who* did not contain a valid process ID, process group ID or user ID.

`setpriority()` can also fail if:

EPERM        A process was found but neither the effective user ID nor the real one matches the effective user ID of the process whose priority is to be changed.  
EACCES      An attempt was made to set the priority to a lower value, which means a higher priority, but the current process does not have the appropriate authorization.

Notes        What effect the changing of the scheduling priority has depends on the algorithm of the process scheduling.

As `getpriority()` can legitimately also return the value -1, the external variable `errno` must be deleted before the call and then checked to establish whether the value -1 indicates an error or a permissible value.

See also     `nice()`, `sys/resource.h`.

## getpwent - read user data from user catalog

Syntax       `#include <pwd.h>`  
              `struct passwd *getpwent(void);`

Description See `endpwent()`.

## getpwnam - get user name

**Syntax**      `#include <pwd.h>`

*Optional*

`#include <sys/types.h> □`

`struct passwd *getpwnam(const char *name);`

**Description** `getpwnam()` searches the user catalog for an entry in which the *pw\_name* component matches *name* (see also `pwd.h` and the manual "POSIX Basics (BS2000/OSD)").

`getpwnam()` is not thread-safe. Use the reentrant function `getpwnam_r()` when needed.

### Return value

Pointer to a structure of type `passwd` (see `pwd.h`) if successful.

Null pointer if an error occurs when reading or no matching entry was found. `errno` is set to indicate the error.

**Errors** `getpwnam()` fails if:

`EINVAL`      *name* is too long.

`EFAULT`      An error occurs when creating the `passwd` structure, or an invalid *name* string is specified.

`ENOENT`      The user is not recognized.

**Notes** The return value may point to a static area which may be overwritten by a subsequent call to `cuserid`, `getpwnam` or `getpwuid`.

Applications wishing to check for error situations should set `errno` to 0 before calling `getpwnam()`. If `errno` is set to non-zero on return, an error occurred.

The three names associated with the current process can be determined as follows: `getpwuid(geteuid())` returns the name associated with the effective user ID of the process; `getlogin()` returns the name associated with the current login activity; and `getpwuid(getuid())` returns the name associated with the real user ID of the process.

**See also** `geteuid()`, `getlogin()`, `getpwnam_r()`, `getpwuid()`, `getuid()`, `limits.h`, `pwd.h`, `sys/types.h`, and the manual "POSIX Basics (BS2000/OSD)".

## getpwnam\_r - get user name (thread-safe)

**Syntax**      `#include <sys/types.h>`  
                 `#include <pwd.h>`

```
int getpwnam_r(const char *nam, struct passwd *pwd, char *buffer,
               size_t bufsize, struct passwd **result);
```

**Description** The `getpwnam_r()` and `getpwuid_r()` functions update the `passwd` structure pointed to by `pwd` and store a pointer to this structure at the address pointed to by `result`. The structure contains the entry from the user catalog whose `pw_name` or `pw_uid` component matches `nam` or `uid`, respectively.

The `passwd` structure found in the user catalog is copied to the memory area of length `bufsize` passed in the parameter `buffer`. The maximum size required for this buffer can be determined via the `sysconf()` parameter `{_SC_GETPW_R_SIZE_MAX}`.

**Return value**

0                    if successful.

Otherwise `errno` is set to indicate the error.

**Errors**        The `getpwnam_r()` and `getpwuid_r()` fail if:

ERANGE            The memory area pointed to by `buffer` of length `bufsize` is not large enough to hold the data pointed to by the resulting group structure.

**Notes**        If an error occurs or if the requested entry is not found, a null pointer is returned in the address pointed to by `result`.

**See also**     `getpwnam()`, `getpwuid()`, `pwd()`, `types()`.

## getpwuid - get user ID

Syntax `#include <pwd.h>`

*Optional*

`#include <sys/types.h> □`

`struct passwd *getpwuid(uid_t uid);`

Description `getpwuid()` searches the user catalog for an entry in which the *pw\_uid* component (see the `passwd` structure in `pwd.h`) matches *uid*. Subsequent structures with the same user ID are not found.

Return value

Pointer to a structure of type `passwd` (see `pwd.h`) if successful.

Null pointer if an error occurs when reading or no matching entry with a `pw_uid` component matching *uid* was found in the user catalog.

Errors `getpwuid()` fails if: :

- EFAULT An error occurs when creating the `passwd` structure.
- ENOENT The user is not recognized.

Notes The return value may point to a static area which may be overwritten by a subsequent call to `cuserid`, `getpwnam` or `getpwuid`.

Applications wishing to check for error situations should set `errno` to 0 before calling `getpwuid()`.

The three names associated with the current process can be determined as follows: `getpwuid(geteuid())` returns the name associated with the effective user ID of the process; `getlogin()` returns the name associated with the current login activity; and `getpwuid(getuid())` returns the name associated with the real user ID of the process.

See also `cuserid()`, `getpwuid_r()`, `getpwnam()`, `geteuid()`, `getuid()`, `getlogin()`, `limits.h`, `pwd.h`, `sys/types.h`, and the manual "POSIX Basics (BS2000/OSD)".

## getpwuid\_r - get user ID (thread-safe)

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
               size_t bufsize, struct passwd **result);
```

Description See `getpwuid()`.



## getrlimit, setrlimit - get or set limit for resource

**Name**        **getrlimit, getrlimit64, setrlimit, setrlimit64**

**Syntax**      `#include <sys/resource.h>`

```
int getrlimit (int resource, struct rlimit *rlp);
int getrlimit64 (int resource, struct rlimit64 *rlp);
int setrlimit (int resource, const struct rlimit *rlp);
int setrlimit64 (int resource, const struct rlimit64 *rlp);
```

**Description** This call limits the use of a variety of resources via a process and all its child processes; `getrlimit()` reads the limits and `setrlimit()` sets them.

Each `getrlimit()` or `setrlimit()` call specifies a particular resource *resource* and a particular limit for it which is referenced by *rlp*. The limit comprises a pair of values located in the `rlimit` structure. *rlp* must be a pointer to such a structure.

`rlimit` contains the following components:

```
rlim_t  rlim_cur;          /* Current limit */
rlim_t  rlim_max;         /* Maximum limit */
```

`rlim_t` is an arithmetical data type to which objects of types `int`, `size_t` and `off_t` can be converted without information getting lost.

`rlim_cur` specifies the current or soft limit, `rlim_max` the maximum or hard limit. Soft limits can be set by a process to a value that is less than or equal to the hard limit. A process can reduce its hard limit (this is not reversible), so that it becomes greater than or equal to the soft limit. Only a process with the appropriate privileges can increase a hard limit. Both hard and soft limits can be changed by a single `setrlimit()` call, taking the above restrictions into account.

The `RLIM_INFINITY` value, which is defined in `sys/resource.h`, is equivalent to an infinitely large limit, i.e. if `getrlimit()` returns `RLIM_INFINITY` for a resource, the implementation does not allow for a limit for this resource. If `setrlimit()` with `RLIM_INFINITY` is successfully executed for a resource, it is no longer checked whether this resource complies with this value.

If the limit for a resource is correctly represented in an object of type `rlimit_t` when the `getrlimit()` function is used, then this representation is returned. However, if the limit for the parameter is equal to the saved limit, then the value `RLIM_SAVED_MAX` is returned. Otherwise the value `RLIM_SAVED_CUR` is returned.

If the requested limit is `RLIM_INFINITY` for the `setrlimit()` function, then no value is intended for the new limit. If the requested limit is `RLIM_SAVED_MAX`, the new limit is the saved hard limit. If `RLIM_SAVED_CUR` is requested as the limit, the new limit is the saved

soft limit. Otherwise the new value is the requested value. Furthermore, the corresponding saved limit is overwritten by the new limit if it can be correctly represented in an object of type `rlim_t`.

If a limit is set to `RLIM_SAVED_MAX` or `RLIM_SAVED_CUR`, the result is undefined unless an earlier `getrlimit()` call returned this value as the hard or soft limit for the corresponding resource limit.

**The same also applies to the `getrlimit64()`, `setrlimit64()` functions and to the `RLIM64_INFINITY`, `RLIM64_SAVED_MAX` and `RLIM64_SAVED_CUR` values.**

The following table lists the possible resources with their descriptions and the resulting measures when a limit is exceeded:

| Resource                   | Description                                                                                                                                                                                   | Measure                                                                                                                                                                                                                                                         |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>RLIMIT_CORE</code>   | Maximum size of a core dump file in bytes that can be generated by a process. A size of 0 prevents the generation of memory dump files.                                                       | Writing of a core dump file ceases when this size is reached.                                                                                                                                                                                                   |
| <code>RLIMIT_CPU</code>    | Maximum CPU time used by a process.                                                                                                                                                           | <code>SIGXCPU</code> is sent to the process. If the process blocks, catches or ignores <code>SIGXCPU</code> , the behavior is undefined.                                                                                                                        |
| <code>RLIMIT_DATA</code>   | Maximum size of the data segment of a process in bytes. Under POSIX the size is unlimited, because <code>sbrk()</code> , <code>brk()</code> and <code>malloc()</code> use independent memory. | <code>brk()</code> , <code>malloc()</code> and <code>sbrk()</code> will fail and <code>errno</code> will contain <code>ENOMEM</code> .                                                                                                                          |
| <code>RLIMIT_FSIZE</code>  | Maximum length of a file in bytes that can be generated by a process. A length of 0 prevents files from being generated.                                                                      | <code>SIGXFSZ</code> is sent to the process. If the process blocks, catches or ignores <code>SIGXFSZ</code> , further attempts to enlarge the file will fail and <code>errno</code> will contain <code>EFBIG</code> .                                           |
| <code>RLIMIT_NOFILE</code> | Maximum number of open file descriptors that a process can have.                                                                                                                              | Functions which create new file descriptors will fail and <code>errno</code> will contain <code>EMFILE</code> .                                                                                                                                                 |
| <code>RLIMIT_STACK</code>  | Maximum size of the process stack in bytes. The system does not let the stack grow automatically beyond this limit.                                                                           | <code>SIGSEGV</code> is sent to the process. If the process blocks, ignores or catches <code>SIGSEGV</code> and does not use an alternative stack (see <code>sigaltstack()</code> ), <code>SIG_DFL</code> is set as the handling mode of <code>SIGSEGV</code> . |
| <code>RLIMIT_AS</code>     | Maximum length of the address area of a process in bytes.                                                                                                                                     | The functions <code>brk()</code> , <code>malloc()</code> , <code>mmap()</code> and <code>sbrk()</code> will fail and <code>errno</code> will contain <code>ENOMEM</code> . Also, the stack can no longer increase and the above-mentioned effects occur.        |

As the limit information is managed for each process, the shell statement `ulimit` must execute this system call directly in order to influence all future processes that are generated by the shell.

The value of the current limit of the following resources influences these implementation-dependent constants:

| Limit         | Implementation-dependent constant |
|---------------|-----------------------------------|
| RLIMIT_FSIZE  | FCHR_MAX                          |
| RLIMIT_NOFILE | OPEN_MAX                          |

There is no difference in functionality between `getrlimit()` / `setrlimit()` and `getrlimit64()`/`setrlimit64()` except that `getrlimit64()` and `setrlimit64()` use a `rlimit64` structure.

The `rlimit64` structure is defined in the same manner as `rlimit`:

```
rlim64_t rlim_cur
rlim64_t rlim_max
```

If threads are used, then the function affects the process or a thread in the following manner:

- RLIMIT\_CPU: ... if the process traps or ignores the SIGXCPU signal or all threads belonging to this process block this signal, then the behavior is undefined.
- RLIMIT\_FSIZE: ... the SIGXFSZ signal is generated for the thread. If the thread blocks the SIGXFSZ signal or the process traps or ignores this signal, further attempts to increase the size of the file will fail and `errno` is set to EFBIG.
- RLIMIT\_STACK: ... the SIGSEGV signal is generated for the thread. If the thread blocks the SIGSEGV signal or the process traps or ignores this signal and does not use an alternative stack, the SIG\_DFL handling mode of SIGSEGV SIG\_DFL is set.

Return value

- 0                   if successful.
- 1                  if an error occurs. `errno` is set to indicate the error.

**Errors**      `getrlimit()` and `setrlimit()` will fail if:

**EINVAL**      An invalid resource was specified, or in a `setrlimit()` call the new value in `rlim_cur` is greater than the value in `rlim_max`.

**EPERM**      The limit specified in `setrlimit()` would increase the maximum limit, but the calling process does not have the appropriate privileges.

In addition, `setrlimit()` will fail if:

**EINVAL**      The specified limit cannot be reduced because a higher value is currently in use.

**See also**    `brk()`, `exec`, `fork()`, `getdtablesize()`, `malloc()`, `open()`, `sigaltstack()`, `sysconf()`, `ulimit()`, `stropts.h`, `sys/resource.h`.

## getrusage - get information on usage of resources

**Syntax**      `#include <sys/resource.h>`

```
int getrusage(int who, struct rusage *r_usage);
```

**Description** `getrusage()` returns information on the resources used by the current process or its terminated child processes and the child processes whose termination the process is waiting for.

The *who* argument can contain the value `RUSAGE_SELF` or `RUSAGE_CHILDREN`. In the first case, information on the resources of the current process is returned. In the second case, `getrusage()` outputs information on the resources of the terminated child processes of the current process and the resources of the child process which the current process is waiting for. If the process never waits for a child process, e.g. because `SA_NOCLDWAIT` is set in the parent process or `SIGCHLD` is set to `SIG_IGN`, no information on the resource usage of the child process will be returned.

The *r\_usage* argument points to an `rusage` structure which contains the following components:

|                                      |                                                                                                           |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>struct timeval ru_utime</code> | The total time the execution takes in user mode. The interval is specified in seconds and microseconds.   |
| <code>struct timeval ru_stime</code> | The total time the execution takes in system mode. The interval is specified in seconds and microseconds. |

**Return value**

|    |                                                                                              |
|----|----------------------------------------------------------------------------------------------|
| 0  | if successful. The <code>rusage</code> structure is filled up with the corresponding values. |
| -1 | if an error occurs. <code>errno</code> is set to indicate the error.                         |

**Errors** `getrusage()` will fail if:

|                     |                                                         |
|---------------------|---------------------------------------------------------|
| <code>EINVAL</code> | The <i>who</i> argument does not contain a valid value. |
|---------------------|---------------------------------------------------------|

*Extension*

|                     |                                                                                                                |
|---------------------|----------------------------------------------------------------------------------------------------------------|
| <code>EFAULT</code> | The address specified by the <i>r_usage</i> argument is not a valid area of the address area of the process. □ |
|---------------------|----------------------------------------------------------------------------------------------------------------|

**See also** `exit()`, `gettimeofday()`, `read()`, `sigaction()`, `time()`, `times()`, `wait()`, `write()`, `sys/resource.h`.

## gets - get string from standard input stream

Syntax `#include <stdio.h>`  
`char *gets(char *s);`

Description `gets()` reads bytes from the standard input stream into the array pointed to by *s* until a newline is read or an end-of-file condition is encountered. A newline character, if any, is overwritten by a null byte.

`gets()` can mark the structure component `st_atime` for the file to which *stream* is assigned for changing (see `sys/stat.h`). The structure component `st_atime` is updated as soon as `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` are called successfully for *stream* and return data which is not was not provided by a preceding call to `ungetc()` or `ungetwc()`.

### Return value

Pointer to the result string  
upon successful completion. `gets()` terminates the string with a null byte.

Null pointer  
if the stream is at end-of-file. The end-of-file indicator for the stream is set; `errno` is not set.

If a read error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

Errors See `fgetc()`.

Notes Reading a line that overflows the array pointed to by *s* causes undefined results. The use of `fgets()` is recommended.

If `gets()` is reading from the standard input `stdin` in the POSIX environment, and EOF is the end criterion for reading, the EOF condition can be achieved by the following actions:

- ▶ on a block-special terminal: by entering the key sequence `@ @ d`
- ▶ on a character-special terminal: by entering `CTRL+D`

### BS2000

If `fgetc()` is reading from the standard input `stdin` in the BS2000 environment, and EOF is the end criterion for reading, the EOF condition can be achieved by means of the following actions at the terminal:

1. by pressing the `K2` key.
2. by entering the system commands EOF and RESUME-PROGRAM. □

The program environment determines whether `getsid()` is executed for a BS2000 or POSIX file.

See also `feof()`, `ferror()`, `fgets()`, `stdio.h`.

## getsid - get process group ID

Syntax `#include <unistd.h>`  
`pid_t getsid(pid_t pid);`

Description The `getsid()` function returns the process group ID of the process that is session leader of the process with the ID *pid*. If *pid* is `(pid_t)0`, then `getsid()` returns the session number of the calling process.

### Return value

Process group ID  
if successful.  
`(pid_t)-1` if an error occurs. `errno` is set to indicate the error.

Errors `getsid()` will fail if:

`EPERM` The process with the process ID *pid* is not in the same session as the calling process, and the implementation does not support access by the calling process to the session number of the specified process.  
`ESRCH` There is no process with the process ID *pid*.

See also `exec`, `fork()`, `getpid()`, `getpgrp()`, `setpgrp()`, `setsid()`, `unistd.h`.

## getsubopt - get suboptions from string

Syntax `#include <stdlib.h>`

```
int getsubopt (char **optionp, char * const *tokens, char **valuep);
```

Description `getsubopt()` extracts suboptions from an option argument which was first processed by `getopt()`. These suboptions must be separated by commas and can consist of either a single token or a pair of token values separated by an equals sign. Because commas are used to delimit suboptions in the option string, they must not be part of the suboption or the value of a suboption. Similarly, a token must not contain an equals sign, because tokens and associated values are separated by equals signs.

`getsubopt()` receives the address of a pointer to the option string which represents an array of possible tokens and the address of a pointer to a value string. The index of the token that corresponds to the suboption from the transferred string is returned; if no corresponding suboption is found, -1 is returned. If the option string under *optionp* only contains one suboption, `getsubopt()` updates *optionp* such that the null byte at the end of the string is pointed to; otherwise, the suboption is isolated through replacement of the separating comma being with a null byte, and *optionp* points to the beginning of the next suboption. If the suboption is assigned a value, `getsubopt()` updates *valuep* such that the first character of the value is pointed to. Otherwise, *valuep* is set to zero.

The token array is organized as a sequence of pointers to null-terminated strings. The end of the token array is identified by a null pointer.

If *valuep* is not zero, `getsubopt()` returns the suboption to which a value was assigned. The calling program can use this information to determine whether the presence or the omission of a value for this suboption represents an error.

If `getsubopt()` does not find a suboption in the *tokens* array, the calling program should decide whether this means an error or whether the non-recognized option should be passed to another program.

Return value

Index of the matching token if successful.

-1 if no matching token was found.

Notes

During processing of the token, commas in the option string are changed into null bytes. Blanks in tokens or pairs of token values must be protected from shell by quotation marks.

See also

`getopt()`, `stdlib.h`



## gettimeofday - read current time of day

**Syntax**      `#include <sys/time.h>`

```
int gettimeofday(struct timeval *tp, void *tzp);
```

**Description** `gettimeofday()` reads the current system time, expressed as seconds and microseconds since 00:00 Coordinated Universal Time (UTC), January 1, 1970. The resolution of the system clock is hardware-dependent; the time may be updated continuously or at specific time intervals.

*tp* points to a structure of type `timeval` containing the following members:

```
long   tv_sec;    /* seconds since January 1, 1970 */
long   tv_usec;  /* and microseconds */
```

If *tp* is a null pointer, the current time is not read.

*tzp* must be a null pointer, otherwise the behavior is undefined.

Information on time zones is contained in the environment variable TZ. See `timezone`.

**Return value**

0                if successful.

-1               if an error occurs.

**Notes**        Programs which want to be portable must not rely on the return value -1 in the event of an error.

**See also**     `ctime()`, `ftime()`, `timezone`, `sys/time.h`.

## gettsn - get TSN (task sequence number) *(BS2000)*

**Syntax**      `#include <stdlib.h>`  
              `char *gettsn(void);`

**Description** `gettsn()` returns the task sequence number (TSN) of the calling program.

**Return value**  
Task sequence number (TSN) of the calling program.

**Notes**        `gettsn()` writes its result into an internal C data area that is overwritten with each call.

## getuid - get real user ID

**Syntax**      `#include <unistd.h>`  
              *Optional*  
              `#include <sys/types.h> □`  
              `uid_t getuid(void);`

**Description** `getuid()` returns the real user ID of the calling process.

**Return value**  
Real user ID of the calling process.  
The function is always successful.

**See also**     `getegid()`, `geteuid()`, `getgid()`, `setuid()`, `sys/types.h`, `unistd.h`.

## getutxent, getutxid, getutxline - get utmpx entry

Syntax      `#include <utmpx.h>`  
             `struct utmpx *getutxent (void);`  
             `struct utmpx *getutxid (const struct utmpx *id);`  
             `struct utmpx *getutxline (const struct utmpx *line);`

Description See `endutxent()`.

## getw - read word from stream

Syntax      `#include <stdio.h>`  
`int getw(FILE *stream);`

Description `getw()` reads the next word from the input stream pointed to by *stream*. The size of a word is the size of an `int` and may vary from machine to machine. The `getw()` function presumes no special alignment in the file.

`getw()` can mark the structure component `st_atime` for the file to which *stream* is assigned for changing (see `sys/stat.h`). The structure component `st_atime` is updated as soon as `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` or `scanf()` are called successfully for *stream* and return data which is not was not provided by a preceding call to `ungetc()` or `ungetwc()`.

`getw()` is not thread-safe.

### Return value

Next word from the input stream pointed to by *stream* (as an `int`) upon successful completion.

EOF            if the stream is at end-of-file. The end-of-file indicator for the stream is set; `errno` is not set.

EOF            if a read error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

Errors        See `fgetc()`.

Notes        Due to possible differences in word length and byte ordering, files written using `putw()` are machine-dependent, and may not be correctly read when `getw()` is used on a different processor.

Since the representation of EOF is a valid integer, applications wishing to check for errors should use `ferror()` and `feof()`.

The program environment determines whether `getw()` is executed for a BS2000 or POSIX file.

See also     `ferror()`, `getc()`, `putw()`, `stdio.h`.

## getwc - get wide character from stream

Syntax `#include <wchar.h>`

*Optional*

`#include <stdio.h>` □

`wint_t getwc(FILE *stream);`

Description `getwc()` is implemented both as a function and as a macro. It is equivalent to `fgetwc()`, except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

Return value

See `fgetwc()`.

Errors

See `fgetwc()`.

Notes

This interface is provided to support some current implementations and possible future ISO standards.

When `getwc()` is implemented as a macro, it may handle a *stream* argument with side effects incorrectly. In particular, `getwc(*f++)` may not work as expected. The use of `fgetwc()` is therefore recommended in such situations.

See also

`fgetwc()`, `stdio.h`, `wchar.h`.

## getwchar - get wide character from standard input stream

**Syntax**      `#include <wchar.h>`  
                `wint_t getwchar(void);`

**Description** The function call `getwchar(void)` is equivalent to `getwc(stdin)`, i.e. it reads a wide character from the standard input stream.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**Return value**  
See `fgetwc()`.

**Errors**      See `fgetwc()`.

**Notes**        If the value returned by `getwchar()` is stored into a variable of type `wchar_t` and then compared against the `wint_t` macro `WEOF`, the comparison may never succeed.

**See also**     `fgetwc()`, `getwc()`, `wchar.h`.

## getwd - get pathname of current working directory

**Syntax**      `#include <unistd.h>`  
`char *getwd(char *path_name);`

**Description** `getwd()` determines the absolute pathname of the current working directory of the calling process and copies it into a string pointed to by the *path\_name* argument.

If the length of the pathname of the current working directory including the null byte is greater than `{PATH_MAX}+1`, `getwd()` will fail and return a null pointer.

**Return value**

Pointer to a string      if successful. The string contains the absolute pathname of the current working directory.

Null pointer      if an error occurs. The string pointed to by *path\_name* contains an error text.

**Notes**      Portable applications should use the `getcwd()` function instead of `getwd()`.

**See also**      `getcwd()`, `unistd.h`.

## gmatch - global pattern matching *(extension)*

**Syntax**      `#include <libgen.h>`  
`int gmatch(const char *str, const char *pattern);`

**Description** `gmatch()` checks whether the null-terminated string *str* matches the null-terminated pattern string *pattern*. A backslash `\` is used as an escape character in pattern strings.

**Return value**

`≠ 0`      if the string matches the pattern.

`0`      if no match was found.

## gmtime - convert date and time to UTC

Syntax `#include <time.h>`

```
struct tm *gmtime(const time_t *clock);
```

Description The `gmtime()` function accepts arguments of type `time_t`, which are pointed to by `clock`, and which supply the time in seconds since 00:00:00 UTC (Universal Time Coordinated, January 1, 1970).

`gmtime()` returns pointers to the `tm` structures explained below.

`gmtime()` converts directly into UTC, i.e. the time used by the SINIX system.

The declarations of all functions, external values, and of the `tm` structure are contained in the header `time.h`. The `tm` structure is defined as follows:

```
struct    tm {
    int    tm_sec;        /* Seconds - [0, 61] for skipped seconds */
    int    tm_min;        /* Minutes - [0, 59] */
    int    tm_hour;       /* Hours - [0, 23] */
    int    tm_mday;       /* Day of month - [1, 31] */
    int    tm_mon;        /* Months - [0, 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;     /* Option for daylight saving time */
};
```

`tm_isdst` is positive if daylight saving time is set,  
null if daylight saving time is not set,  
and negative if the information is not available.

### BS2000

`gmtime()` interprets the time specification of type `time_t` as the number of seconds that have elapsed since January 1, 1950, 00:00:00. From this number, `gmtime()` calculates the date and time and stores the result in a structure of type `tm`. In this implementation, `gmtime()` is equivalent to `localtime()`; both functions return the local time. □

`gmtime()` is not thread-safe. Use the reentrant function `gmtime_r()` when needed.

Return value

Pointer to a structure of type `struct tm`.

Notes

The return values of `gmtime()` point to static data that is overwritten with each call.

`gmtime()` does not support local date and time formats; to ensure maximum portability, `strftime()` should be used instead.



`gmtime()` writes its result to an internal C data area that is overwritten with each call. Furthermore, `gmtime()` and `localtime()` use the same data area, which means that if they are called in succession, the result of the first call will be overwritten.

See also `altzone()`, `ctime()`, `daylight`, `localtime()`, `strftime()`, `tzname`, `tzset()`.

## gmtime\_r - convert date and time to UTC (thread-safe)

Syntax `#include <time.h>`  
`struct tm *gmtime_r(const time_t * clock, struct tm * result);`

Description The `gmtime_r()` function converts the time (number of seconds since the beginning of the epoch) pointed to by *clock* to the UTC time (Coordinated Universal Time) in the format described in the `struct tm` structure. The result is stored in the data area pointed to by *result*.

Return value  
Address of the structure pointed to by *result*,  
if successful.  
Null pointer if an error occurs or if UTC is not available.

See also `gmtime()`.

## grantpt - grant access to the slave pseudoterminal

**Syntax**      `#include <stdlib.h>`  
`int grantpt(int fildes);`

**Description** The `grantpt()` function changes the access permissions and the owner of the slave pseudoterminal assigned to its master counterpart. *fildes* is a file descriptor that was returned when the master pseudoterminal was opened successfully. A program with the `s` bit set for `root` is called (`/usr/lib/pt-chmod`). The user ID of the slave device is the same as the effective user ID of the calling process, and the group ID is set to a reserved group ID. The access permissions are set such that for the slave pseudoterminal reading and writing are permitted for the owner and writing is permitted for the group.

**Return value**

|    |                                                                        |
|----|------------------------------------------------------------------------|
| 0  | if successful.                                                         |
| -1 | if there is an error. <code>errno</code> is set to indicate the error. |

**Errors**      `grantpt()` will fail if:

|        |                                                         |
|--------|---------------------------------------------------------|
| EBADF  | <i>fildes</i> is not a valid open file descriptor       |
| EINVAL | <i>fildes</i> is not assigned to a main pseudoterminal. |
| EACCES | The corresponding slave device could not be accessed.   |

**Notes**      `grantpt()` will also fail if the application has implemented a signal handling routine to catch `SIGCHLD` signals.

**See also**      `open()`, `ptsname()`, `setuid()`, `unlockpt()`, `stdlib.h`.

## hsearch, hcreate, hdestroy - manage hash tables

**Syntax**

```
#include <search.h>

ENTRY *hsearch(ENTRY item, ACTION action);
int hcreate(size_t nel);
void hdestroy(void);
```

**Description** `hsearch()` is a hash-table search routine. It returns a pointer into a hash table indicating the location at which an entry can be found. The comparison function used by `hsearch()` is `strcmp()`. *item* is a structure of type `ENTRY` (defined in the `search.h` header) containing two pointers: *item.key* points to the comparison key (of type `char*`), and *item.data* (a `void*`) points to any other data to be associated with that key.

### Extension

Pointers to types that are not `void` must be converted to pointers to `void`. □

*action* is a member of the enumeration type `ACTION` (defined in `search.h`) indicating the disposition of the entry if it cannot be found in the table. `ENTER` indicates that *item* should be inserted in the table at an appropriate point.

### Extension

If a duplicate to an existing entry is present, the new item is not entered, and `hsearch()` returns the pointer to the existing entry. □

`FIND` indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a null pointer.

`hcreate()` allocates sufficient space for the table and must be called before `hsearch()` is used. The *nel* argument is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

`hdestroy()` destroys the search table and may be followed by another call to `hcreate()`. After the call to `hdestroy()`, the data can no longer be considered accessible.

### Return value

**Null pointer** `hsearch()`: if the action is `FIND` and the item could not be found, or if the action is `ENTER` and the table is full.  
`hcreate()`: if it cannot allocate sufficient space for the table.  
`hdestroy()` returns no value.

### Errors

`hsearch()` will fail if:

`ENOMEM` Insufficient storage space is available.

Notes `hsearch()` and `hcreate()` use `malloc()` to allocate space.

*Extension*

Only one hash search table may be active at a given time. □

See also `bsearch()`, `lsearch()`, `malloc()`, `strcmp()`, `tsearch()`, `search.h`.

## hypot - Euclidean distance function

Syntax `#include <math.h>`

`double hypot(double x, double y);`

Description `hypot()` computes the Euclidean distance.  $x$  and  $y$  are the coordinates of the point for which the Euclidean distance is to be computed.

Return value

`sqrt(x*x + y*y)` if successful.

`HUGE_VAL` if an overflow occurs.  
`errno` is set to indicate the error.

Errors `hypot()` will fail if:

`ERANGE` Overflow; the result is too large.

Notes If the result overflows, the program will abort (SIGFPE signal)!

See also `cabs()`, `sqrt()`, `math.h`.

## iconv - code conversion function

**Syntax**      `#include <iconv.h>`

```
size_t iconv(iconv_t cd, const char **inbuf, size_t *inbytesleft, char **outbuf,
             size_t *outbytesleft);
```

**Description** `iconv()` converts a sequence of characters from one codeset into a sequence of corresponding characters in another codeset. The sequence to be converted is located in the array specified by *inbuf*; the converted sequence is placed in the array specified by *outbuf*. The codesets are those specified in the `iconv_open()` call that returned the conversion descriptor *cd*. The *inbuf* argument points to a variable that points to the first character in the input buffer and *inbytesleft* indicates the number of bytes to be converted. The *outbuf* argument points to a variable that points to the first byte in the output buffer, and *outbytesleft* indicates the number of the bytes.

For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When `iconv()` is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, `iconv()` will place, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is not large enough to hold the entire reset sequence, `iconv()` will fail and set `errno` to `E2BIG`. Subsequent calls with *inbuf* as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified codeset, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output buffer to overflow. The variable pointed to by *inbuf* is updated to point to the byte following the last byte successfully used in the conversion. The value pointed to by *inbytesleft* is decremented to reflect the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is updated to point to the byte following the last byte of converted output data. The value pointed to by *outbytesleft* is decremented to reflect the number of bytes still available in the output buffer.

For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If `iconv()` encounters a character in the input buffer that is valid, but for which an identical character does not exist in the target codeset, `iconv()` performs an implementation-dependent conversion on this character.

**Return value**

`iconv()` updates the variables pointed to by the arguments to reflect the extent of the conversion and returns the number of non-identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by *inbytesleft* will be 0. If the input conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft* will be non-zero and `errno` is set to indicate the error condition. If an error occurs `iconv()` returns `(size_t)-1` and sets `errno` to indicate the error.

**Errors**

`iconv()` will fail if:

|        |                                                                                                           |
|--------|-----------------------------------------------------------------------------------------------------------|
| EILSEQ | Input conversion stopped due to an input byte that does not belong to the input codeset.                  |
| E2BIG  | Input conversion stopped due to lack of space in the output buffer.                                       |
| EINVAL | Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer. |
| EBADF  | The <i>cd</i> argument is not a valid conversion descriptor for an open file.                             |

**See also** `iconv_open()`, `iconv_close()`, `iconv.h`.

## iconv\_close - deallocate code conversion descriptor

**Syntax** `#include <iconv.h>`

```
int iconv_close(iconv_t cd);
```

**Description** `iconv_close()` deallocates the conversion descriptor *cd* and all other associated resources allocated by `iconv_open()`.

**Return value**

Upon successful completion, 0 is returned.  
Otherwise, -1 is returned and `errno` is set to indicate the error.

**Errors** `iconv_close()` will fail if:

|       |                                       |
|-------|---------------------------------------|
| EBADF | The conversion descriptor is invalid. |
|-------|---------------------------------------|

**See also** `iconv()`, `iconv_open()`, `iconv.h`.

## iconv\_open - allocate code conversion descriptor

**Syntax**      `#include <iconv.h>`

```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

**Description** `iconv_open()` returns a conversion descriptor that describes a conversion from the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified by the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion descriptor will be in a codeset-dependent initial shift state, ready for immediate use with `iconv()`.

A conversion descriptor remains valid in a process until that process closes it.

`iconv_open()` uses the `malloc()` function to allocate space for internal buffer areas. The `iconv_open()` function will fail if insufficient space is available for these buffers.

**Return value**

Conversion descriptor

for use on subsequent calls to `iconv()`.

Otherwise, `iconv_open()` returns `(iconv_t)-1` and sets `errno` to indicate the error.

**Errors**      `iconv_open()` will fail if:

|        |                                                                                                        |
|--------|--------------------------------------------------------------------------------------------------------|
| EMFILE | OPEN_MAX file descriptors are currently open in the calling process.                                   |
| ENFILE | Too many files are currently open in the system.                                                       |
| ENOMEM | Insufficient storage space is available.                                                               |
| EINVAL | The conversion specified by <i>fromcode</i> and <i>tocode</i> is not supported by the current version. |

**See also**      `iconv()`, `iconv_close()`, `iconv.h`.

## ilogb - get exponent part of floating-point number

**Syntax**      `#include <math.h>`  
                `int ilogb (double x)`

**Description** The `ilogb()` function returns the exponent part of *x*. In form, for all *x*s not equal to zero, the return value is the integral, signed part of  $\log_r |x|$ , where *r* is the base of the floating-point arithmetic of the processor (in BS2000, *r* = 16).

The `ilogb(x)` function call is equivalent to the `(int)logb(x)` call.

**Return value**  
Exponent part of *x*  
                    if successful.  
`INT_MIN`          if *x* = 0.0.

**See also**      `logb()`, `math.h`.

## index - get first occurrence of character in string

**Syntax**      `#include <strings.h>`  
                `char *index(const char *s, int c);`

**Description** `index()` searches for the first occurrence of character *c* in string *s* and returns a pointer to the located position in *s* if successful.

The terminating null byte (`\0`) is also treated as a character.

**Return value**  
Pointer to the position of *c* in string *s*,  
                    if successful.  
Null pointer      if *c* is not contained in string *s*.

**Notes**          `index()` and `strchr()` are equivalent.  
Portable applications should use the `strchr()` function instead of `index()`.

**See also**      `rindex()`, `strchr()`, `strings.h`.



## initstate, random, setstate, srandom - generate pseudo-random numbers

**Syntax**

```
#include <stdlib.h>

char *initstate(unsigned int seed, char *state, size_t size);

long random(void);

char *setstate(const char *state);

void srandom(unsigned int seed);
```

**Description** `random()` uses a non-linear, additive feedback random-number generator and uses a standard status array with the size of 31 long integers to generate successive pseudo-random numbers in the range 0 through  $2^{31}-1$ . The period of this random-number generator is very large - approximately  $16 \times (2^{31}-1)$ . The size of the status array determines the period of the random-number generator. If a larger status array is used, the period is extended.

With 256-byte status information the period of the random-number generator is greater than  $2^{69}$ .

Like `rand()`, `random()` generates by default a sequence of numbers which can be duplicated by calling `srandom()` with *seed* equals 1 beforehand.

`srandom()` initializes the current status array with the contents of *seed*.

The `initstate()` and `setstate()` functions handle the restart and the modification of random-number generators. With `initstate()` the status vector pointed to by the *state* argument can be initialized for later use. The *size* argument specifies the size of the status vector in bytes. `initstate()` uses *size* to establish how demanding the random-number generator used is to be - the more status information, the better the random numbers generated. Optimum values for the amount of status information are 8, 32, 64, 128 and 256 bytes; other specifications  $> 8$  are rounded down to the next lowest of these values. For values  $< 8$ , `random()` uses a simple, linear, congruent random-number generator. The *seed* argument determines the start value for the initialization via which a starting point for the random-number sequence is specified which also serves simultaneously for a restart. `initstate()` returns a pointer to the previous array with status information.

If `random()` is called without `initstate()` having been executed beforehand, `random()` behaves as if `initstate()` had been executed with *seed*=1 and *size*=128 beforehand.

Once a status has been initialized, the `setstate()` function allows a quick change of the status arrays. The status array pointed to by *state* is used for the generation of further random numbers until the next call of `initstate()` or `setstate()`. `setstate()` returns a pointer to the previous status array.

`initstate()` is not thread-safe. Use the reentrant function `rand_r()` when needed.

## Return value

random():

Pseudo-random number

The function is always successful.

initstate() and setstate():

Pointer to the previous status array

if successful.

Null pointer if an error occurs.

## Notes

After a status array has been initialized, it can be restarted in a different place:

- by calling `initstate()` with the desired start value, the status array and its size.
- by calling `setstate()` with the status array, followed by `srandom()` with the desired start value. The advantage of calling these two functions is that the size of the status vector does not need to be stored after its initialization.

## Example

The following statements initialize a status array, transfer an `initstate()` and output a random number generated with `random()`:

```
static long statel[32] = { 3, 0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd, 0x8c2e680f, 0xeb3d799f,
0xb11ee0b7, 0x2d436b86, 0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc, 0xde3b81e0, 0xdf0a6fb5,
0xf103bc02, 0x48f340fb, 0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
0xf5ad9d0e, 0x8999220b, 0x27fb47b9 };
```

```
main()
{
    unsigned seed;
    int n;
    seed = 1;
    n = 128;
    initstate(seed, statel, n);
    setstate(statel);
    printf("%d", random());
}
```

## See also

`drand48`, `drand48()`, `rand()`, `rand_r()`, `srand()`, `stdlib.h`.

## insque, remque - Insert element in queue or remove element from queue

**Syntax**

```
#include <search.h>

void insque(void *element, void *pred);

void remque(void *element);
```

**Description** `insque()` and `remque()` modify queues which are generated from double-concatenated elements. The queue can be concatenated in linear or ring form. For the `insque()` and `remque()` functions to be used, a structure must be defined in the application that first/initially contains two pointers to this structure. The other components of the structure are application-specific. The first pointer of the structure references the next entry in the queue. The second pointer references the previous entry in the queue. If the queue is linear, it is completed by null pointers. The names of the structure and the pointers it contains are freely selectable.

`insque()` inserts the element pointed to by *element* in a queue directly after *pred*.

`remque()` removes the element pointed to by *element* from a queue.

The call `insque(&element, NULL)`, where *element* is the first element in the queue, serves to initialize a linear list. Both pointers of *element* are occupied by null pointers.

To build up a ring-concatenated list, the application must first enter the address of the start element of the queue in both pointers of the start element.

## ioctl - control devices and STREAMS

Syntax `#include <stropts.h>`

```
int ioctl(int fildev, int request, .../ * arg */);
```

Description `ioctl()` executes a variety of control functions for devices and STREAMS. With non-stream files the functions executed by this call are undefined. The *request* argument and an optional third argument of varying type are forwarded to the file identified by *fildev* and are interpreted by the device driver.

*fildev* is an open file descriptor that refers to a device.

*request* selects the control function to be executed and depends on the respective devices addressed.

*arg* contains additional information required by these specific devices to execute the requested function. The data type of *arg* depends on the relevant control function, but is either an integer or a pointer to a device-specific data structure.

The following `ioctl()` commands, with the respective error IDs specified, can be applied to all stream files:

**I\_PUSH** 'Pushes' the module whose name is pointed to by *arg* into the beginning of the current stream, directly below the stream head. If the stream is a pipe, the module will be pushed between the stream heads of both ends of the pipe. This command then calls the `open()` function of the newly pushed-in module.

`ioctl()` with the **I\_PUSH** command will fail if:

|               |                                                                   |
|---------------|-------------------------------------------------------------------|
| <b>EINVAL</b> | Invalid module name.                                              |
| <b>EFAULT</b> | <i>arg</i> references a point outside the reserved address space. |
| <b>ENXIO</b>  | The <code>open()</code> function of the new module failed.        |
| <b>ENXIO</b>  | Hang-up signal received for <i>fildev</i> .                       |

**I\_POP** 'Pops' the module directly below the stream head out of the stream referenced by *fildev*. If a module is to be popped out of a pipe, it must be popped out from the side from which it was pushed in. In this case, *arg* should equal 0. In the event of an error, `errno` assumes one of the following values:

`ioctl()` with the **I\_POP** command will fail if:

|               |                                             |
|---------------|---------------------------------------------|
| <b>EINVAL</b> | No module stream exists.                    |
| <b>ENXIO</b>  | Hang-up signal received for <i>fildev</i> . |

**I\_LOOK** Determines the name of the module directly below the stream head in the stream specified by *fildev*, and stores it in a null-terminated string pointed to by *arg*. The buffer to which *arg* points should be at least `FMNAMESZ+1` bytes long. `FMNAMESZ` is defined in `stropts.h`.

`ioctl()` with the `I_LOOK` command will fail if:

- `EINVAL` There is no module in the stream.
- `EFAULT` *arg* references a point outside the reserved address space.

`I_FLUSH` Flushes all read and/or write queues, depending on the value of *arg*. *arg* can have any of the following values:

- `FLUSHR` Flush read queues.
- `FLUSHW` Flush write queues.
- `FLUSHRW` Flush read and write queues.

`ioctl()` with the `I_FLUSH` command will fail if:

- `EINVAL` Invalid value for *arg*.
- `EAGAIN` or `ENOSR` No buffer could be reserved for the flush message, because not enough `STREAMS` storage space was available.
- `ENXIO` Hang-up signal received for *fildes*.

`I_FLUSHBAND`

Flushes a particular band of messages. *arg* points to a `bandinfo` structure that has the following components:

```
unsigned char bi_pri;
int bi_flag;
```

The *bi\_flag* component can equal `FLUSHR`, `FLUSHW` or `FLUSHRW` (see above). The *bi\_pri* component determines the priority band.

`I_SETSIG` Informs the stream head that the user wants the system kernel to issue the `SIGPOLL` signal (see `signal()`) if a particular event occurs for the stream assigned to *fildes*. `I_SETSIG` supports the capability of asynchronous processing under `STREAMS`. The value of *arg* is a bit mask which specifies the events for which the signal is to be issued. It is the bit-wise `OR` of any combination of the following constants:

- `S_RDNORM` There is a message with normal priority (priority band = 0) at the head of the read queue of the stream head. A signal is issued even if the message has the length 0.
- `S_RDBAND` There is a message in the priority band > 0 at the head of the read queue of the stream head. A signal is issued even if the message has the length 0.
- `S_INPUT` A message not equal to `M_PCPROTO` (high priority) has arrived in the read queue of the stream head. This event is still supported for reasons of compatibility with earlier versions of UNIX System V. A signal is issued even if the message has the length 0.

- S\_HIPRI** There is a message with high priority (`M_PCPR0T0`) at the head of the read queue of the stream head. A signal is issued even if the message has the length 0.
- S\_OUTPUT** A write queue for regular data (priority band = 0) directly below the stream head is no longer full (without flow control). This informs the user that there is room in the queue to write or send normal data downstream.
- S\_WRNORM** Exactly like `S_OUTPUT`.
- S\_WRBAND** A write queue for data in the priority band  $\neq 0$ ) just below the stream head is no longer full. This informs the user that there is room in the queue to write or send priority data downstream.
- S\_MSG** An `M_SIG` or `M_PCSIG` message containing the `SIGPOLL` signal has reached the head of the read queue of the stream head.
- S\_ERROR** An `M_ERROR` message has reached the stream head.
- S\_HANGUP** An `M_HANGUP` message has reached the stream head.
- S\_BANDURG**  
If this event is used together with `S_RDBAND`, then `SIGURG` is generated instead of `SIGPOLL` if a high-priority message reaches the head of the read queue of the stream head.

If *arg* is 0, the calling process logs off again and does not receive any further `SIGPOLL` signals.

A user process can decide to only receive a signal in the event of messages with high priority, by setting the *arg* bit mask to the value `S_HIPRI`.

Processes which want to receive the `SIGPOLL` signal must sign on explicitly for its receipt by using `I_SETSIG`. If several processes sign on for this signal requesting the same event for the same stream, then every process receives the signal when the event occurs.

`ioctl()` with the `I_SETSIG` command will fail if:

- EINVAL** The value of *arg* is invalid or *arg* is 0 and the process does not sign on for receipt of the `SIGPOLL` signal.
- EAGAIN** The reservation of a data structure for the signal request failed.

**I\_GETSIG** Returns the events for which the calling process has currently signed on to receive a `SIGPOLL` signal. The events are returned in the bit mask pointed to by *arg*, where the events are the ones specified in the description of `I_SETSIG` (see above).

`ioctl()` with the `I_GETSIG` command will fail if:

`EINVAL` The process is not signed on for receipt of the `SIGPOLL` signal.

`EFAULT` *arg* references a point outside the reserved address space.

`I_FIND` Compares the names of all modules currently located in the stream with the name pointed to by *arg*. It returns the value 1 if the specified module is present in the stream. It returns the value 0 if the specified module is not popped in.

`ioctl()` with the `I_FIND` command will fail if:

`EINVAL` *arg* does not contain a valid module name.

`EFAULT` *arg* references a point outside the reserved address space.

`I_PEEK` Allows a user to read the information in the first message in the read queue of the stream head without removing the message from the queue. `I_PEEK` works in the same way as `getmsg()`, except that it does not remove the message from the queue.

*arg* points to an `strpeek` structure containing the following components:

```
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
```

The `maxlen` component in the `strbuf` structures *ctlbuf* and *databuf* (see `getmsg()`) must be set to the number of bytes to be read as control and/or data information. *flags* can be set to `RS_HIPRI` or to 0. If `RS_HIPRI` is set, `I_PEEK` searches for a message with high priority in the read queue of the stream head. Otherwise, `I_PEEK` searches for the first message in the read queue of the stream head.

`I_PEEK` returns the value 1 if a message was found, and 0 if no message was found in the read queue of the stream head or if *flags* was set to `RS_HIPRI` and no message with a high priority was found. This command does not wait for a message to arrive. After the return, *ctlbuf* supplies the information from the control section, *databuf* the information from the data section, and *flags* contains the value `RS_HIPRI` or 0.

`ioctl()` with the `I_PEEK` command will fail if:

`EFAULT` *arg* references a point outside the reserved address space, or the buffer area specified in *ctlbuf* or *databuf* is located outside the reserved address space.

`EBADMSG` The message to be read is invalid for `I_PEEK`.

`EINVAL` *flags* has an invalid value.

## I\_SRDOPT

Sets the setting for the reading (see `read()`) to the value of the *arg* argument. *arg* can take the following values:

RNORM     Byte-stream mode (default).

RMSGD     Message-discard mode.

RMSGN     Message-nondiscard mode.

If the value for *arg* is the result of bit-wise inclusive OR of RMSGD and RMSGN, this produces the error EINVAL. Bit-wise inclusive OR of RNORM with RMSGD produces RMSGN; bit-wise inclusive OR of RNORM with RMSGN produces RMSGD.

In addition, the handling of control messages by the stream head can be changed via the following identifiers for *arg*:

RPROTNORM

`read()` will fail with EBADMSG if there is a control message at the beginning of the read queue of the stream head. This is the default behavior.

RPROTDAT

Returns the control section of a message as data if a user calls `read()`.

RPROTDIS

Discards the control section of a message and delivers an existing data section if a user calls `read()`.

`ioctl()` with the I\_SRDOPT command will fail if:

EINVAL     *arg* does not have any of the valid values listed above.

I\_GRDOPT   Supplies the currently valid setting for the reading in the `int` variable pointed to by *arg*. The settings for the reading are described under `read()`.

`ioctl()` with the I\_GRDOPT command will fail if:

EFAULT     *arg* references a point outside the reserved address space.

I\_NREAD    Counts the number of data bytes in the data blocks of the first message in the read queue of the stream head and stores this number in the variable pointed to by *arg*. The result for this command is the number of messages in the read queue of the stream head. If, for example, the value 0 is returned in *arg*, but the `ioctl` call returns a result greater than 0, this indicates that the next message in the queue has the length 0.

`ioctl()` with the I\_NREAD command will fail if:

EFAULT     *arg* references a point outside the reserved address space.



## I\_FDINSERT

Generates a message from user-defined buffers, adds information about a different stream and sends the message downstream. The message contains a control section and an optional data section. The data and control sections to be sent are stored in separate buffers (see below).

*arg* points to an `strfdinsert` structure that has the following components:

```
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
int fildev;
int offset;
```

The *len* component in the `strbuf` structure *ctlbuf* (see `putmsg()`) must be the same as the size of a pointer plus the number of bytes for the control information of this message. *fildev* in the `strfdinsert` structure specifies the file descriptor of the other stream. *offset* must be aligned with a word boundary and specifies the number of bytes after which `I_FDINSERT` stores a pointer after the start of the control buffer. This pointer is the address of the read queue structure of the driver for the stream that corresponds to *fildev* in the `strfdinsert` structure. The *len* component in the `strbuf` structure *databuf* must be set to the same as the number of bytes to be sent as data information with the message, or 0 if no data section is to be sent.

*flags* indicates what type of message is to be generated. A normal message is generated if *flags* is 0, and a high-priority message is generated if *flags* is `RS_HIPRI`. With normal messages, `I_FDINSERT` blocks if the write queue of the stream is full because of the internal flow control. With high-priority messages, `I_FDINSERT` does not block in this case. With normal messages, `I_FDINSERT` does not block if the write queue is full but `O_NDELAY` or `O_NONBLOCK` is set. Instead, the call fails and `errno` is then `EAGAIN`.

`I_FDINSERT` also blocks if the call is waiting for the availability of message blocks and is not prevented from doing this because internal resources are missing. Here it is irrelevant which priority is set and whether `O_NDELAY` or `O_NONBLOCK` were specified. No partial message is sent.

`ioctl()` with the `I_FDINSERT` command will fail if:

**EAGAIN** A message without priority was specified, `O_NDELAY` or `O_NONBLOCK` is set and the write queue of the stream is full because of the internal flow control.

**EAGAIN or ENOSR**

No buffers could be reserved for the message to be generated, because there was too little storage space available under `STREAMS`.

- EINVAL** One of the following applies:
- *fildes* in the `strfdinsert` structure is not a valid open file descriptor for a stream.
  - The size of a pointer plus *offset* is greater than the *len* component of the buffer specified by *ctlptr*.
  - *offset* does not specify a correctly aligned location in the data buffer.
  - *flags* has an undefined value.
- ENXIO** A hang-up signal was received for *fildes* in the `ioctl` call or for *fildes* in the `strfdinsert` structure.
- ERANGE** The *len* component for the buffer specified by *databuf* is not in the range defined by the values for the maximum and minimum packet size of the highest module in the stream. Or the *len* component for the buffer specified by *databuf* is greater than the configured maximum size of the data section of a message. Or the *len* component for the buffer specified by *ctlbuf* is greater than the configured maximum size of the control section of a message.
- EFAULT** *arg* references a point outside the reserved address space, or the buffer area specified in *ctlbuf* or *databuf* is outside the reserved address space.

`I_FDINSERT` can also fail if an error message by the head of the stream is received which belongs to *fildes* in the `strfdinsert` structure. In this case, `errno` has the value in the message.

**I\_STR** Generates an internal `ioctl` message from the data to which *arg* points and sends this message downstream.

This mechanism is provided for sending user-defined `ioctl()` requests downstream to modules and drivers. It allows information to be sent with `ioctl()` and returns to the user all information that is sent upstream from the downstream recipient. `I_STR` blocks until the system responds with a positive or negative confirmation, or until a timeout occurs after a certain length of time. If a timeout occurs, the call fails with `errno` set to `ETIME`.

There can never be more than one active `I_STR` call in a stream. Other `I_STR` calls will block until the active `I_STR` call terminates at the stream head. The default value for a timeout with these requests is 15 seconds. `O_NDELAY` and `O_NONBLOCK` (see `open()`) do not affect this call.

For requests to be sent downstream, *arg* must point to an `strioc1` structure containing the following components:

```
int ic_cmd;
int ic_timeout;
int ic_len;
char *ic_dp;
```

*ic\_cmd* is the internal `ioctl()` command that is to be sent to a module located downstream or a driver, and *ic\_timeout* is the number of seconds for a timeout (-1 = infinite, 0 = default, > 0 = as specified). *ic\_len* is the number of bytes in the data argument and *ic\_dp* is a pointer to the data argument. The *ic\_len* component has two uses: at input it contains the length of the transferred data argument, and on return from the command it contains the number of bytes returned to the user (the buffer pointed to by *ic\_dp* should be large enough to hold the maximum length of the data to be returned from a module or driver).

The stream head converts the information in the `strioc1` structure into an internal `ioctl()` message and sends it downstream.

`ioctl()` with the `I_STR` command will fail if:

`EAGAIN` or `ENOSR`

Due to insufficient storage space, no buffer could be reserved for the `ioctl()` message.

`EINVAL` *ic\_len* is less than 0, or *ic\_len* is greater than the configured maximum size of the data section of a message, or *ic\_timeout* is less than -1.

`ENXIO` Hang-up signal received for *fildev*.

`ETIME` A downstream `ioctl()` call received a timeout before a confirmation was received.

`EFAULT` *arg* references a point outside the reserved address space, or the buffer area that was specified by *ic\_dp* and *ic\_len* (separate for sent and received data) is outside the reserved address space.

An `I_STR` call can also fail if an error message or hang-up signal message is received by the head of the stream while it is waiting for confirmation. In addition, an error ID can be returned in the positive or negative message if the `ioctl` command fails further downstream. In this case, `errno` has the value in the message.

- I\_SWROPT** Defines the settings for the writing, where the value of the *arg* argument is used. *arg* can have the following values:
- SNDZERO** Sends a message of length 0 downstream if a `write()` call with 0 bytes occurs.  
If in this case no message of length 0 is to be sent, then this bit must not be set in *arg*.
- `ioctl()` with the **I\_SWROPT** command will fail if:
- EINVAL** *arg* does not contain the value specified above.
- I\_GWROPT** Returns the currently valid setting for the writing in the `int` variable pointed to by *arg* (see under **I\_SWROPT**).
- I\_SENDFD** Requests the stream assigned to *fildev* to send a message containing a file pointer to the stream head at the other end of the pipe. The file pointer corresponds to the *arg* argument, which must be an open file descriptor.
- I\_SENDFD** converts *arg* into the corresponding file pointer. The command reserves a message block and inserts the file pointer in this block. The user ID and group ID of the sending process are also inserted. This message is entered directly in the read queue of the stream head at the other end of the pipe.
- `ioctl()` with the **I\_SENDFD** command will fail if:
- EAGAIN** The sending stream is not in a position to reserve a message block that can include the file pointer, or the read queue of the receiving stream head is full and cannot take the message sent by **I\_SENDFD**.
- EBADF** *arg* is not a valid open file descriptor.
- EINVAL** *fildev* is not linked with a pipe.
- ENXIO** Hang-up signal received for *fildev*.
- I\_RECVFD** Determines the assignment to an open file description of a message that was sent with the **I\_SENDFD** command for `ioctl()` via a pipe and reserves a new file descriptor in the calling process which refers to this open file description. *arg* is a pointer to a data buffer large enough to take an `strrecvfd` data structure. The `strrecvfd` structure is defined in `stropts.h` and contains the following components:
- ```
int fd;
uid_t uid;
gid_t gid;
char fill[8];
```
- fd* is a file descriptor. *uid* and *gid* are the user ID and group ID of the sending stream.

If `O_NDELAY` and `O_NONBLOCK` are not set (see `open()`), then `I_RECVFD` blocks until there is a message at the stream head. If `O_NDELAY` or `O_NONBLOCK` is set, `I_RECVFD` will fail, with `errno` equaling `EAGAIN`, if there is no message at the stream head.

If the message at the stream head is a message that was sent by `I_SENDFD`, a new user file descriptor is reserved for the file pointer contained in the message. The new file descriptor is stored in the `fd` component of the `strrecvfd` structure. The structure is copied into the data buffer of the user to which `arg` points.

`ioctl()` with the `I_RECVFD` command will fail if:

**EAGAIN** There is no message in the read queue of the stream head and `O_NDELAY` or `O_NONBLOCK` is set.

**EBADMSG** The message in the read queue of the stream head does not contain a transferred file descriptor.

**EMFILE** `NOFILES` file descriptors are already open.

**ENXIO** Hang-up signal received for *fildev*.

**EOVERFLOW**  
*uid* or *gid* is too big to be stored in the structure pointed to by *arg*.

**EFAULT** *arg* references a point outside the reserved address space.

**I\_LIST** Allows a user to output all module names in the stream, including the highest driver. If *arg* is zero, the result of the call is the number of modules (including drivers) in the stream referenced by *fildev*. This allows the user to reserve enough space for the module names. If *arg* is not zero, this argument should point to an `str_list` structure which has the following components:

```
int sl_nmods;
struct str_mlist *sl_modlist;
```

The `str_mlist` structure has the following components:

```
char l_name[FMNAMESZ+1];
```

*sl\_nmods* specifies the number of entries reserved by the user in the array. After the return, *sl\_modlist* contains the list of module names and *sl\_nmods* contains the number of entries in the *sl\_modlist* array; this is the number of all modules including the driver. The return value of `ioctl()` is 0. When the entries are written, they start at the top of the stream and continue downstream until either the end of the stream or the number of desired modules (*sl\_nmods*) is reached.

`ioctl()` with the `I_LIST` command will fail if:

`EINVAL` The `sl_nmods` component is less than 1.

`EAGAIN` or `ENOSR`  
Buffer could not be reserved.

#### `I_ATMARK`

Enables the user to check whether the current message in the read queue of the stream head was “marked” by a module further downstream. `arg` defines how the check is carried out if there can be more than one “marked” message in the read queue of the stream head. It can take the following values:

`ANYMARK` Check whether the message is marked.

`LASTMARK` Check whether the message is the last one marked in the queue.

The result is 1 if the relevant marking condition is fulfilled. Otherwise it is 0. In the event of an error, `errno` can have the following value:

`ioctl()` with the `I_ATMARK` command will fail if:

`EINVAL` The value of `arg` is invalid.

#### `I_CKBAND`

Checks whether a message exists in a given priority band in the read queue of the stream head. The result is 1 if such a message exists, or -1 in the event of an error. `arg` should be an integer containing the value of the priority band to be checked.

`ioctl()` with the `I_CKBAND` command will fail if:

`EINVAL` The value of `arg` is invalid.

#### `I_GETBAND`

Returns the priority band of the first message in the read queue of the stream head in the integer value pointed to by `arg`.

`ioctl()` with the `I_GETBAND` command will fail if:

`ENODATA` There is no message in the read queue of the stream head.

#### `I_CANPUT`

Checks whether a band can be written to. `arg` is the same as the priority band to be checked. The result is 0 if the priority band `arg` is subject to flow control, 1 if the band can be written to, or -1 for an error.

`ioctl()` with the `I_CANPUT` command will fail if:

`EINVAL` The value of `arg` is invalid.

**I\_SETCLTIME**

Enables a user to define how long the stream head is to wait if a stream is closed while there is still data in the write queue. Before it closes every module and every driver, the stream head waits the specified length of time so that the data can still be transferred. If there is still data in the queue after the wait time, this data is discarded. *arg* is a pointer to the number of milliseconds to be waited, always rounded up to the next highest valid value in the system. The default value is 15 seconds.

`ioctl()` with the `I_SETCLTIME` command will fail if:

`EINVAL` The value of *arg* is invalid.

**I\_GETCLTIME**

Returns the wait time when closing in the `long` variable pointed to by *arg*.

**Multiplex configurations under STREAMS**

**I\_LINK** Links two data streams, where *fildev* is the file descriptor of the stream linked to the multiplex driver, and *arg* is the file descriptor of the stream that is linked to another driver. The stream specified by *arg* is linked below the multiplex driver. `I_LINK` expects the multiplex driver to send a confirmation to the stream head. This call supplies a multiplexer identifier (this identifier is necessary for unlinking the multiplexer; see `I_UNLINK`) if successful and `-1` if an error occurs.

`ioctl()` with the `I_LINK` command will fail if:

`ENXIO` Hang-up signal received for *fildev*.

`ETIME` Timeout occurred before the confirmation was received by the stream head.

`EAGAIN` or `ENOSR`  
Insufficient memory available under `STREAMS` to execute `I_LINK`.

`EBADF` *arg* is not a valid open file descriptor.

`EINVAL` One of the following errors has occurred:

- The stream assigned to *fildev* does not support multiplexing.
- *arg* is not a stream, or is already linked under a multiplexer.
- The specified link would create a loop in the resulting configuration, i.e. a particular driver exists in more than one place in a multiplex configuration.
- *fildev* is the file descriptor of a pipe or a FIFO file.

The `I_LINK` operation can also fail if it waits for the multiplex driver to confirm the link request. This can happen if a message arrives at the stream head of *fildev*, indicating an error or a hang-up signal. In addition, the positive or negative confirmation can contain an error ID. In these cases, `I_LINK` fails, with `errno` equal to the value in the message.

## `I_UNLINK`

Cancels the link between the two data streams specified by *fildev* and *arg*. *fildev* is the file descriptor of the stream linked to the multiplex driver. *arg* is the multiplexer identifier that was returned by `I_LINK`. If *arg* equals `MUXID_ALL`, all data streams that were linked with *fildev* are unlinked. Like `I_LINK`, this command also expects the multiplex driver to confirm cancellation of the link.

`ioctl()` with the `I_UNLINK` command will fail if:

`ENXIO` Hang-up signal received for *fildev*.

`ETIME` Timeout occurred before a confirmation was received by the stream head.

`EAGAIN` or `ENOSR`

Not enough storage space can be reserved for the confirmation.

`EINVAL` *arg* is not a valid multiplexer identifier, or *fildev* is not the stream for which the `I_LINK` operation supplied by *arg* was executed.

`EINVAL` *fildev* is the file descriptor of a pipe or FIFO file.

The `I_UNLINK` operation can also fail if it waits for the multiplex driver to confirm the link request. This can happen if a message arrives at the stream head of *fildev* indicating an error or a hang-up signal. In addition, the positive or negative confirmation can contain an error ID. In these cases, `I_UNLINK` will fail, with `errno` having the value in the message.

## `I_PLINK`

Links two data streams, where *fildev* is the file descriptor of the stream that is linked to the multiplex driver, and *arg* is the file descriptor of the stream that is linked to another driver. The stream specified by *arg* is linked below the multiplex driver via a constant link. This call generates a constant link, which can also exist if the file descriptor *fildev*, which is assigned to the upper stream of the multiplex driver, is closed. `I_PLINK` expects the multiplex driver to send confirmation to the stream head. This call supplies a multiplexer identifier (this identifier is necessary for unlinking the multiplexer; see `I_PUNLINK`) if successful and -1 if an error occurs.

`ioctl()` with the `I_PLINK` command will fail if:

`ENXIO` Hang-up signal received for *fildev*.

`ETIME` A timeout occurred before a confirmation was received by the stream head.



EAGAIN or ENOSR

Insufficient memory available under STREAMS to execute I\_PLINK.

EBADF *arg* is not a valid open file descriptor.

EINVAL One of the following errors has occurred:

- The stream assigned to *fildev* does not support multiplexing.
- *arg* is not a stream, or it is already mounted under a multiplexer.
- The specified link would generate a loop in the resulting configuration, i.e. a particular driver exists in more than one place in a multiplex configuration.
- *fildev* is the file descriptor of a pipe or a FIFO file.

The I\_PLINK operation can also fail if it waits for the multiplex driver to acknowledge the link request. This can happen if a message arrives at the stream head of *fildev* indicating an error or a hang-up signal. In addition, the positive or negative acknowledgement can contain an error ID. In these cases, I\_PLINK fails, with `errno` having the value in the message.

## I\_PUNLINK

Cancels the constant link between the two data streams specified by *fildev* and *arg*. *fildev* is the file descriptor of the stream linked to the multiplex driver. *arg* is the multiplexer identifier that was returned by I\_PLINK when a stream was mounted under the multiplex driver. If *arg* equals MUXID\_ALL, all data streams that were connected to *fildev* via constant links are unmounted. Like I\_PLINK, this command also expects the multiplex driver to acknowledge the cancellation of the connection.

`ioctl()` with the I\_PUNLINK command will fail if:

ENXIO Hang-up signal received for *fildev*.

ETIME Timeout occurred before a confirmation was received by the stream head.

EAGAIN or ENOSR Buffer for the confirmation could not be reserved.

EINVAL Invalid multiplexer identifier.

EINVAL *fildev* is the file descriptor of a pipe or a FIFO file.

The I\_PUNLINK operation can also fail if it waits for the multiplex driver to acknowledge the link request. This can happen if a message arrives at the stream head of *fildev* indicating an error or a hang-up signal. In addition, the positive or negative acknowledgement can contain an error ID. In these cases, I\_PUNLINK fails, with `errno` set to the value in the message.

## Return value

non-negative integer

if successful. The value returned depends on the relevant device control function.

-1 if an error occurs. `errno` is set to indicate the error.

Errors `ioctl()` will fail with any file type if one or more of the following apply:

EBADF *fildev* is not a valid open file descriptor.

EINTR A signal was caught during the `ioctl()` system call.

EINVAL The stream or multiplexer identified by *fildev* is (directly or indirectly) mounted under a multiplexer.

`ioctl()` will also fail if the device driver detects an error. In this case, the error is forwarded to the caller by `ioctl()` without any changes. Not all of the errors listed below can occur with any driver:

EINVAL *request* or *arg* is not valid for this device.

EIO A physical I/O error has occurred.

ENOTTY *fildev* does not identify a device driver/STREAMS file which accepts control functions.

ENXIO *request* and *arg* are valid for this device driver but the requested service cannot be executed on this device.

ENODEV *fildev* identifies a valid STREAMS file, but the associated device driver does not support the `ioctl()` function.

ENOLINK *fildev* is located on a remote computer and the link to this computer is no longer active.

EFAULT *request* requests a data transfer to or from a buffer pointed to by *arg*, but part of the buffer is outside the address space allocated to the process.

If a stream is connected downstream from a multiplexer, every `ioctl()` command except for `I_UNLINK` and `I_PUNLINK` leads to the error `EINVAL`.

## See also

`streamio()` in “Programmer Reference Guide: STREAMS”,

`termio()` in “System Administrator Reference Guide”,

`close()`, `fcntl()`, `getmsg()`, `open()`, `pipe()`, `poll()`, `putmsg()`, `read()`, `sigaction()`, `write()`, `stropts.h`.

## isalnum - test for alphanumeric character

**Syntax**      `#include <ctype.h>`  
`int isalnum(int c);`

**Description** `isalnum()` tests whether the character `c` is a letter or a digit.

In all cases, the argument `c` is an `int`, the value of which must be representable as an `unsigned char` or must equal the value of the macro `EOF`. If the argument `c` has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Alphanumeric
<code>0</code>	Not alphanumeric

**Notes** `isalnum()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isalnum`).

The behavior of `isalnum()` is determined by the classes `alpha` and `digit` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also** `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`, `stdio.h`.

## isalpha - test for alphabetic character

**Syntax**      `#include <ctype.h>`  
`int isalpha(int c);`

**Description** `isalpha()` tests whether the character *c* is a letter.

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Letter
<code>0</code>	Not a letter

**Notes** `isalpha()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isalpha`).

The behavior of `isalpha()` is determined by the classes `alpha` and `digit` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also** `isalnum()`, `isalnum_l()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`, `stdio.h`.

## isascii - test for 7-bit ASCII character

**Syntax**      `#include <ctype.h>`

`int isascii (int c)`

**Description** `isascii()` tests whether *c* is less than 128.  
(The US-ASCII codeset is defined for values from 0 through 127).

`isascii()` is defined on all integer values.

*BS2000*

`isascii()` is a synonym for `isebcdic()`. `isascii()` tests whether the value of the character *c* represents an EBCDIC character (values 0 - 255). □

**Return value**

`≠ 0`              The value of *c* lies between 0 and 127 (ASCII character).

`0`                 Not an ASCII character (values `≠ 0` - 127).

*BS2000*

`≠ 0`              The value of *c* lies between 0 and 255 (EBCDIC character).

`0`                 Not an EBCDIC character (values `≠ 0` - 255). □

**Notes**          `isascii()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isascii`).

**See also**       `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `ctype.h`, `ascii_to_ebcdic()`, `ebcdic_to_ascii()`.

## isastream - test file descriptor

Syntax `#include <stropts.h>`  
`int isastream(int fildev);`

Description The `isastream()` function checks whether a file descriptor represents a STREAMS file. *fildev* refers to an open file.

### Return value

- 1 if *fildev* represents a STREAMS file.
- 0 if *fildev* does not represent a STREAMS file.
- 1 if an error occurs. `errno` is set to indicate the error.

Errors `isastream()` will fail if:

EBADF *fildev* is not a valid open file descriptor.

See also `stropts.h`.

## isatty - test for terminal device

Syntax `#include <unistd.h>`  
`int isatty(int fildev);`

Description `isatty()` tests whether the file descriptor specified with *fildev* is associated with a terminal device.

### Return value

- 1 if successful. *fildev* is associated with a terminal.
- 0 if an error occurs. `errno` is set to indicate the error.

Errors `isatty()` will fail if:

EBADF *fildev* is not a valid file descriptor.

ENOTTY *fildev* is not associated with a terminal.

See also `unistd.h`.

## isctrnl - test for control character

**Syntax**      `#include <ctype.h>`  
`int isctrnl(int c);`

**Description** `isctrnl()` whether the character *c* is a control character. Control characters are non-printable characters, e.g. for printer control.

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Control character
<code>0</code>	Not a control character

**Notes** `isctrnl()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isctrnl`).

The behavior of `isctrnl()` is determined by the class `cntrl` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also** `isalnum()`, `isalpha()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

## isdigit - test for decimal digit

**Syntax**      `#include <ctype.h>`  
`int isdigit(int c);`

**Description** `isdigit()` whether the character *c* is a decimal digit.

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Decimal digit
<code>0</code>	Not a decimal digit

**Notes**      `isdigit()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isdigit`).

The behavior of `isdigit()` is determined by the class `digit` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also**      `isalnum()`, `isalpha()`, `iscntrl()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `ctype.h`.



## isebcdc - test for EBCDIC character *(BS2000)*

**Syntax**      `#include <ctype.h>`  
                 `int isebcdc(int c);`

**Description** `isebcdc` tests whether the value of the character `c` represents an EBCDIC character (values 0 - 255).

**Return value**

<code>≠ 0</code>	The value of <code>c</code> represents an EBCDIC character (values 0 - 255).
<code>0</code>	Not an EBCDIC character (values <code>≠ 0</code> - 255).

**Notes**      `isebcdc` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isebcdc`).

`isebcdc` is a synonym for `isasci`.

**See also**    `isalpha()`, `isalnum()`, `isasci`, `isctrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`.

## isgraph - test for visible character

**Syntax**      `#include <ctype.h>`  
`int isgraph(int c);`

**Description** `isgraph()` tests whether *c* is a character with a visible representation, i.e. an alphanumeric or a special character. Spaces are not considered to be visible.

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Character with a visible representation
<code>0</code>	Not a character with a visible representation

**Notes** `isgraph()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isgraph`).

The behavior of `isgraph()` is determined by the class `graph` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also** `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

## islower - test for lowercase letter

**Syntax**      `#include <ctype.h>`  
`int islower(int c);`

**Description** `islower()` tests whether the character *c* is a lowercase letter.

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Lowercase letter
<code>0</code>	Not a lowercase letter

**Notes** `islower()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef islower`).

The behavior of `islower()` is determined by the class `lower` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also** `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

## isnan - test for NaN (not a number)

**Syntax**      `#include <math.h>`  
`int isnan(double x);`

**Description** `isnan()` tests whether *x* is not NaN.  
Not NaN means that *x* is a valid bit pattern of a floating-point number.

**Return value**

<code>0</code>	if <i>x</i> is not NaN.
----------------	-------------------------

**Notes** In this implementation, `isnan()` always returns the value `0`, i.e. all bit patterns for floating-point numbers are valid.

**See also** `math.h`.

## isprint - test for printing character

Syntax `#include <ctype.h>`

```
int isprint(int c);
```

Description `isprint()` tests whether *c* is a printing character, i.e. an alphanumeric character, a special character, or a space.

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

Return value

`≠ 0`            Printing character (alphanumeric, special character or space).

`0`              Non-printing character

Notes `isprint()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isprint`).

The behavior of `isprint()` is determined by the class `print` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

See also `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

## ispunct - test for punctuation character

**Syntax**      `#include <ctype.h>`

```
int ispunct(int c);
```

**Description** `ispunct()` tests whether *c* is a punctuation character, i.e. not a control, alphanumeric, or white-space character (see `isspace`).

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Punctuation character
<code>0</code>	Not a punctuation character

**Notes** `ispunct()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef ispunct`).

The behavior of `ispunct()` is determined by the class `punct` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also** `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

## isspace - test for white-space character

**Syntax**      `#include <ctype.h>`  
`int isspace(int c);`

**Description** `isspace()` tests whether *c* is a white-space character, i.e. a blank, horizontal tab, carriage return, newline, form-feed, or vertical tab.

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	White-space character
<code>0</code>	Not a white-space character

**Notes** `isspace()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isspace`).

The behavior of `isspace()` is determined by the class `space` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also** `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

## isupper - test for uppercase letter

**Syntax**      `#include <ctype.h>`  
`int isupper(int c);`

**Description** `isupper()` tests whether the character *c* is an uppercase letter.

In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Uppercase letter
<code>0</code>	Not an uppercase letter

**Notes**      `isupper()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isupper`).

The behavior of `isupper()` is determined by the class `upper` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also**      `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isxdigit()`, `setlocale()`, `ctype.h`.

## iswalnum - test for alphanumeric wide character

Syntax `#include <wchar.h>`

```
int iswalnum(wint_t wc);
```

Description `iswalnum()` tests whether the wide character `wc` is alphanumeric.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined

Return value

`≠ 0`            Alphanumeric

`0`                Not alphanumeric

Notes `iswalnum()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswalnum`).

The behavior of `iswalnum()` is determined by the classes `alpha` and `digit` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

See also `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`, `stdio.h`.



## iswalpha - test for alphabetic wide character

**Syntax**      `#include <wchar.h>`  
`int iswalpha(wint_t wc);`

**Description** `iswalpha` tests whether the wide character `wc` is alphabetic, i.e. a letter.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Alphabetic
<code>0</code>	Not alphabetic

**Notes** `iswalpha()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswalpha`).

The behavior of `iswalpha()` is determined by the class `alpha` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also** `iswalnum()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`, `stdio.h`.

## iswcntrl - test for control wide character

**Syntax**      `#include <wchar.h>`  
`int iswcntrl(wint_t wc);`

**Description** `iswcntrl()` tests whether the wide character `wc` is a control character. Control characters are non-printing characters, typically used for printer control.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Control wide character code
<code>0</code>	Not a control wide character code

**Notes** `iswcntrl()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswcntrl`).

The behavior of `iswcntrl()` is determined by the class `cntrl` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also** `iswalnum()`, `iswalpha()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

## iswctype - test wide character for class

**Syntax**      `#include <wchar.h>`  
                  `int iswctype(wint_t wc, wctype_t charclass);`

**Description** `iswctype()` tests whether the wide character `wc` has the character class `charclass`.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**Return value**

≠ 0                    Wide character in character class `charclass`

0                      Wide character not in character class `charclass`

**Notes**            The twelve strings "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit" are reserved for the standard character classes. In the table below, the functions in the left column are equivalent to the functions in the right column:

<code>iswalnum(wc)</code>	<code>iswctype(wc, wctype("alnum"))</code>
<code>iswalpha(wc)</code>	<code>iswctype(wc, wctype("alpha"))</code>
<code>iswcntrl(wc)</code>	<code>iswctype(wc, wctype("cntrl"))</code>
<code>iswdigit(wc)</code>	<code>iswctype(wc, wctype("digit"))</code>
<code>iswgraph(wc)</code>	<code>iswctype(wc, wctype("graph"))</code>
<code>iswlower(wc)</code>	<code>iswctype(wc, wctype("lower"))</code>
<code>iswprint(wc)</code>	<code>iswctype(wc, wctype("print"))</code>
<code>iswpunct(wc)</code>	<code>iswctype(wc, wctype("punct"))</code>
<code>iswspace(wc)</code>	<code>iswctype(wc, wctype("space"))</code>
<code>iswupper(wc)</code>	<code>iswctype(wc, wctype("upper"))</code>
<code>iswxdigit(wc)</code>	<code>iswctype(wc, wctype("xdigit"))</code>

The call `iswctype(wc, wctype("blank"))` does not have an equivalent `isw*` function.

**See also**        `wctype()`, `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `wchar.h`.

## iswdigit - test for decimal digit wide character

**Syntax**      `#include <wchar.h>`  
`int iswdigit(wint_t wc);`

**Description** `iswdigit()` tests whether the wide character `wc` is a decimal digit.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**Return value**

<code>≠ 0</code>	Decimal digit
<code>0</code>	Not a decimal digit

**Notes**      `iswdigit()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswdigit`).

The behavior of `iswdigit()` is determined by the class `digit` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also**    `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `wchar.h`.

## iswgraph - test for visible wide character

**Syntax**        `#include <wchar.h>`  
`int iswgraph(wint_t wc);`

**Description** `iswgraph()` tests whether the wide character specified by `wc` is a character with a visible representation, i.e. an alphanumeric or a special character. Spaces are not considered to be visible.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Character with a visible representation
<code>0</code>	Not a character with a visible representation

**Notes** `iswgraph()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswgraph`).

The behavior of `iswgraph()` is determined by the class `graph` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also** `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

## iswlower - test for lowercase wide character

Syntax      `#include <wchar.h>`  
`int iswlower(wint_t wc);`

Description `iswlower()` tests whether the wide character `wc` is a lowercase letter.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

Return value

<code>≠ 0</code>	Lowercase letter
<code>0</code>	Not a lowercase letter

Notes `iswlower()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswlower`).

The behavior of `iswlower()` is determined by the class `lower` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

See also `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

## iswprint - test for printing wide character

**Syntax**        `#include <wchar.h>`  
`int iswprint(wint_t wc);`

**Description** `iswprint()` tests whether `wc` is a printing wide character. Printing wide characters include alphanumeric characters, special characters, and blanks.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Printing character (alphanumeric character, special character or blanks)
<code>0</code>	Not a printing character

**Notes**        `iswprint()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswprint`).

The behavior of `iswprint()` is determined by the class `print` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also**     `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

## iswpunct - test for punctuation wide character

Syntax `#include <wchar.h>`

```
int iswpunct(wint_t wc);
```

Description `iswpunct()` tests whether `wc` is a punctuation wide character, i.e. not a control, alphanumeric or white-space wide character (see `iswspace`).

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

Return value

`≠ 0` Punctuation wide character

`0` Not a punctuation wide character

Notes `iswpunct()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswpunct`).

The behavior of `iswpunct()` is determined by the class `punct` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

See also `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.



## isspace - test for white-space wide character

Syntax `#include <wchar.h>`

```
int isspace(wint_t wc);
```

Description `isspace()` tests whether `wc` is a white-space wide character. White-space wide characters include: blanks, horizontal tabs, carriage returns, newlines, form-feeds, and vertical tabs.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

Return value

`≠ 0`            White-space wide character

`0`              Not a white-space wide character

Notes `isspace()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isspace`).

The behavior of `isspace()` is determined by the class `space` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

See also `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

## iswupper - test for uppercase wide character

Syntax `#include <wchar.h>`

```
int iswupper(wint_t wc);
```

Description `iswupper()` tests whether the wide character `wc` is an uppercase letter.

In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

Return value

<code>≠ 0</code>	Uppercase letter
<code>0</code>	Not an uppercase letter

Notes `iswupper()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswupper`).

The behavior of `isprint()` is determined by the class `print` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

The behavior of `iswupper()` is determined by the class `upper` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

See also `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

## iswxdigit - test for hexadecimal digit wide character

**Syntax**      `#include <wchar.h>`  
`int iswxdigit(wint_t wc);`

**Description** `iswxdigit` tests whether the wide character `wc` is a hexadecimal digit (0-9, A-F or a-f).  
In all cases, `wc` is an argument of type `wint_t`, the value of which must be a wide character code corresponding to a valid character in the current locale or must equal the value of the macro `WEOF`. If the argument `wc` has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Hexadecimal digit wide character code
<code>0</code>	Not a hexadecimal digit wide character code

**Notes** `iswxdigit()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef iswxdigit`).

The behavior of `iswxdigit()` is determined by the class `xdigit` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also** `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `wchar.h`.

## isxdigit - test for hexadecimal digit

**Syntax**      `#include <ctype.h>`  
`int isxdigit(int c);`

**Description** `isxdigit` tests whether the character *c* is a hexadecimal digit character (0-9, A-F or a-f). In all cases, the argument *c* is an *int*, the value of which must be representable as an *unsigned char* or must equal the value of the macro `EOF`. If the argument *c* has any other value, the behavior is undefined.

**Return value**

<code>≠ 0</code>	Hexadecimal digit
<code>0</code>	Not a hexadecimal digit

**Notes** `isxdigit()` is implemented both as a function and as a macro. To generate a function call, the definition of the macro name must be first undefined (`#undef isxdigit`). The behavior of `isxdigit()` is determined by the class `xdigit` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale()`.

**See also** `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `ctype.h`.

## j0, j1, jn - Bessel functions of first kind

Syntax        `#include <math.h>`  
  
              `double j0(double x);`  
              `double j1(double x);`  
              `double jn(int n, double x);`

Description `j0()`, `j1()` and `jn()` compute the Bessel functions of the first kind for floating-point values `x` and the integer orders 0, 1 or `n`.

Return value  
              Bessel value of `x` if successful.

See also     `y0()`, `y1()`, `yn()`, `math.h`.

## jrand48 - generate pseudo-random numbers between $-2^{31}$ and $2^{31}$ with initialization value

Syntax        `#include <stdlib.h>`  
  
              `long int jrand48 (unsigned short int xsubi[3]);`

Description See `drand48()`.

## kill - send signal to process or process group

Syntax `#include <signal.h>`

*Optional*

`#include <sys/types.h> □`

`int kill(pid_t pid, int sig);`

Description If the function is called with POSIX functionality, its behavior conforms with XPG4 as described below:

- `kill()` sends a signal *sig* to a process or a group of processes specified by *pid*, where *sig* is either one from the list given in `signal.h` or 0. If *sig* is 0 (the null signal), error checking is performed, but no signal is actually sent. The null signal can be used to check the validity of *pid*.
- `{_POSIX_SAVED_IDS}` is defined on all X/Open-conformant systems. For a process to have permission to send a signal to a process designated by *pid*, the real or effective user ID of the sending process must match the real or saved set-user-ID of the receiving process, unless the sending process has appropriate privileges.
- If *pid* is greater than 0, *sig* is sent to the process whose process ID is equal to *pid*.
- If *pid* is 0, *sig* is sent to all processes (excluding a number of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.
- If *pid* is -1, *sig* is sent to all processes (excluding system processes) for which the process has permission to send that signal.
- If *pid* is negative, but not -1, *sig* is sent to all processes whose process group ID is equal to the absolute value of *pid*, and for which the process has permission to send a signal.
- If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked, either *sig* or at least one pending unblocked signal is delivered to the sending process before `kill()` returns.
- No user ID test is applied when sending `SIGCONT` to a process that is a member of the same session as the sending process.
- `kill()` is successful if the process has permission to send *sig* to any of the processes specified by *pid*. If `kill()` fails, no signal is sent.

If threads are used, then the function affects the process or a thread in the following manner:

- A signal is sent to a process or a process group;  
The following applies to the (special) case caused by the value of *pid* in which *sig* is generated for the sending process: If the signal is not blocked for the calling thread and all

other threads of the process block the signal or do not wait for the signal in a `sigwait()` function, then `sig` (or at least a follow-up non-blocking signal) is sent to the sending thread before `kill()` returns.

– *BS2000*

The following deviations in behavior must be noted if the function is called with BS2000 functionality:

- `pid` must be 0, so the signal is sent to the calling process.
- The following subset of the signals defined in `signal.h` can be used for `sig`:

Signal	STXIT class	Meaning
SIGHUP	ABEND	Disconnection of link to terminal
SIGINT	ESCPBRK	Interrupt from the terminal with <code>[K2]</code>
SIGILL	PROCHK	Execution of an invalid instruction
SIGABRT	–	raise signal for program abort with <code>_exit(-1)</code>
SIGFPE	PROCHK	Error in a floating-point operation
SIGKILL	–	raise signal for program abort with <code>exit(-1)</code>
SIGSEGV	ERROR	Memory access with invalid segment access
SIGALRM	RTIMER	A time interval has elapsed (real time)
SIGTERM	TERM	Signal at program termination
SIGUSR1	–	Defined by the user
SIGUSR2	–	Defined by the user
SIGDVZ	PROCHK	Division by 0
SIGXCPU	RUNOUT	CPU time has run out
SIGTIM	TIMER	A time interval has elapsed (CPU time)
SIGINTR	INTR	SEND-MESSAGE command



Return value

- 0 upon successful completion.
- 1 if an error occurs. `errno` is set to indicate the error.

Errors

`kill()` will fail if:

- EINVAL The value of the `sig` argument is an invalid or unsupported signal number.
- EPERM The process does not have permission to send the signal to any receiving process.  
*BS2000*  
EPERM is not supported. □
- ESRCH No process or process group can be found corresponding to that specified by `pid`.

See also `getpid()`, `raise()`, `setsid()`, `sigaction()`, `signal.h`, `sys/types.h`.

## killpg - send signal to process group

**Syntax**      `#include <signal.h>`  
`int killpg(pid_t pgrp, int sig);`

**Description** `killpg()` sends the signal *sig* to the process group *pgrp*.

The real or effective user ID of the sending process must match the real or saved “set-user-ID” of the receiving process, unless the effective user ID of the sending process comes from a user with corresponding permission. The only exception is the SIGCONT signal, which can always be sent to any successor of the current process.

If *pgrp* is greater than 1, `killpg(pgrp, sig)` corresponds to the call of `kill(-pgrp, sig)`. If *pgrp* is less than or equal to 1, the behavior of `killpg()` is undefined.

**Return value**  
See `kill()`.

**Errors**      See `kill()`.

**See also**    `getpgid()`, `getpid()`, `kill()`, `raise()`, `signal.h`.



## I64a - convert 32-bit integer number to string

Syntax `#include <stdlib.h>`  
`char *I64a (long value);`

Description See `a641()`.

## labs - return long integer absolute value

Syntax `#include <stdlib.h>`  
`long int labs(long int j);`

Description `labs()` computes the absolute value of an integer *j* of type `long`.

Return value  
Absolute value of the long integer *j* if successful.

Notes The absolute value of the negative integer with the largest magnitude is not representable. If a negative number with the highest magnitude ( $-2^{31}$ ) is specified as the argument *j*, the program will terminate with an error.

See also `abs()`, `cabs()`, `stdlib.h`.

## lchown - change owner/group of file

Syntax `#include <unistd.h>`

```
int lchown(const char *path, uid_t owner, gid_t group);
```

Description The `lchown()` function sets the owner and the group affiliation of the specified file. This is the same as `chown()` unless the file consists of a symbolic link. In this case `lchown()` changes the ownership of the link file, whereas `chown()` changes the ownership of the file or directory to which the link refers.

If `chown()`, `lchown()` or `fchown()` is called by a process that does not have system administrator status, the bit for setting the user and group IDs on execution, or `S_ISUID` and `S_ISGID`, is deleted [see `chmod()`].

The operating system has the configuration option `_POSIX_CHOWN_RESTRICTED` to prevent affiliation changes for `chown()`, `lchown()` and `fchown()` system calls. In POSIX, `_POSIX_CHOWN_RESTRICTED` is active, therefore the `chown()`, `lchown()` and `fchown()` system calls protect the owner of a file from having the owner IDs of his/her files changed, and they restrict the group change of the file to the list of supplementary group IDs.

After successful completion, `chown()`, `lchown()` and `fchown()` mark the `ST_CTIME` field of the file for update.

Return value

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error. If -1 is returned, the user ID and group ID of the file are not changed.

Errors

`lchown()` will fail if:

`EACCES` Search permission is denied for a component of *path*.

`EINVAL` The value of the specified user ID or group ID is not supported, e.g. if the value is less than 0, or an attempt was made to access a BS2000 file.

`ENAMETOOLONG` The length of the pathname exceeds `{PATH_MAX}`, or the length of a component of the pathname exceeds `{NAME_MAX}`.

`ENOENT` A component of the pathname does not exist, or *path* points to an empty string.

`ENOTDIR` A component of the pathname prefix is not a directory.

`EOPNOTSUPP` The *path* argument identifies a symbolic link and the implementation does not support changing the owner or the group affiliation of a symbolic link.

ELOOP	Too many symbolic links were encountered in resolving <i>path</i> .
EPERM	The effective user ID does not match the owner of the file, and the calling process does not have the appropriate access permissions.
EROFS	The file resides on a read-only file system.
EIO	An I/O error occurred while reading from or writing to the file system.
EINTR	A signal was caught during execution of the function.
ENAMETOOLONG	The resolution of symbolic links in the pathname leads to an interim result whose length exceeds <code>{PATH_MAX}</code> .

**See also** `chmod()`, `chown()`, `symlink()`, `unistd.h`.

## lcong48 - pseudo-random number (signed long int) generator

**Syntax**      `#include <stdlib.h>`  
                  `void lcong48 (unsigned short int param[7]);`

**Description** See `drand48()`.

## ldexp - load exponent of floating-point number

**Syntax**      `#include <math.h>`  
                  `double ldexp(double x, int exp);`

**Description** `ldexp()` computes the quantity:  
 $x * 2^{exp}$   
 where *x* is the mantissa and *exp* is the exponent.  
`ldexp()` is the inverse function of `frexp()`.

**Return value**  
 Value of  $x * 2^{exp}$   
                  if successful.  
 +/-HUGE\_VAL (depending on the sign of *x*) if an overflow occurs.  
                  `errno` is set to indicate the error.

**Errors**      `ldexp()` will fail if:  
                  ERANGE      **Overflow.**

**See also**      `frexp()`, `modf()`, `math.h`.

## ldiv - long division of integers

**Syntax**      `#include <stdlib.h>`

`ldiv_t ldiv(long int numer, long int denom);`

**Description** `ldiv()` computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*.

Both the arguments and the result are of type `long int`.

The sign of the quotient is the same as the sign of the algebraic quotient. The value of the quotient is the highest integer less than or equal to the absolute value of the algebraic quotient.

The remainder is expressed by the following equation:

Quotient \* Divisor + Remainder = Dividend

**Return value**

Structure of type `ldiv_t`

if successful. The structure includes the quotient `quot` as well as the remainder `rem` as long values.

**See also**      `div()`, `stdlib.h`.

## lfind - find entry in linear search table

**Syntax**      `#include <search.h>`

`void *lfind(const void *key, const void *base, size_t *nelp, size_t width  
int (*compar) (const void *, const void *))`

**Description** See `lsearch()`.

## lgamma - compute logarithm of gamma function

**Syntax**

```
#include <math.h>

double lgamma(double x);

extern int signgam;
```

**Description** `lgamma()` computes the mathematical gamma function for a given floating-point number  $x$ :

$$\int_0^{\infty} e^{-t} t^{x-1} dt$$

The sign of this value is stored as +1 or -1 in the internal C variable `signgam`. The `signgam` variable may not be defined by the user.

**Return value**

`lgamma(x)` if successful.  
`HUGE_VAL` if the correct value results in an overflow. `errno` is set to indicate the error.  
`HUGE_VAL` if  $x$  is a non-positive integer. `errno` is set to indicate the error.

**Errors** `lgamma()` will fail if:

`ERANGE` Overflow; the return value is too large.  
`EDOM`  $x$  is a non-positive integer.

**See also** `gamma()`, `math.h`.

## `__LINE__` - macro for current source program line number

**Syntax** `__LINE__`

**Description** This macro generates the current line number of the source program as a decimal number.

**Notes** This macro need not be defined in an header file. Its name is recognized and replaced by the compiler.

## link - create link to file

**Syntax**      `#include <unistd.h>`  
                  `int link(const char *path1, const char *path2);`

**Description** `link()` creates a new link (directory entry) for the existing file, *path1*.

*path1* points to a pathname naming an existing file. *path2* points to a pathname naming the new directory entry to be created. The `link()` function will atomically create a new link for the existing file, and the link count of the file is incremented by one.

If *path1* names a directory, `link()` will fail.

Upon successful completion, `link()` will mark for update the `st_ctime` structure component of the file. The `st_ctime` and `st_mtime` fields of the directory that contains the new entry are also marked for update.

If the `link()` function fails, no link is created, and the link count of the file remains unchanged.

The calling process must have permission to access the existing file.

`link()` is not executed between files of different file systems.

If `link(*path1, *path2)` is called successfully, and both *path1* and *path2* point to files of the POSIX file system, an internal link count is incremented by 1. Similarly, any successful call to `unlink(*path)` or `remove(*path)` decreases this link count by 1. If the count = 0 and the file is no longer open for any process, the file is deleted.

### Return value

0                    if successful

-1                   if the process that calls `link()` is denied access to the file involved; `errno` is set to `EACCES` to indicate the error.  
                       for a call with a BS2000 file (`/BS2/name`); `errno` is set to `EINVAL`.

### Errors

`link()` will fail if:

`EACCES`            Search permission is denied for a component of either path prefix, or the requested link requires writing in a directory with a mode that denies write permission, or the calling process does not have permission to access the existing file.

`EEXIST`            The link named by *path2* exists.

*Extension*

EFAULT	<i>path1</i> or <i>path2</i> points outside the allocated address space.
EINTR	A signal was caught during the <code>link()</code> system call.
EINVAL	An attempt was made to access a BS2000 file.
ELOOP	Too many symbolic links were encountered in resolving <i>path1</i> or <i>path2</i> . □
EMLINK	The number of links to the file named by <i>path1</i> would exceed <code>{LINK_MAX}</code> .
ENAMETOOLONG	The length of <i>path1</i> or <i>path2</i> exceeds <code>{PATH_MAX}</code> , or a pathname component is longer than <code>{NAME_MAX}</code> .
ENOENT	A component of either path prefix does not exist; the file named by <i>path1</i> does not exist; or <i>path1</i> or <i>path2</i> points to an empty string.
ENOSPC	The directory to contain the link cannot be extended.
ENOTDIR	A component of one of the paths is not a directory.
EPERM	The file named by <i>path1</i> is a directory, and the process does not have appropriate privileges.
EROFS	The requested link requires writing in a directory on a read-only file system.
EXDEV	The link named by <i>path2</i> and the file named by <i>path1</i> are on different file systems.

**Notes** `link()` is executed only for POSIX files

**See also** `readlink()`, `remove`, `symlink()`, `unlink()`, `unistd.h`.



## llabs - return absolute value of an integer (long long int)

**Syntax**      `#include <stdlib.h>`  
`long long int llabs(long long int j);`

**Description** `llabs()` computes the absolute value of an integer *j* of type `long long int`.

**Return value**  
`|j|`              for an integer *j*.  
`undefined`        for overflow or underflow. `errno` is set to `ERANGE` to indicate an error.

**Errors**        `llabs()` fails if:  
`ERANGE`        The absolute value of the negative integer of type `long long int` with the largest magnitude is not representable. If a negative number with the highest magnitude is specified as the argument *j*, the program will terminate with an error.

**See also**      `abs()`, `cabs()`, `labs()`

## lldiv - division of integers (long long int)

**Syntax**      `#include <stdlib.h>`

```
lldiv_t lldiv(long long int dividend, long long int divisor);
```

**Description** `lldiv()` computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. Both the arguments and the result are of type `long long int`.

The sign of the quotient is the same as the sign of the algebraic quotient. The value of the quotient is the highest integer less than or equal to the absolute value of the algebraic quotient.

The remainder is expressed by the following equation:

$$\text{Quotient} * \text{Divisor} + \text{Remainder} = \text{Dividend}$$

**Return value**

Structure of type `lldiv_t`, if successful. The structure includes the quotient *quot* as well as the remainder *rem* as `long long` values.

**See also**      `div()`, `ldiv()`

## llrint, llrintf, llrintl - round to nearest integer value (long long int)

**Syntax**

```
#include <math.h>

long long int llrint(double x);
long long int llrintf (float x);
long long int llrintl (long double x);
```

**Description** The functions return the integer value (displayed as a number of type `long long int`) nearest to  $x$ .

The returned value is rounded according to the currently set rounding mode of the computer. If the default mode is set to 'round-to-nearest' and the difference between  $x$  and the rounded result is exactly 0.5, the next even integer is returned.

If the currently set rounding mode rounds infinitely in the positive direction, `llrint()` is identical to `ceil()`. If the currently set rounding mode rounds infinitely in the negative direction, `llrint()` is identical to `floor()`.

In this version the rounding mode is set to round infinitely in the positive direction.

### Return value

**Integer** value (type `long long int`) nearest to  $x$  if successful.

**Undefined** for overflow or underflow. `errno` is set to `ERANGE` to indicate an error.

**Errors** `llrint()`, `llrintf()`, `llrintl()` fails if:

**ERANGE** The value is too large, and `errno` is set to indicate an error.

**See also** `abs()`, `ceil()`, `floor()`, `llround()`, `lrint()`, `lround()`, `rint()`, `round()`

## llround, llroundf, llroundl - round up to next integer value (long long int)

**Syntax**

```
#include <math.h>

long long int llround(double x);

long long int llroundf (float x);

long long int llroundl (long double x);
```

**Description** The functions return the integer value (displayed as a number of type `long long int`) nearest to  $x$ .

The returned value does not depend on the rounding mode currently set. If the difference between  $x$  and the rounded result is exactly 0.5, the next highest integer is returned.

### Return value

**Integer** value (type `long long int`) nearest to  $x$  if successful.

**Undefined** for overflow or underflow. `errno` is set to `ERANGE` to indicate an error.

**Errors** `llround()`, `llroundf()`, `llroundl()` fails if:

**ERANGE** The value is too large, and `errno` is set to indicate an error.

**Errors** `abs()`, `ceil()`, `floor()`, `llrint()`, `lrint()`, `lround()`, `rint()`, `round()`

## loc1, loc2 - pointers to characters matched by regular expressions

Syntax      `#include <regexp.h>`  
             `extern char *loc1;`  
             `extern char *loc2;`

Description See `regexp()`.

Notes        This function will not be supported by the X/Open standard in the future.

             New applications should use `fnmatch()`, `glob()`, `regcomp()` and `regexec()`, which guarantee full internationalized regular expression functionality (see "Regular expressions" in the manual "POSIX Commands (BS2000/OSD)").

## localeconv - change components of locale

Syntax `include <locale.h>`

```
struct lconv *localeconv(void);
```

Description `localeconv()` sets the components of a structure of type `struct lconv` (defined in `locale.h`) with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the current locale.

The `*char` members of the structure `lconv` are pointers to strings, any of which (except `decimal_point`) can point to "", to indicate that the value is not available in the current locale or is of zero length.

The `*char` members of the structure `lconv` are non-negative numbers, any of which can assume the value `{CHAR_MAX}` (see `limits.h`), to indicate that the value is not available in the current locale.

The members for non-monetary numeric values (`LC_NUMERIC`) are interpreted as follows:

```
char *decimal_point
```

The radix character used to format non-monetary quantities.

```
char *thousands_sep
```

The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.

```
char *grouping
```

A string whose elements taken as one-byte integer values indicate the size of each group of digits in non-monetary quantities (see below).

The members for monetary numeric values (`LC_MONETARY`) are interpreted as follows:

```
char *int_curr_symbol
```

The international currency symbol used in the current locale. The operand consists of a string of four characters: the first three characters contain the alphabetic international currency symbol, as defined in ISO 4217:1897; the fourth character, which immediately precedes the null byte, is the separator between the international currency symbol and the monetary quantity.

In the "De.EDF04F@euro" locale the value "EUR" is entered as an alphabetic international currency symbol.

```
char *currency_symbol
```

The local currency symbol applicable to the current locale.

```
char *mon_decimal_point
```

The radix character used to format monetary quantities. This member is restricted to one byte in the ISO-C standard. If a multi-byte operand is specified, the result is undefined.

char \*mon\_thousands\_sep

The separator for groups of digits before the decimal point in formatted monetary quantities. This member is restricted to one byte in the ISO-C standard. If a multi-byte operand is specified, the result is undefined.

char \*mon\_grouping

A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted monetary quantities. The operand consists of a sequence of integers, delimited by semi-colons. Each number specifies the number of positions in each group; the first number indicates the size of the group that immediately precedes the decimal separator, and the following numbers define the preceding groups. If the last number is not equal to -1, the preceding group (if one exists) is repeatedly used for the remaining positions. If the last number is -1, no further grouping is performed (see below).

char \*positive\_sign

The string used to indicate a non-negative, formatted monetary quantity.

char \*negative\_sign

The string used to indicate a negative formatted monetary quantity.

char int\_frac\_digits

The number of decimal places to be displayed in internationally formatted monetary quantities, where `int_curr_symbol` is used.

char frac\_digits

The number of decimal places to be displayed in a formatted monetary quantity, where `currency_symbol` is used.

char p\_cs\_precedes

Set to 1 if `currency_symbol` or `int_curr_symbol` precedes the value for a non-negative formatted monetary quantity. Set to 0 if either of these symbols follows the value.

char p\_sep\_by\_space

Set to 0 if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a non-negative formatted monetary quantity. Set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

char n\_cs\_precedes

If the value of this member is 1, the `currency_symbol` or `int_curr_symbol` precedes the value for a negative formatted monetary quantity. Otherwise, the member is set to 0.

char n\_sep\_by\_space

Set to 0 if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a negative formatted monetary quantity. Set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

char p\_sign\_posn

This member is set to a value that indicates the position of the `positive_sign` for a non-negative formatted monetary quantity (see below).

char n\_sign\_posn

Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity (see below).

The elements of `grouping` and `mon_grouping` are interpreted as follows:

CHAR-MAX No further grouping is to be performed.

0 The previous element is to be repeatedly used 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:

0 Parentheses surround the quantity and `currency_symbol` or `int_curr_symbol`.

1 The sign precedes the quantity and `currency_symbol` or `int_curr_symbol`.

2 The sign follows the quantity and `currency_symbol` or `int_curr_symbol`.

3 The sign immediately precedes the `currency_symbol` or `int_curr_symbol`

4 The sign immediately follows the `currency_symbol` or `int_curr_symbol`.

The implementation will behave as if no function calls `localeconv()`.

## Return value

Pointer to the structure in which the values were entered upon successful completion.



**Notes** The structure pointed to by the return value must not be modified by the program, but may be overwritten by a subsequent call to `localeconv()`. In addition, calls to `setlocale()` with the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

**Example** The following table illustrates the rules which may be used by three countries to format monetary quantities:

Country	Positive format	Negative format	International format
Germany	EUR 1.234,56	-EUR 1.234,56	EUR 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these three countries, the respective values for the monetary members of the structure returned by `localeconv()` are:

Component value	Germany	Norway	Switzerland
<code>int_curr_symbol</code>	"EUR"	"NOK "	"CHF "
<code>currency_symbol</code>	"?"	"kr"	"SFrs."
<code>mon_decimal_point</code>	","	","	."
<code>mon_thousands_sep</code>	."	."	","
<code>mon_grouping</code>	3;3	"\3"	"\3
<code>positive_sign</code>	" "	" "	" "
<code>negative_sign</code>	"_"	"_"	"C"
<code>int_frac_digits</code>	2	2	2
<code>frac_digits</code>	2	2	2
<code>p_cs_precedes</code>	0	1	1
<code>p_sep_by_space</code>	1	0	0
<code>n_cs_precedes</code>	0	1	1
<code>n_sep_by_space</code>	1	0	0
<code>p_sign_posn</code>	1	1	1
<code>n_sign_posn</code>	1	2	2

**See also** `isalpha()`, `isascii()`, `nl_langinfo()`, `printf()`, `scanf()`, `setlocale()`, `strcat()`, `strchr()`, `strcmp()`, `strcoll()`, `strcpy()`, `strftime()`, `strlen()`, `strpbrk()`, `strspn()`, `strtok()`, `strxfrm()`, `strtod()`, `langinfo.h`, `local.h`, **section "Locale" on page 53.**

## localtime - convert date and time to local time

Syntax `#include <time.h>`

```
struct tm *localtime(const time_t *clock);
```

Description `localtime()` accepts arguments of type `time_t`, which are pointed to by `clock`, and which supply the time in seconds since 00:00:00 UTC (Universal Time Coordinated, January 1, 1970).

`localtime()` returns pointers to the `tm` structures explained below.

The `localtime()` function corrects for the timezone and any seasonal time adjustments.

The declarations of all functions, external values, and of the `tm` structure are contained in the header `time.h`. The `tm` structure is defined as follows:

```
struct    tm {
    int    tm_sec;        /* Seconds - [0, 61] for skipped seconds */
    int    tm_min;        /* Minutes - [0, 59] */
    int    tm_hour;       /* Hours - [0, 23] */
    int    tm_mday;       /* Day of month - [1, 31] */
    int    tm_mon;        /* Months - [0, 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;      /* Option for daylight saving time */
};
```

`tm_isdst` is positive if daylight saving time is set, null if daylight saving time is not set, and negative if the information is not available.

`localtime()` is not thread-safe. Use the reentrant function `localtime_r()` when needed.

### *BS2000*

`localtime()` interprets the time specification of type `time_t` as the number of seconds that have elapsed since January 1, 1950, 00:00:00. From this number, `localtime()` calculates the date and time and stores the result in a structure of type `tm`.

In this implementation, `localtime()` is equivalent to `gmtime()`; both functions return the local time. □

Return value

Pointer to the `tm` structure.

- Notes**        The return values of `localtime()` point to static data that is overwritten with each call.
- `localtime()` does not support local date and time formats; to ensure maximum portability, `strptime()` should be used instead.
- `localtime()` writes its result to an internal C data area that is overwritten with each call.
- Furthermore, `localtime()` and `gmtime()` use the same data area, which means that if they are called in succession, the result of the first call will be overwritten.
- See also**     `altzone`, `ctime()`, `daylight`, `gmtime()`, `localtime_r()`, `strptime()`, `tzname`, `tzset()`, `time.h`.

## localtime\_r - convert date and time to string (thread-safe)

**Syntax**        `#include <time.h>`

```
struct tm *localtime_r(const time_t *clock, struct tm *result);
```

**Description** `localtime_r()` converts the time value pointed to by *clock* to exactly the same time format as `localtime()` and writes the result in the memory area pointed to by *result* (with at least 26 bytes).

**Return value**

Pointer	to a string pointed to by <i>result</i> if successful.
Null pointer	if an error occurs.

**See also**     `asctime()`, `asctime_r()`, `ctime()`, `ctime_r()`, `localtime()`, `time()`.

## lockf - lock file section

**Name** lockf, lockf64

**Syntax** #include <unistd.h>

```
int lockf(int fildes, int function, off_t size);
int lockf64(int fildes, int function, off64_t size);
```

**Description** `lockf()` is used to lock file sections, whereby recommended or mandatory write locks depend on the respective mode bits of the file (see `chmod()`). Lock calls from other processes attempting to lock an already locked file section either cause an error value to be returned or they pause until the resource is released. All locks for a process are removed if the process is terminated. `lockf()` can be used on normal files.

*fildes* is an open file descriptor. The file descriptor must have `O_WRONLY` or `O_RDWR` permission so that the lock can be set up with this function call.

*function* is control value which specifies the measures to be taken. The permissible values for *function* are defined as follows in `unistd.h`:

```
#define F_ULOCK 0 /* Release locked section */
#define F_LOCK 1 /* Lock section exclusively */
#define F_TLOCK 2 /* Test section and lock it exclusively */
#define F_TEST 3 /* Test section for locks of other processes */
```

All other values of *function* are reserved for future extensions and lead to an error message if they are not implemented.

`F_TEST` is used to determine whether a section contains a lock from another process. `F_LOCK` and `F_TLOCK` each lock a section of a file if this section is available. `F_ULOCK` removes the locks of a file section.

*size* is the number of contiguous bytes to be locked or unlocked. The resource to be locked or unlocked begins at the current offset in the file and extends forward for a positive *size* and backward for a negative *size* (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset to the largest file offset is locked, i.e. from the current offset up to the current or any future end of file. An area does not need to be allocated to a file in order to be locked, because these locks can also extend beyond the end of the file.

The sections locked with `F_LOCK` or `F_TLOCK` can contain or be contained in all or part of a section which was previously locked by the same process. If this situation occurs in this or neighboring sections, the sections are combined into one section. If the request requires a new element to be added to the table of active locks and this table is already full, an error message is issued and the new section is not locked.

The requirements of `F_LOCK` and `F_TLOCK` differ only in the action that is taken if the resource is not available. `F_LOCK` causes the calling process to pause until the resource is available. `F_TLOCK` causes the function to return -1 and set `errno` to the `EACCES` error if the section is already locked by another process.

Locked sections are released by the first `close` call issued by the process which set the lock for a file descriptor of the associated file.

`F_ULOCK` requests can fully or partially release one or more locked sections controlled by the process. Locked sections are unlocked as of the point of the offset until *size* bytes have been unlocked or until the end of the file if *size* has the value `(off_t)0`. If the sections are not fully unlocked, the remaining sections stay locked by the process. The release of the middle segment of a locked section requires an additional entry in the table of active locks. If this table is full, `errno` is set to `ENOLK` and the requested section is not released.

A deadlock situation can arise if a process that controls a locked resource is made to pause by a request for the locked resource of another process. Therefore when `lockf()` or `fcntl()` are called, a check is first made for possible deadlocks before the process is suspended until a locked resource is released. If the waiting for a locked resource would cause a deadlock, the call fails and `errno` is set to `EDEADLK`.

Simultaneous locking with `lockf()` and `fcntl()` leads to undefined interactions.

The waiting for a resource is interrupted with a random signal. The `alarm()` system call can be used for the provision of a time lock for applications which require such a facility.

There is no difference in functionality between `lock()` and `lock64()` except that `lock64()` the size of the area to be locked is specified in an offset type `off64_t`.

If threads are used, then the function affects the process or a thread in the following manner:

File section is locked, lock calls from other threads that attempt to lock a file section that is already locked will result in the return of an error number or the calling thread will be blocked until the section is released. All locks for a process are deleted when the process is terminated.

#### Return value

0	if successful.
-1	if an error occurs. <code>errno</code> is set to indicate the error. Existing locks are not changed.

Errors	lockf() and lockf64() will fail if:
EBADF	<i>fdes</i> is not a valid open file descriptor, or <i>function</i> is F_LOCK or F_TLOCK and the file addressed via <i>fdes</i> is not opened for writing.
EACCES	<i>function</i> is F_TLOCK or F_TEST, and the section is already locked by another process.
EDEADLK	<i>function</i> is F_LOCK and a deadlock would occur.
EINTR	A signal was caught during execution of the function.
EAGAIN	<i>function</i> is F_LOCK or F_TLOCK and the file was generated with mmap().
ENOLCK	<i>function</i> is F_LOCK, F_TLOCK or F_ULOCK, and there is no longer enough storage space for additional entries in the lock table.
EINVAL	<i>fdes</i> points to a file type that cannot be locked in this implementation, or the contents of <i>function</i> are invalid, or the sum of <i>size</i> plus the current file offset is less than 0 or greater than the highest permissible file offset.
ECOMM	<i>fdes</i> is on a remote computer and the link to this computer is no longer active.
EOVERFLOW	The offset of the first byte or, when the size is not equal to 0, the last byte in the requested section cannot be represented correctly in an object of type <code>off_t</code> .

Notes Unexpected events can occur in processes that buffer in the address space of the user. The process can later read or write data that is or was locked. The standard I/O package is the most common cause of unexpected buffering. Instead of this, simpler functions should be used which work unbuffered, e.g. `open()`.

Because the `errno` variable will in future be set to `EAGAIN` and not to `EACCES` if a file section is already locked by another process, portable user programs must expect and check both values.

The `alarm()` function can be used to monitor a timeout which may occur.

See also `alarm()`, `chmod()`, `close()`, `creat()`, `fcntl()`, `mmap()`, `open()`, `read()`, `write()`, `unistd.h`.

## locs - stop regular expression matching in string

**Syntax**      `#include <regex.h>`  
                 `extern char *locs;`

**Description** See `regex()`.

**Notes**        This function will not be supported by the X/Open standard in the future.

New applications should use `fnmatch()`, `glob()`, `regcomp()` and `regexexec()`, which guarantee full internationalized regular expression functionality (see "Regular expressions" in the manual "POSIX Commands (BS2000/OSD)").

## log - natural logarithm function

**Syntax**      `#include <math.h>`  
                 `double log(double x);`

**Description** `log()` computes the natural logarithm of the positive floating-point number  $x$  to the base  $e$ .

**Return value**  
                 `ln(x)`            for a positive  $x$ .  
                 `-HUGE_VAL`        if  $x$  is less than or equal to 0.  
                 `errno` is set to indicate the error.

**Errors**        `log()` will fail if:  
                 `EDOM`                The value of  $x$  is negative.  
                 `ERANGE`         The value of  $x$  is 0.

**See also**      `exp()`, `log10()`, `pow()`, `sqrt()`, `math.h`.

## log10 - base 10 logarithm function

**Syntax**        `#include <math.h>`  
                 `double log10(double x);`

**Description** `log10()` computes the natural logarithm of the positive floating-point number  $x$  to the base 10.

**Return value**  
                 `lg(x)`                for a positive  $x$ .  
                 `-HUGE_VAL`        if  $x$  is less than or equal to 0.  
                 `errno` is set to indicate the error.

**Errors**        `log10()` will fail if:  
                 `EDOM`                The value of  $x$  is negative.  
                 `ERANGE`         The value of  $x$  is 0.

**See also**      `exp()`, `log()`, `pow()`, `sqrt()`, `math.h`.



## log1p - compute natural log

**Syntax**      `#include <math.h>`  
                `double log1p (double x);`

**Description** The `log1p()` function computes  $\log_e(1.0 + x)$ , where  $x$  must be greater than  $-1.0$ .

**Return value**  
                 $\ln(1.0 + x)$       if successful.  
                `-HUGE_VAL`      if  $x \leq -1.0$ .

**Errors**      `log1p()` will fail if:  
                `EDOM`              The value of  $x$  is less than  $-1.0$ .

**See also**    `log()`, `math.h`.

## logb - get exponent part of floating-point number

**Syntax**      `#include <math.h>`  
                `double logb(double x);`

**Description** `logb()` is identical to `ilogb()`, except that `logb()` does not return the exponent part of  $x$  as `int`, but as a double-precision signed floating-point number.

**Return value**  
                Exponent part of  $x$   
                                    if successful  
                `-HUGE_VAL`      for  $x = 0.0$ . `errno` is set to indicate the error.

**Errors**      `logb()` will fail if:  
                `EDOM`              The value of  $x$  is  $0.0$ .

**See also**    `ilogb()`, `math.h`.

## **\_longjmp, \_setjmp - non-local jump (without signal mask)**

**Syntax**      `#include <setjmp.h>`  
`void _longjmp(jmp_buf env, int val);`  
`int _setjmp(jmp_buf env);`

**Description** The `_longjmp()` and `_setjmp()` functions are identical to `longjmp()` and `setjmp()` respectively, except that they leave the signal mask unchanged.

If `_longjmp()` is called without *env* having been previously initialized by `_setjmp()`, or if the last `_setjmp()` call was in a function which has returned in the meantime, the behavior is undefined.

**Return value**  
See `longjmp()` and `setjmp()`.

**Notes** Errors can occur if `_longjmp()` is executed and the environment in which `_setjmp()` was executed no longer exists. The environment of the `_setjmp()` call no longer exists if the function containing the call terminates, or leaves the save area with the automatic variables. This error might not be detected, which leads to `_longjmp()` being executed. In this case the contents of the save area are unpredictable. This error can also cause the process to terminate. When the function returns, the result is undefined.

If a pointer to an area that was not generated by `setjmp()`, `_setjmp()` or `sigsetjmp()` is passed to `longjmp()`, `_longjmp()` or `siglongjmp()`, or if the area was changed by the user, the errors described above as well as addition problems can occur.

`_longjmp()` and `_setjmp()` are offered for reasons of compatibility. New applications should use `siglongjmp()` or `sigsetjmp()`.

**See also** `longjmp()`, `setjmp()`, `siglongjmp()`, `sigsetjmp()`, `setjmp.h`.

## longjmp - execute non-local jump

**Syntax**      `#include <setjmp.h>`  
`void longjmp(jmp_buf env, int val);`

**Description** `longjmp()` can only be used in combination with `setjmp()`. A call to `longjmp()` causes the program to branch to a position previously saved by `setjmp()`. In contrast to `goto` jumps, which are only permitted within the same function (i.e. locally), `longjmp()` allows jumps from any given function to any other active function (i.e. non-local jumps).

`setjmp()` saves the current process environment (address in the C runtime stack, program counter, register contents) in a variable of type `jmp_buf` (see `setjmp.h`). `longjmp()` restores the process environment saved by `setjmp()`, and the program is then continued with the statement immediately following the `setjmp()` call.

If no call to `setjmp()` precedes the `longjmp()` call, or if the function containing the `setjmp()` call has already completed execution, the results are undefined.

*env* is the array in which `setjmp()` stores its values (see `setjmp.h`).

*val* is an integer that is interpreted as the return value of the `setjmp` call when the process returns. If *val* is equal to 0, `setjmp()` returns a value of 1; 0 would imply that control was transferred "normally" to the position after the `setjmp()` call, i.e. that no branch was made with `longjmp()` (see also `setjmp()`).

All accessible objects will have the same values as when `longjmp()` was called, except for the values of "automatic" objects (i.e. objects of automatic storage duration), which are undefined under the following conditions:

- They are local to the function containing the corresponding `setjmp` call.
- They are not of type `volatile`.
- They are changed between the `setjmp` and `longjmp` calls.

Since `longjmp()` bypasses the usual function call and return mechanisms, `longjmp()` will execute correctly in contexts of interrupts, signals and any of their associated functions. However, if `longjmp()` is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

After `longjmp()` is completed, program execution continues as if the corresponding call to `setjmp()` had just returned the value specified by *val*. The `longjmp()` function cannot cause `setjmp()` to return 0; if *val* is 0, `setjmp()` returns 1.

The result of a call to this function is undefined if the `jmp_buf` structure was not initialized in the calling thread.

The `jmp_buf` structure must be initialized by `setjmp()`. This must be done in the same thread when threads are used.

**Notes** Non-local jumps are useful in the handling of interrupts (see `signal()`). For example, if error handling or interrupt handling is carried out in routines on a low level (i.e. when a number of previously called functions are still active), `longjmp()` and `setjmp()` can be used to circumvent normal processing of still active functions and immediately branch to a function on a higher level. A `longjmp` call from an interrupt or error handling routine flushes the entries in the runtime stack up to the position marked by `setjmp()`. In other words, functions that were active thus far on a lower level are now no longer active, and the program is continued on a higher level.

When program execution is resumed, the variables will have the same values as after a `goto` call, i.e. the global variables will have the values they had at the time of the `longjmp` call, and the register variables and other local variables will be undefined, i.e. should be checked and re-initialized, if required.

**See also** `setjmp()`, `sigaction()`, `siglongjmp()`, `sigsetjmp()`, `setjmp.h`.

## **lrand48 - generate pseudo-random numbers between 0 and $2^{31}$**

**Syntax** `#include <stdlib.h>`  
`long int lrand48 (void);`

**Description** See `drand48()`.

## **lrint, lrintf, lrintl - round to nearest integer value (long int)**

**Syntax**      `#include <math.h>`  
`long int lrint(double x);`  
`long int lrintf (float x);`  
`long int lrintl (long double x);`

**Description** The functions return the integer value (displayed as a number of type `long int`) nearest to *x*.

The returned value is rounded according to the currently set rounding mode of the computer. If the default mode is set to 'round-to-nearest' and the difference between *x* and the rounded result is exactly 0.5, the next even integer is returned.

If the currently set rounding mode rounds infinitely in the positive direction, `lrint()` is identical to `ceil()`. If the currently set rounding mode rounds infinitely in the negative direction, `lrint()` is identical to `floor()`.

In this version the rounding mode is set to round infinitely in the positive direction.

### **Return value**

**Integer**      value (type `long int`) nearest to *x*  
if successful.

**Undefined**    for overflow or underflow. `errno` is set to `ERANGE` to indicate an error.

**See also**      `abs()`, `ceil()`, `floor()`, `llrint()`, `llround()`, `lround()`, `rint()`, `round()`

## lround, lroundf, lroundl - round up to next integer value (long int)

**Syntax**      `#include <math.h>`  
                 `long int lround (double x);`  
                 `long int lroundf (float x);`  
                 `long int lroundl (long double x);`

**Description** The functions return the integer value (displayed as a number of type `long int`) nearest to *x*.

The returned value does not depend on the rounding mode currently set. If the difference between *x* and the rounded result is exactly 0.5, the next highest integer is returned.

### Return value

**Integer**      value (type `long int`) nearest to *x*  
                 if successful.

**Undefined**    for overflow or underflow. `errno` is set to `ERANGE` to indicate an error.

**See also**      `abs()`, `ceil()`, `floor()`, `llrint()`, `llround()`, `lrint()`, `rint()`, `round()`

## lsearch, lfind - linear search and update

**Syntax**

```
#include <search.h>

void *lsearch (const void *key, void *base, size_t *nelp,
              size_t width, int (*compar) (const void *, const void *));

void *lfind (const void *key, const void *base, size_t *nelp,
            size_t width, int (*compar)(const void *, const void *));
```

**Description** `lsearch()` is a linear search routine. It returns a pointer into a table indicating the position at which a specific value may be found. If the searched value is not present, it is added at the end of the table. *key* points to the entry to be sought in the table; *base* points to the first element in the table; *nelp* points to an integer containing the current number of elements in the table. The integer to which *nelp* points is incremented if the entry is added to the table. *width* is the size of an element in bytes. *compar* points to a comparison function which the user must supply (`strcmp()`, for example). It is called with two arguments that point to the elements being compared. The function must return 0 if the elements are equal and non-zero otherwise.

`lfind()` has the same effect as `lsearch()` except that if the entry is not found, it is not added to the table. Instead, a null pointer is returned.

**Return value**

<i>*key</i>	<code>lfind()</code> : if successful. <code>lsearch()</code> : if successful, and also for a newly added element.
Null pointer	<code>lfind()</code> : if an error occurs.

**Notes** The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Undefined results can occur if there is not enough room in the table to add a new item.

### *Extension*

The pointers to the key and the element at the base of the table may be pointers of any type.

The returned value should be convertible to the type pointer to element. □

**See also** `bsearch()`, `hsearch()`, `tsearch()`, `search.h`.

## lseek - move read/write file offset

**Name**        **lseek, lseek64**

**Syntax**

*Optional*

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
off_t lseek (int fildes, off_t offset, int whence);
```

```
off64_t lseek64 (int fildes, off64_t offset, int whence);
```

**Description** If POSIX files are executed, the behavior of this function conforms to the XPG standard as described below:

`lseek()` sets the file offset (i.e. the file position indicator) for the file with the file descriptor *fildes* as follows:

If *whence* is `SEEK_SET`, the file offset is set to *offset* bytes.

If *whence* is `SEEK_CUR`, the file offset is set to its current location plus *offset*.

If *whence* is `SEEK_END`, the file offset is set to the size of the file plus *offset*.

The symbolic constants `SEEK_SET`, `SEEK_CUR` and `SEEK_END` are defined in the header `unistd.h`.

The `lseek()` function has no effect when applied on a file that is incapable of seeking.

`lseek()` allows the file offset to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return bytes with the value 0 until data is actually written into the gap.

`lseek()` will not, by itself, extend the size of a file.

There is no difference in functionality between `lseek()` and `lseek64()` except that `lseek64()` uses the offset type `off64_t`.

*BS2000*

The following must be noted when executing BS2000 files:

`lseek()` sets the file position indicator for the file with file descriptor *fildes* according to the specifications in *offset* and *whence*. This allows a file to be processed non-sequentially. The return value of `lseek()` is the current position in the file.

Text files (SAM, ISAM) can be positioned absolutely to the beginning or end of the file as well as to any position previously marked with `tell()`.



Binary files (PAM, INCORE) can be positioned absolutely (see above) or relatively, i.e. relative to beginning of file, end of file, or current position (by a desired number of bytes). SAM are always processed as text files with elementary functions.

The significance, combination options, and effects of the *offset* and *whence* parameters differ for text and binary files and are therefore discussed individually below.

**Text files (SAM, ISAM)**

Possible values:

*offset*            0L or value determined by a previous `tell/lseek` call.  
*whence*            SEEK\_SET (beginning of file)  
                       SEEK\_CUR (current position)  
                       SEEK\_END (end of file)

Meaningful combinations and their effects:

<i>offset</i>	<i>whence</i>	<b>Effect</b>
<code>tell/lseek</code> value	SEEK_SET	Position to the location marked by <code>tell()</code> or <code>lseek()</code> .
0L	SEEK_SET	Position to the beginning of the file.
0L	SEEK_CUR	Check current position without moving.
0L	SEEK_END	Position to the end of the file.

**Binary files (PAM, INCORE)**

Possible values:

*offset*            Number of bytes by which the current file position indicator is to be shifted. This number may be

- positive: position forwards toward the end of the file
- negative: position backwards toward the beginning of the file
- 0L: absolute positioning to the beginning or end of the file

*whence*            For absolute positioning to the beginning or end of the file, the location at which the file position indicator is to be set. For relative positioning, the reference point from which the file position indicator is to be moved by *offset* bytes:

SEEK\_SET (beginning of file)  
 SEEK\_CUR (current position)  
 SEEK\_END (end of file)

Meaningful combinations and their effects:

<i>offset</i>	<i>whence</i>	<b>Effect</b>
0L	SEEK_SET	Position to the beginning of the file.
0L	SEEK_CUR	Check current position without moving.
0L	SEEK_END	Position to the end of the file.
positive number	SEEK_SET SEEK_CUR SEEK_END	Forward positioning from beginning of file, from current position, from end of file (beyond the end of file).
negative number	SEEK_CUR SEEK_END	Backward positioning from current position, from end of file.
tell/lseek value	SEEK_SET	Position to the location marked by a tell() or lseek call.

□

Return value

New value of the file position indicator, measured in bytes from the beginning of the file, if successful.

(off\_t) -1 if an error occurs; `errno` is set to indicate the error. The value of the file position indicator remains the same.

*BS2000*

New value of the file position indicator, measured in bytes from the beginning of the file; for binary files,  
if successful

Absolute position

in text files,  
if successful.

-1 if an error occurs.

Errors

`lseek()` and `lseek64()` will fail if:

EBADF *files* is not an open file descriptor.

EINVAL *whence* is not a proper value, or the resulting file offset would be invalid.

ESPIPE *files* is associated with a pipe or FIFO.

EOVERFLOW The resulting file offset cannot be represented correctly in the structure pointed to by *offset*.

Notes The program environment determines whether a BS2000 or POSIX file is created.

### BS2000

The calls `lseek(stream, 0L, SEEK_CUR)` and `tell(stream)` are equivalent, i.e. they both seek the current position in the file without moving it.

If new records are written to a text file (opened for creation or in append mode) and an `lseek` call is issued, any data that may still be in the internal C buffer is first written to the file and terminated with a newline character (`\n`).

Exception for ANSI functionality:

If the data of an ISAM file in the buffer does not end in a newline character, `lseek()` does not insert a change of line (or record). In other words, the data is not automatically terminated with a newline character when it is written from the buffer. Subsequent data extends the record in the file. Consequently, when an ISAM file is read, only the newline characters that were explicitly written by the program are read in.

If a binary file is positioned past the end of file, a gap appears between the last physically stored data and the newly written data. Reading from this gap returns binary zeros.

It is not possible to position to system files (SYSDTA, SYSLST, SYSOUT).

Since information on the file position is stored in a field that is 4 bytes long, the following restrictions apply to the size of SAM and ISAM files when processing them with `tell()/lseek()`:

### SAM file

Record length	≤ 2048 bytes
Number of records/block	≤ 256
Number of blocks	≤ 2048

### ISAM file

Record length	≤ 32 KB
Number of records	≤ 32 K

See also `fseek()`, `ftell()`, `open()`, `tell()`, `sys/types.h`, `unistd.h`.

## Istat - query file status

**Name** Istat, lstat64

**Syntax** #include <sys/stat.h>  
#include <sys/types.h>

```
int lstat (const char *path, struct stat *buf);
int lstat64 (const char *path, struct stat64 *buf);
```

**Description** Like `stat()`, `lstat()` returns file attributes, except that if `path` points to a symbolic link, `lstat()` outputs information on the link, while `stat()` outputs information on the file to which the link refers.

`buf` is a pointer to a `stat` structure to which the information on the specified file is written.

There is no difference in functionality between `lstat()` and `lstat64()` except that `lstat64()` returns the file status in a `stat64` structure.

The `stat` structure contains the following elements:

```
mode_t    st_mode;    /* File mode (see mknod()) */
ino_t     st_ino;     /* Inode number */
dev_t     st_dev;     /* Device ID which contains a directory entry for
                        this file */
dev_t     st_rdev;    /* Device ID, defined for character-special or
                        block-special files only */
nlink_t   st_nlink;   /* Number of links */
uid_t     st_uid;     /* User ID of the file owner */
gid_t     st_gid;     /* Group ID of the file owner */
off_t     st_size;    /* File size in bytes */
time_t    st_atime;   /* Time of the last access */
time_t    st_mtime;   /* Time of the last data modification */
time_t    st_ctime;   /* Time of the last change of file status
                        The time is measured in seconds as of
                        January 1, 1970, 00:00:00 */
long      st_blksize; /* Preferred I/O block size */
blkcnt_t  st_blocks;  /* Number of assigned st_blksize blocks */
```

The `stat64` structure is defined like `stat` except for the following components:

```
ino64_t   st_ino
off64_t   st_size and
blkcnt64_t st_blocks
```

In addition to the modes described in `mknod()`, `st_mode` can also be `S_IFLNK` if the file is a symbolic link.

The `st_size` component contains the length of the pathname in the symbolic link. Trailing zeros are not counted. The contents of all remaining components of the `stat` structure are undefined.

## Return value

- 0 if successful.
- 1 if an error occurs. `errno` is set to indicate the error.

## Errors

`lstat()` and `lstat64()` will fail if:

- `EACCES` Search permission is denied for a component of the path.
- `EIO` An I/O error occurred when reading from or writing to the file system.
- `ELOOP` Too many symbolic links were encountered in resolving *path*.
- `ENAMETOOLONG` The length of the pathname exceeds `{PATH_MAX}`, or the length of a component of the pathname exceeds `{NAME_MAX}`.
- `ENOTDIR` A component of the pathname prefix is not a directory.
- `ENOENT` A component of the pathname does not exist, or *path* points to an empty string.
- `EOVERFLOW` A component is too large to be stored in the structure pointed to by *buf*.
- BS2000*
- `EINVAL` An attempt was made to access a BS2000 file.
- `ENAMETOOLONG` The resolving of symbolic links in the pathname leads to an interim result whose length exceeds `{PATH_MAX}`.
- `EOVERFLOW` A component is too big to be stored in the structure pointed to by *buf*.
- `EFAULT` *buf* or *path* point to an invalid address.
- `EINTR` A signal was caught during the `stat()` or `lstat()` system call.

## major - get major component of device number *(extension)*

**Syntax**      `#include <sys/types.h>`  
                 `#include <sys/mkdev.h>`  
  
                 `major_t major(dev_t device);`

**Description** `major()` returns the major component of the device number for a named *device*.

### Return value

Formatted device number  
                         if successful.  
  
NODEV                if an error occurs. `errno` is set to indicate the error.

**Errors**        `major()` will fail if:

EINVAL              The *device* argument is NODEV,  
                         or the major component of *device* is too large.

**See also**      `makedev()`, `minor()`, `mknod()`, `stat()`.

## makecontext, swapcontext - set up user context

**Syntax**      `#include <ucontext.h>`

```
void makecontext (ucontext_t *ucp, (void *func) (), int argc,...);  
int swapcontext (ucontext_t *oucp, const ucontext_t *ucp);
```

**Description** These functions serve to implement a change of context between several control flows within a user process.

`makecontext()` changes the context specified by *ucp* which was initialized via `getcontext()`. If this context is activated with `swapcontext()` or `setcontext()` (see `getcontext()`), the program execution is continued with the call of the function *func*. The arguments which follow *argc* are passed to `makecontext()`. The integer value of *argc* must correspond to the number of arguments that follow *argc*. Otherwise, the behavior is undefined.

Before `makecontext()` is called, the context to be modified should be assigned a stack. The structure element `uc_link` defines the context that is activated when the context modified by `makecontext()` returns.

`swapcontext()` saves the current context in the context structure pointed to by *oucp*, and sets the context to the context structure pointed to by *ucp*.

**Return value**

0                    after successful execution of `swapcontext()`.  
-1                   if an error occurs. `errno` is set to indicate the type of the error.

**Errors**            These functions will fail if:

ENOMEM             *ucp* no longer has enough space in the stack to perform the operation.

**See also**          `exit()`, `getcontext()`, `sigaction()`, `sigprocmask()`, `ucontext.h`.

## makedev - get formatted device number *(extension)*

```
#include <sys/types.h>
#include <sys/mkdev.h>

dev_t makedev(major_t maj, minor_t min);
```

**Description** `makedev()` returns a formatted device number. *maj* is the major number of the device, and *min* is its minor number. `makedev()` can be used to generate a device number for `mknod()`.

### Return value

Formatted device number  
if successful.

NODEV if an error occurs. `errno` is set to indicate the error.

**Errors** `makedev()` will fail if:

EINVAL One or both of the *maj* and *min* arguments is too large, or the device number created from *maj* and *min* is NODEV.

**See also** `major()`, `minor()`, `mknod()`, `stat()`.



## malloc - memory allocator

Syntax `#include <stdlib.h>`

```
void *malloc(size_t size);
```

Description `malloc()` allocates contiguous memory of *size* bytes at execution time.

If *size* = 0, `malloc()` returns a null pointer.

`malloc()` is part of a C-specific memory management package that internally administers memory areas which are requested and subsequently freed. As far as possible, all new requests are first satisfied from the areas that are already being managed and only then from the operating system.

Return value

Pointer to the new memory area if *size* was not 0 and `malloc()` was able to allocate new memory. This pointer may be used for any data type.

Null pointer if `malloc()` was unable to provide the memory, e.g. because the available memory space was insufficient for the request or because an error occurred.  
`errno` is set to indicate the error.

Errors `malloc()` will fail if:

ENOMEM The available memory space is insufficient.

Notes The new data area begins on a double-word boundary.

The actual length of the data area is equal to the requested length *size* + 8 bytes for internal administration data. If required, this amount is rounded up to the next power of 2.

The `sizeof` operator should be used to ensure that sufficient space for a variable is requested.

If the length of the allocated memory area is exceeded when writing, critical errors may occur in the working memory.

`malloc()` is interrupt-protected as of this version, i.e. the function can now also be used in signal handling and contingency routines.

See also `calloc()`, `free()`, `realloc()`, `stdlib.h`.

## mblen - get number of bytes in multi-byte character

**Syntax**      `#include <stdlib.h>`  
`int mblen(const char *s, size_t n);`

**Description** `mblen()` returns the number of bytes of a multi-byte character to which *s* points. A maximum of *n* bytes in *s* are evaluated.

No characters consisting of multiple bytes are implemented in this version. Multi-byte characters always have a length of 1 (`MB_CUR_MAX = 1`).

**Return value**

-1	if <i>n</i> = 0.
0	if <i>s</i> is a null pointer or points to a null byte.
1	in all other cases.

**See also**    `mbstowcs()`, `mbtowc()`, `wcstombs()`, `wctomb()`, `stdlib.h`.

## mbrlen - get number of bytes in multi-byte character

**Syntax**      `#include <wchar.h>`  
`size_t mbrlen(const char *s, size_t n, mbstate_t *ps);`

**Description** `mbrlen()` returns the number of bytes required to complete a multi-byte character starting at the position *\*s*. A maximum of *n* bytes are evaluated.

`mbrlen()` corresponds to the call  
`mbrtowc(NULL, s, n, ps!= NULL ? ps: internal)`  
where *internal* is the `mbstate_t` object for the function.

See `mbrtowc()` for a detailed description.

## mbrtowc - complete and convert multi-byte string to wide-character string

**Syntax**      `#include <wchar.h>`

```
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

**Description** If *s* is not a null pointer, `mbrtowc()` determines how many bytes (starting at the position pointed to by *s*) are required to complete the next multi-byte character. Any Shift sequences are also taken into account. A maximum of the next *n* bytes are tested. If `mbrtowc()` can complete the multi-byte character, the corresponding wide character is determined and stored in *pwc* as long as *pwc* is not a null pointer. If the corresponding wide character is the null character, the final state corresponds to the “initial conversion” state.

If *s* is a null pointer, `mbrtowc()` corresponds to the call `mbrtowc(NULL, "", 1, ps)`. In this case the parameters *pwc* and *n* are ignored.

### Return value

Depending on the current conversion state, `mbrtowc()` returns the value of the first condition of the following conditions that is met:

0                    if the next (maximum of *n*) bytes result in a valid multi-byte character that corresponds to the wide character “null”.

Number of bytes required to complete the multi-byte character  
if the next (maximum of *n*) bytes result in a valid multi-byte character. The wide character corresponding to this multi-byte character is stored.

(`size_t`)-2        if the next *n* bytes result in an incomplete, but potentially valid multi-byte character. No value is stored.

(`size_t`)-1        if a coding error occurs, i.e. if the next (maximum of *n*) bytes do not result in a complete and valid multi-byte character. No value is stored and the value of the `EILSEQ` macro is written in `errno`. The conversion status is undefined.

**Notes**            In this version of the C runtime system, only 1-byte characters are supported as wide characters.

**See also**        `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

## mbsinit - test for “initial conversion” state

Syntax `#include <wchar.h>`

```
int mbsinit(const mbstate_t *ps);
```

Description If *ps* is not a null pointer, `mbsinit()` tests if the `mbstate_t` object pointed to by *ps* describes an “initial conversion” state.

Return value

Value $\neq$ 0	if <i>ps</i> is a null pointer or points to an object that describes an “initial conversion” state.
0	otherwise.

## mbsrtowcs - convert multi-byte string to wide-character string

**Syntax**      `#include <wchar.h>`  
`size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);`

**Description** `mbsrtowcs()` converts a sequence of multi-byte characters in the array indirectly pointed to by *src* to wide characters. `mbsrtowcs()` starts the conversion with the conversion state described in *\*ps*. The converted characters are written to the array pointed to by *dst* as long as *dst* is not a null pointer. Every character is converted as if the `mbtowc()` was called.

The conversion terminates when a terminating null character is encountered. The null character is also converted and written into the array.

The conversion is terminated abnormally if

- a sequence of bytes is found that does not represent a valid multi-byte character or
- *dst* is not a null pointer and *len* characters were written into the array pointed to by *dst*.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned one of the following two values:

- a null pointer if the conversion terminated when it reached a null character
- the address directly after the last multi-byte character converted

If *dst* is not a null pointer and the conversion terminated when it reached a null character, then the final state is the same as the “initial conversion” state.

**Return value**  
`(size_t)-1` if a conversion error occurred, i.e. a sequence of bytes that does not represent a valid multi-byte character was found. The value of the `EILSEQ` macro is written in `errno`. The conversion status is undefined.

Otherwise the number of successfully converted multi-byte characters.  
The terminating null character (if present) is not counted.

**See also** `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

## mbstowcs - convert multi-byte string to wide-character string

Syntax `#include <stdlib.h>`

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Description `mbstowcs()` converts a sequence of multi-byte characters in the string *s* to the appropriate `wchar_t` values and stores a maximum of *n* `wchar_t` values in the area *pwcs*. `mbstowcs()` converts until either *n* values have been converted or a null value is encountered (null is converted into the `wchar_t` value 0).

If *pwcs* is a null pointer, `mbstowcs()` returns the length required to convert the entire string, regardless of the value *n*, but does not save any values.

If an invalid character is present in the string to be converted, `mbstowcs()` returns the value `(size_t)-1`.

The `wchar_t` values (type `long`) which are stored by `mbstowcs()` in the *pwcs* area correspond to the values of the individual bytes in the string *s*.

Return value

Number of `wchar_t` values stored in *pwcs* (excluding the terminating null byte)  
if *pwcs* is not a null pointer.

If the return value corresponds to the value *n*, the result area *pwcs* is not terminated with the null byte.

Length required to convert the entire string,  
if *pwcs* is a null pointer. No values are stored.

`(size_t)-1` if an error occurs.

Notes The behavior is undefined if memory areas overlap.

No characters consisting of multiple bytes are implemented in this version. Multi-byte characters always have a length of 1 byte, and `wchar_t` values are always of type `long`.

See also `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`, `stdlib.h`.

## mbtowc - convert multi-byte character to wide character

**Syntax**      `#include <stdlib.h>`

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

**Description** `mbtowc()` converts a multi-byte character in *s* to the corresponding `wchar_t` value and stores this value in the area *pwc*. A maximum of *n* bytes in *s* are evaluated.

The `wchar_t` value (type `long`) stored by `mbtowc()` in the area *pwc* corresponds to the value of the byte in *s*.

No assignment takes place if:

*pwc* or *s* is a null pointer, or

*n* = 0.

**Return value**

-1              if *n* = 0.

0                if *s* is a null pointer or points to a null byte.

1                in all other cases.

**Notes**        No characters consisting of multiple bytes are implemented in this version. Multi-byte characters always have a length of 1 byte, and `wchar_t` values are always of type `long`.

**See also**     `mblen()`, `mbstowcs()`, `wcstombs()`, `wctomb()`, `stdlib.h`.

## memalloc - memory allocator *(BS2000)*

**Syntax**        `#include <stdlib.h>`  
                 `void *memalloc(size_t num);`

**Description** `memalloc()` allocates contiguous memory of *num* bytes at execution time.

`memalloc()` passes the request for memory directly to the appropriate operating system call. This function is particularly suitable for memory areas with a size of more than 2 KB (see also `memfree()`).

### Return value

**Pointer**        to the new memory area  
                 if `memalloc()` was able to allocate new memory. This pointer may be used for any data type.

**Null pointer**   if `memalloc()` was unable to provide the memory, e.g. because the available memory space was insufficient for the request.

### Notes

The new memory area begins on a double-word boundary.

The requested length *num* is rounded up to the next multiple of 2 KB.

If the length of the allocated memory area is exceeded when writing, a serious disruption in working memory may occur.

The memory area requested with `memalloc()` can be released by using `memfree()`.

**See also**        `memfree()`.



## memccpy - copy bytes in memory

**Syntax**      `#include <string.h>`

```
void *memccpy(void *s1, const void *s2, int c, size_t n);
```

**Description** `memccpy()` copies bytes from memory area `s2` into `s1` until

- either `c` is copied for the first time (where `c` is converted to an unsigned char),
- or `n` bytes have been copied.

If the copy process affects objects which overlap, the behavior is undefined.

**Return value**

**Pointer**            to the byte after the copy of `c` in `s1`  
if successful.

**Null pointer**      if `c` was not found in the first `n` characters of `s2`.

**Notes**            `memccpy()` does not check whether there will be an overflow in the memory area to which it copies.

**See also**        `memchr()`, `memcmp()`, `memcpy()`, `memset()`, `string.h`.

## memchr - find byte in memory

**Syntax**      `#include <string.h>`

```
void *memchr(const void *s, int c, size_t n);
```

**Description** `memchr()` locates the first occurrence of `c` in the initial `n` bytes of the memory area pointed to by `s`.

`s` is the pointer to the memory area in which byte `c` is to be found.

`c` is the EBCDIC value of the byte to be found.

`n` is the integer value that specifies the number of bytes to be found in `s`.

**Return value**

Pointer            to the position of `c` in area `s`  
                    if successful.

Null pointer      if `c` does not occur in the specified area.

**Notes**            The function is suitable for processing character arrays, which, in contrast to character strings, need not be terminated by the null byte (`\0`).

**See also**        `memcmp()`, `memcpy()`, `memset()`, `string.h`.

## memcmp - compare bytes in memory

**Syntax**      `#include <string.h>`

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

**Description** `memcmp()` compares the contents of the first  $n$  bytes of the memory areas to which  $s1$  and  $s2$  point.

$s1$  and  $s2$  are pointers to the memory areas to be compared.

$n$  is an integer value that specifies the number of bytes to be compared.

**Return value**

Integer value, which may be:

< 0            In the first  $n$  bytes, the contents of  $s1$  are lexically smaller than the contents of  $s2$ .

0              In the first  $n$  bytes, the contents of  $s1$  and  $s2$  are of equal lexical size (i.e. identical).

> 0            In the first  $n$  bytes, the contents of  $s1$  are lexically larger than the contents of  $s2$ .

**Notes**        This function is suitable for processing character arrays, which, in contrast to character strings, need not be terminated by the null byte (`\0`).

**See also**     `memchr()`, `memcpy()`, `memset()`, `string.h`.

## memcpy - copy bytes in memory

**Syntax**      `#include <string.h>`

```
void *memcpy(void *s1, const void *s2, size_t n);
```

**Description** `memcpy()` copies the first *n* bytes of the memory area to which *s2* points into the memory area pointed to by *s1*.

*s1* is a pointer to the memory area to which the bytes are to be copied.

*s2* is a pointer to the memory area from which the first *n* bytes are to be copied.

*n* is an integer value that specifies the number of bytes in *s2* to be copied.

**Return value**

Pointer            to the memory area *s1*  
if successful.

**Notes**        This function is suitable for processing character arrays, which, in contrast to character strings, need not be terminated by the null byte (`\0`).

`memcpy()` does not check whether data in result area *s1* is in danger of being overwritten.

The behavior is undefined if memory areas overlap.

**See also**     `memccpy()`, `memchr()`, `memcmp()`, `memset()`, `string.h`.

## memfree - free memory area *(BS2000)*

**Syntax**      `#include <stdlib.h>`  
`void memfree(const void *ptr, size_t num);`

**Description** `memfree()` releases *num* bytes of the memory area to which *ptr* points.  
`memfree()` passes on the release request directly to the appropriate operating system call.  
`memfree()` can only be used in conjunction with `mema11oc()`. Both functions are mainly suitable for memory areas with a size of more than 2 KB.

*ptr* is a pointer to the memory area to be freed.

*ptr* must be the result of a preceding `mema11oc()` call.

*num* is an integer value that specifies the size of the memory area in bytes.

**Notes**      `memfree()` can only be used to free a memory area requested by `mema11oc()`.

The values passed to `memfree()` must match those of the corresponding `mema11oc()` call.  
Random values will lead to critical errors in the working memory!

**See also**    `mema11oc()`.

## memmove - copy bytes in memory with overlapping areas

Syntax `#include <string.h>`

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

Description `memmove()` copies the first  $n$  bytes of the memory area to which  $s2$  points into the memory area pointed to by  $s1$ .

`memmove()` first copies the  $n$  bytes to a temporary field that does not overlap memory areas  $s1$  and  $s2$  and then copies the bytes from that field to the memory area  $s1$ .

$s1$  is a pointer to the memory area to which the bytes are to be copied.

$s2$  is a pointer to the memory area from which the first  $n$  bytes are to be copied.

$n$  is an integer value that specifies the number of bytes in  $s2$  to be copied.

Return value

Pointer to memory area  $s1$   
if successful.

Notes This function is suitable for processing character arrays, which, in contrast to strings, need not be terminated with the null byte (`\0`).

`memmove()` also works with memory areas that overlap; `memcpy()`, by contrast, does not.

See also `memcpy()`, `string.h`.

## memset - initialize memory area

**Syntax**        `#include <string.h>`  
`void *memset(void *s, int c, size_t n);`

**Description** `memset()` copies the value of character *c* into each of the first *n* bytes of the memory area to which *s* points.

*s* is a pointer to the memory area to be initialized with character *c*.

*c* is the EBCDIC value of the character to be copied.

*n* is an integer value that specifies the number of bytes in *s* to be initialized with character *c*.

**Return value**  
Pointer            to the memory area *s*  
                    if successful.

**Notes**            This function is suitable for processing character arrays, which, in contrast to strings, need not be terminated by the null byte (`\0`).

`memset()` does not check whether data in result area *s* is in danger of being overwritten.

**See also**        `memcpy()`, `memchr()`, `memcmp()`, `memcpy()`, `string.h`.

## minor - get minor component of device number *(extension)*

**Syntax**

```
#include <sys/types.h>
#include <sys/mkdev.h>

minor_t minor(dev_t device);
```

**Description** `minor()` returns the minor component of the device number for a named *device*.

**Return value**

Formatted device number  
if successful.

NODEV if an error occurs. `errno` is set to indicate the error.

**Errors** `minor()` will fail if:

EINVAL The *device* argument is NODEV.

**See also** `makedev()`, `major()`, `mknod()`, `stat()`.



## mkdir - make directory

Syntax `#include <sys/stat.h>`

*Optional*

`#include <sys/types.h>`

`int mkdir(const char *path, mode_t mode);`

Description `mkdir()` creates a new directory with the name *path*. The mode of the new directory is initialized from *mode* (see `chmod()` for values of *mode*). The file permission bits of the *mode* argument are modified by the file creation mask of the process (see `umask()`).

The directory's user ID is set to the process' effective user ID. The directory's group ID is set to the process' effective group ID, or if the `S_ISGID` bit is set in the parent directory, then the group ID of the directory is inherited from the parent. The `S_ISGID` bit of the new directory is inherited from the parent directory.

If *path* is a symbolic link, it is not used.

The newly created directory will be an empty directory, except for the entries for itself and its parent directory.

Upon successful completion, `mkdir()` will mark for update the `st_atime`, `st_ctime` and `st_mtime` fields of the directory. The `st_ctime` and `st_mtime` fields of the directory that contains the new entry are also marked for update.

Return value

0 if successful.

-1 if an error occurs. No directory is created and `errno` is set to indicate the error.

Errors

`mkdir()` will fail if:

`EACCES` Either there is no search permission for a component of the path prefix, or there is no write permission for the parent directory of the new directory.

`EEXIST` The specified file already exists.

*Extension*

`EFAULT` *path* points outside the allocated address space of the process.

`EIO` An I/O error occurred when accessing the file system.

`ELOOP` Too many symbolic links were encountered in resolving *path*.

`EMLINK` The maximum number of links `{LINK_MAX}` in the parent directory was exceeded.

ENAMETOOLONG

The length of the *path* argument exceeds `{PATH_MAX}` or a component of *path* is longer than `{NAME_MAX}`.

ENOENT

A component of the path does not exist or *path* points to an empty string.

*Extension*

ENOLINK

*path* points to a remote computer and the link to that computer is no longer active. □

ENOSPC

No free space is available on the device containing the directory.

ENOTDIR

A component of the path is not a directory.

EROFS

The specified file resides on a read-only file system.

**Notes**      `mkdir()` is executed only for POSIX files

**See also**    `chmod()`, `mknod()`, `umask()`, `stat()`, `sys/stat.h`, `sys/types.h`.

## mkfifo - create FIFO file

Syntax `#include <sys/stat.h>`

*Optional*

`#include <sys/types.h> □`

`int mkfifo(const char *path, mode_t mode);`

**Description** `mkfifo()` creates a new FIFO special file (FIFO for short) with the pathname *path*. The access mode of the new FIFO is initialized from *mode*. The file permission bits of the *mode* argument are modified by the process' file creation mask (see `umask()`).

The user ID of the FIFO is set to the effective user ID of the process, and the group ID of the FIFO is set to the effective group ID of the process, unless the `S_ISGID` bit is set in the parent directory, in which case the group ID of the FIFO is inherited from the parent directory.

Upon successful completion, `mkfifo()` will mark for update the `st_atime`, `st_ctime` and `st_mtime` fields of the file. The `st_ctime` and `st_mtime` fields of the directory that contains the new entry are also updated (see `sys/stat.h`).

**Return value**

0 if successful.

-1 if no FIFO was created. `errno` is set to indicate the error.

**Errors**

`mkfifo()` will fail if:

`EACCES` Either no search permission exists for a component of the path, or no write permission exists for the parent directory of the new FIFO file.

`EEXIST` The specified file already exists.

`ELOOP` Too many symbolic links were encountered during in resolving *path*.

*Extension*

`EINVAL` An attempt was made to access a BS2000 file. □

`ENAMETOOLONG`

The length of the *path* argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}` and `{_POSIX_NO_TRUNC}` is set.

`ENOENT` A component of the path prefix does not exist or *path* points to an empty string.

`ENOSPC` The directory that would contain the new file cannot be extended or the file system is out of file-allocation resources.

ENOTDIR      A component of the path prefix is not a directory.

EROFS        The specified file resides on a read-only file system.

Notes        Bits other than the file permission bits in *mode* are ignored.

*path* can only be a POSIX file.

See also     `umask()`, `sys/stat.h`, `sys/types.h`.

## mknod - make directory, special file, or text file

Syntax `#include <sys/stat.h>`

```
int mknod(const char *path, mode_t mode, dev_t dev);
```

`mknod()` creates a new file with the pathname pointed to by *path*. The file type and permissions of the new file are initialized from *mode*.

If *path* is a symbolic link it is not traced.

Description The file type for *path* is incorporated in the *mode* argument via a bit-wise OR. The file type must be one of the following symbolic constants:

<code>S_IFIFO</code>	FIFO special file
<code>S_IFCHR</code>	Character-special file (not portable)
<code>S_IFDIR</code>	Directory (not portable)
<code>S_IFBLK</code>	Block-special file (not portable)
<code>S_IFPOSIXBS2</code>	File in the POSIX file system (not portable)
<code>S_IFREG</code>	Regular file (not portable)

`mknod()` can only be used portably in accordance with the X/Open standard if a FIFO file is generated. If the file type is not `S_IFIFO`, or *dev* does not have the value 0, the behavior of `mknod()` is undefined.

The access permissions of the file are also incorporated in the *mode* argument via a bit-wise OR. They can be defined by any combination of the following symbolic constants:

Symbolic name	Bit pattern	Meaning
<code>S_ISUID</code>	04000	Set user ID on execution
<code>S_ISGID</code>	020#0	Set group ID on execution
<code>S_IRWXU</code>	00700	Read, write or execute (search if a directory is involved) by owner
<code>S_IRUSR</code>	00400	Read by owner
<code>S_IWUSR</code>	00200	Write by owner
<code>S_IXUSR</code>	00100	Execute by owner (search if a directory is involved)
<code>S_IRWXG</code>	00070	Read, write or execute (search) by group
<code>S_IRGRP</code>	00040	Read by group
<code>S_IWGRP</code>	00020	Write by group
<code>S_IXGRP</code>	00010	Execute (search) by group
<code>S_IRWXO</code>	00007	Read, write or execute (search) by others
<code>S_IROTH</code>	00004	Read by others
<code>S_IWOTH</code>	00002	Write by others
<code>S_IXOTH</code>	00001	Execute by others
<code>S_ISVTX</code>	01000	For directories: unrestricted delete permission

The user ID of the file is set to the effective user ID of the process, and the group ID of the file is set to the effective group ID of the process, unless the `S_ISGID` bit is set in the parent directory, in which case the group ID of the file is inherited from the parent directory.

The access permission bits of *mode* are modified by the file mode creation mask of the process: all bits which are set in the file mode creation mask are set to 0 by `mknod()`.

If *mode* indicates a block- or character-special file, *dev* is the configuration-dependent specification of that file; if *mode* does not indicate a block- or character-special file, *dev* is ignored (see `mkdev()`).

For non-FIFO file types, `mknod()` can only be invoked with appropriate privileges (`uid = 0`).

#### Return value

- 0 if successful.
- 1 if an error occurs; `errno` is set to indicate the error. In the event of an error, no new file is created.

#### Errors

`mknod()` will fail if:

- `EACCES` Either there is no search permission for a component of the path, or there is no write permission for the parent directory of the new file.
- `EEXIST` The specified file already exists.
- `EINTR` A signal was caught during the `mknod()` system call.
- `EINVAL` An argument is invalid.
- `EIO` An I/O error occurred during access to the file system.
- `ELOOP` Too many symbolic links were encountered in resolving *path*.
- `ENAMETOOLONG` The length of the *path* argument exceeds `{PATH_MAX}`, or a component of *path* is longer than `{NAME_MAX}`.  
The resolving of a symbolic link led to an interim result whose length exceeds `{PATH_MAX}`.
- `ENOENT` A component of the path prefix does not exist or *path* is an empty string.
- `ENOLINK` *path* refers to a remote computer and the link to this computer is no longer active.
- `ENOSPC` The directory in which the file is to be created cannot be extended, or no memory is available.
- `ENOTDIR` A component of the path prefix is not a directory.

EPERM	The effective user ID is not that of the system administrator and the file type is not FIFO.
EROFS	The directory in which the file is to be created is located on a read-only file system.

**Notes** `mknod()` is executed only for POSIX files.  
If `mknod()` creates a special file in a remote directory with RFS (remote file sharing), the device class and device number will be interpreted by the server.

For reasons of portability to implementations which comply with earlier versions of the X/Open standard, the `mkfifo()` function is recommended for creating FIFO files.

**See also** `chmod()`, `creat()`, `exec()`, `mkdir()`, `mkfifo()`, `open()`, `stat()`, `umask()`, `sys/stat.h`, `sys/types.h`.

## mkstemp - make unique temporary file name

**Syntax**      `#include <stdlib.h>`  
`int mkstemp(char *template);`

**Description** `mkstemp()` creates a unique file name, normally in a temporary file system, and returns an open file descriptor for this file. The file is opened for reading and writing. In this way, `mkstemp()` prevents a possible race between an existence check and the opening of the file.

The string to which *template* points should contain a file name followed by six Xs. `mkstemp()` replaces these Xs with a letter and the current process ID to create a unique file name. The letter is chosen so that the new file name will be unique, i.e. will not match any existing file name.

**Return value**  
open file descriptor  
                  if successful  
  
-1                if no suitable file could be created.

**Notes**        It is possible that the letters may run out.

`mkstemp()` does not check whether the file name component in *template* exceeds the maximum permitted length for file names.

For reasons of portability to implementations that comply with earlier versions of the X/Open standard, the `tmpfile()` function is recommended for creating a unique file name.

`mkstemp()` changes the transferred string that is specified by *template*. This means that you cannot use a string that is specified by *template* more than once. For each unique temporary file you want to open, you need a new template.

If `mkstemp()` creates a new unique file name, the system first checks whether a file with this name has already existed beforehand. Therefore, if you create more than one unique file name, the same file name component should not be used in *template* for more than one `mkstemp()` call.

**See also**     `getpid()`, `open()`, `tmpfile()`, `tmpnam()`, `stdlib.h`.



**mktemp - make unique temporary file name** (*extension*)

Syntax `#include <stdlib.h>`

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

Description `mktemp()` replaces the contents of the string pointed to by *template* with a unique file name and returns the address of *template*.

The string to which *template* points should contain a file name followed by six Xs. `mkstemp()` replaces these Xs with a letter and the current process ID to create a unique file name.

The letter is chosen so that the new file name will be unique, i.e. will not match any existing file name.

*BS2000*

`mktemp()` creates a unique file name for a temporary SAM file. The name must consist of at least 8 characters and is constructed as follows:

- The first three characters are replaced by "#T."
- The fourth character is replaced by a character that varies for each `mktemp` call (letters A - Z, digits 0 - 9).
- The last four characters are replaced by the TSN of the current task (since LOGON).
- Characters between the first and last four characters remain unchanged.

For example, if the value of *template* is "XXXX.ABC.XXXX" and the TSN of the current task is 6082, the temporary name generated by `mktemp()` at the first call will be:  
#T.A.ABC.6082 □

Return value

Pointer to a string containing the new name  
if successful.

Pointer to an empty string  
if no unique name can be created, e.g. because no more letters are free.

Notes

In the time between the creation of the file name and the opening of the file, another process can create a file with the same name. To avoid this problem, use the `mkstemp()` function.

For reasons of portability to implementations which comply with earlier versions of the X/Open standard, the `tmpnam()` function is recommended for creating a unique file name.

`mktemp()` can create a maximum of 26 unique file names per process for each unique *template*.

*BS2000*

Temporary files are automatically deleted on termination of a task (LOGOFF). However, if the standard prefix (#) for temporary files was changed at system generation, the files are retained. □

The program environment determines whether a BS2000 or POSIX file is created.

**See also** `tmpfile()`, `stdlib.h`.

## mktime - convert local time into time since the Epoch

Syntax `#include <time.h>`

```
time_t mktime(struct tm *timeptr);
```

Description `mktime()` converts the time represented by the `tm` structure pointed to by `timeptr` into a calendar time (the number of seconds since 00:00:00 UTC, Universal Time Coordinated, January 1, 1970).

The `tm` structure has the following format:

```
struct tm {
    int    tm_sec;        /* Seconds [0, 61] */
    int    tm_min;        /* Minutes [0, 59] */
    int    tm_hour;       /* Hours [0, 23] */
    int    tm_mday;       /* Day of the month [1, 31] */
    int    tm_mon;        /* Month [0, 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;      /* Flag for daylight saving time */
};
```

Besides computing the calendar time, `mktime()` normalizes the supplied `tm` structure. 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 indicated in the definition of the structure. Upon successful completion, the values of the `tm_wday` and `tm_yday` components are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to be within the appropriate ranges. The final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

The original values of the components may be either greater than or less than the specified range. For example, a `tm_hour` of -1 means 1 hour before midnight; a `tm_mday` of 0 means the day preceding the current month, and a `tm_mon` of -2 means 2 months before January of the `tm_year`.

If `tm_isdst` is > 0, the original values are assumed to be in the alternate timezone, i.e. summer time applies. If it turns out that the alternate timezone is not valid for the computed calendar time, then the components are adjusted to the main timezone. Conversely, if `tm_isdst` is zero, the original values are assumed to be in the main timezone, i.e. normal time applies, and are converted to the alternate timezone if the main timezone is invalid. If `tm_isdst` is negative, `mktime()` determines the correct timezone.

Local timezone information is set as if `mktime()` had called the `tzset()` function.

### BS2000

`mktime()` converts the date and time, which are specified by the user in a structure of type `tm`, into a time specification of type `time_t`. This is the number of seconds that have elapsed since 00:00:00, January 1, 1950. □

## Return value

Number of seconds

if successful.

`time_t - 1` if the calendar time cannot be represented.

*BS2000*

For local times as of January 1, 1950, 00.00.00, the number of seconds that have elapsed since then (positive value).

For local times prior to January 1, 1950, 00.00.00, the number of elapsed seconds up to that point (negative value). □

## Example Which day of the week is July 4, 2001?

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

struct tm time_str;
char daybuf[20];

int main (void)
{
    time_str.tm_year = 2001 - 1900;
    time_str.tm_mon = 7 - 1;
    time_str.tm_mday = 4;
    time_str.tm_hour = 0;
    time_str.tm_min = 0;
    time_str.tm_sec = 1;
    time_str.tm_isdst = -1;

    if (mktime (&time_str) == -1)
        (void) puts (" -unknown-");
    else {
        (void) strftime (daybuf, sizeof (daybuf), "%A", &time_str);
    }
    return 0;
}
```

## Notes

The value for `tm_year` in the `tm` structure must be for the year 1970 or later. Calendar times before 00:00:00 UTC, January 1, 1970 or after 03:14:07 UTC, January 19, 2038 cannot be represented.

*BS2000*

`mktime()` returns valid values for times ranging from 1.1.1880, 00:00:00, through 1.1.2021, 00:00:00.

## See also

`ctime()`, `getenv()`, `timezone`, `time.h`.

## mmap - map memory pages

**Name**        **mmap**

**Syntax**      #include <sys/mman.h>

```
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

**Description** `mmap()` produces a mapping between the address area of a process ( $[pa, pa + len)$ ) and a file section ( $[off, off + len)$ ).

The call has the following format:

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

A mapping is produced between the address space of the process at the address  $pa$  for  $len$  bytes on the one hand and the file described by the file descriptor  $fildes$  with the offset  $off$  for  $len$  bytes on the other.

The value of  $pa$  is an implementation-dependent function of  $addr$  and the value  $flags$ . A successful `mmap()` call returns  $pa$  as the result. The address areas defined by  $[pa, pa + len)$  and  $[off, off + len)$  must be permissible for the possible (not necessarily the current) address area of the process or the file. `mmap()` cannot enlarge a file.

The mapping, which is produced by `mmap()` with `MAP_FIXED`, replaces all previous mappings for the pages of the process in the area  $[pa, pa + len)$ .

If the size of the mapped file is changed after the `mmap()` call, the effect on references to mapping sections which correspond to a newly added or deleted part of the file is undefined.

`mmap()` is supported for normal files only.

The  $prot$  parameter determines whether read, write or execute accesses or combinations of these are to be allowed for the mapped pages. The access permissions are defined in `sys/mman.h` as follows:

<code>PROT_READ</code>	Page can be read.
<code>PROT_WRITE</code>	Page can be written.
<code>PROT_EXEC</code>	Page can be executed.
<code>PROT_NONE</code>	Page cannot be accessed.

`PROT_WRITE` is implemented as `PROT_WRITE|PROT_EXEC` and `PROT_EXEC` as `PROT_READ|PROT_EXEC`.

Three states are possible:

- the page cannot be accessed
- access to the page is read-only
- the page can be accessed for both reading and writing

The behavior of `PROT_WRITE` can be influenced by the `MAP_PRIVATE` in the *flags* parameter, as described in more detail below.

The *flags* parameter contains further information on the handling of the mapped pages. The options are defined in `sys/mman.h` as follows:

<code>MAP_SHARED</code>	Changes are shareable
<code>MAP_PRIVATE</code>	Changes are private
<code>MAP_FIXED</code>	<i>addr</i> must be interpreted exactly

`MAP_SHARED` and `MAP_PRIVATE` control the visibility of write accesses to the memory pages. Either `MAP_SHARED` or `MAP_PRIVATE` must be specified.

The mapping type is retained after a `fork()`.

If `MAP_SHARED` is specified, write accesses to the memory pages modify the file, and the changes are visible in all mappings of the relevant file section produced with `MAP_SHARED`.

If `MAP_PRIVATE` is specified, write accesses to the memory pages do not modify the file, and the changes are not visible to any other process which maps the relevant file section. The first write access generates a privately kept copy of the memory pages and redirects the mapping to the copy. Note that the privately kept copy is not generated until the first write access; until then, other users who have mapped the file section with `MAP_SHARED` can modify the file section.

`MAP_FIXED` defines that the value of *pa* must match *addr* exactly. Use of `MAP_FIXED` is not recommended, as this parameter can prevent effective use of the system resources.

If `MAP_FIXED` is not set, depending on the implementation an address *pa* is returned through selection of an area from the address space of the process which the system considers suitable for mapping *len* bytes.

An *addr* value of zero means that *pa* can be freely selected in accordance with the conditions described below. If *addr* has a value other than zero, this is interpreted as a suggestion to select the mapping close to this address.

On no account is the value 0 selected for *pa*, is an existing mapping overwritten or does mapping take place to dynamically assigned memory areas.

The *off* parameter is subject to restrictions in its size and alignment, which are based on the return value of `sysconf()` with regard to the `_SC_PAGESIZE` and `_SC_PAGE_SIZE` parameters. If `MAP_FIXED` is specified, the *addr* parameter must also comply with these restrictions.

The system performs mapping operations over entire pages. Because the *len* parameter is not tied to specific sizes or alignments, the system includes in the mapping operation all page remainders which arise during mapping of the area [*pa*, *pa* + *len*).

The system fills such part-pages with zeros at the end of a [*pa*, *pa* + *len*) memory area. Changes to this area are not written back.

If the mapping extends over whole pages which are located after the last byte of the file,

references to these pages generate a SIGSEGV signal.

SIGSEGV signals can also be sent in the case of various error conditions of the file system, including exceeding of the quota.

`mmap()` generates an additional reference to the file that is described by *fildev*. This reference is not deleted when a `close()` is issued for *fildev*, but only when no more mappings to the file exist.

#### Return value

*pa* Address where the mapping was placed.  
 -1 in the event of an error. `errno` is set to indicate the error.

#### Errors

`mmap()` will fail if:

EACCES *fildev* is not opened for reading, regardless of the specified *prot* argument, or *fildev* is not opened for writing and `PROT_WRITE` was requested in an mapping of type `MAP_SHARED`.

EAGAIN The mapping cannot be locked in the memory.

EBADF *fildev* is not a valid open file descriptor.

ENXIO Addresses in the [*off*, *off + len*) area are invalid for *fildev*.

EINVAL The *off* argument (or *addr* if `MAP_FIXED` was specified) does not contain a multiple of the page length returned by `sysconf()`, or *off* or *addr* has an invalid value.  
 The value in *flags* is invalid (neither `MAP_PRIVATE` nor `MAP_SHARED` is set).  
 The *len* argument has a value less than or equal to 0.

EMFILE The number of mappings exceeds the maximum permissible value.

ENOMEM `MAP_FIXED` was specified and the [*addr*, *addr + len*) area exceeds the address area allowed for a process, or `MAP_FIXED` was not specified but there is not enough storage space available in the address area for the mapping.

ENODEV *fildev* refers to a file whose type is not supported by `mmap()`, e.g. a special file.

EOVERFLOW The value of *off* plus *len* exceeds the offset maximum specified in the internal description of the open file assigned to *fildev*.

#### Notes

The use of `mmap()` reduces the space available for other functions which also occupy storage space.

The specification `MAP_FIXED` is not recommended, as this parameter can prevent effective use of the system resources.

The application must make sure that the file accesses are synchronized if `mmap()` is used together with other file access methods like `read()`, `write()`, standard input/output and `shmat()`.

`mmap()` allows access to resources via address area manipulations in place of the read/write interface. If a file is mapped, a processes need only access the address to which the file object is mapped. Observe the following (incomplete) code:

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
    /* Use data in buf */
```

Using `mmap()`, the code can be rewritten as follows:

```
fildes = open(...)
address =mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
    /* Use address data */
```

**See also** `exec()`, `fcntl()`, `fork()`, `lockf()`, `munmap()`, `msync()`, `mprotect()`, `shmat()`, `sysconf()`, `sys/mman.h`.



## modf - split floating-point number into integral and fractional parts

**Syntax**      `#include <math.h>`  
`double modf(double x, double *iptr);`

**Description** `modf()` splits a floating-point number *x* into its integral and fractional parts. Both parts receive the sign of *x*. `modf()` returns the fractional part of *x* as the result and writes the integral part as a value of type `double` to the address to which *iptr* points.

**Return value**  
Fractional part of *x* with the sign of *x* if successful.  
0                    if an error occurs.

**Notes**        The *iptr* argument must be a pointer!

**See also**     `frexp()`, `ldexp()`, `math.h`.

## mount - mount file system *(extension)*

Syntax

```
#include <sys/types.h>
#include <sys/mount.h>

int mount(const char *spec, const char *dir, int mflag,
          [int fstyp, const char *dataptr, size_t datalen]);
```

Description `mount()` mounts a removable file system, which is contained in the block-special file identified by *spec*, in an existing directory *dir* (mount point).

*spec* and *dir* are pointers to pathnames.

*mflag* can assume the following values:

MS\_FSS           to describe a file system type.

MS\_DATA          to describe a block of file-system specific data of length *datalen* starting at the address *dataptr*.

MS\_RDONLY        if the file system is to be mounted as read-only, in which case, no further arguments are expected.

The argument *fstyp* is interpreted by `mount()` when either MS\_FSS or MS\_DATA is set in *mflag*. *fstyp* is the file system type number or a pointer to a string containing the file system type. The system call `sysfs()` can be used to determine the file system type number.

If neither MS\_FSS nor MS\_DATA are set in *mflag*, `mount()` defaults to the root file system type.

If MS\_DATA is set in *mflag*, the system expects the *dataptr* and *datalen* arguments. This data is interpreted by file-system specific code within the operating system. Its format depends on the file system type. If a particular file system type does not require this data, *dataptr* and *datalen* should both be zero.

Upon successful completion of `mount()`, the name in *dir* refers to the root directory on the newly mounted file system.

Return value

0                upon successful completion.

-1               if an error occurs. `errno` is set to indicate the error.

Errors	<code>mount()</code> will fail if:
EBUSY	<i>dir</i> is already mounted at the time of the call, or <i>dir</i> has some other owner, or <i>dir</i> is otherwise busy, or the special file associated with <i>spec</i> is currently mounted, or no more mount table entries are available.
EFAULT	<i>spec</i> , <i>dir</i> or <i>datalen</i> points outside the allocated address space of the process.
EINVAL	The super-block has an invalid magic number or <i>fstyp</i> is invalid.
ELOOP	Too many symbolic links were encountered in resolving <i>dir</i> .
ENAMETOOLONG	The length of the <i>dir</i> argument exceeds <code>PATH_MAX</code> or <code>NAME_MAX</code> .
ENOENT	One of the named files is not recognized.
ENOSPC	The file system state in the super-block is not <code>FsOKAY</code> and <i>mflag</i> requests write permission.
ENOTBLK	<i>spec</i> is not a block-special file.
ENOTDIR	A component of <i>spec</i> or <i>dir</i> is not a directory.
ENXIO	The special file associated with <i>spec</i> is not recognized.
EPERM	The effective user ID is not that of a process with appropriate privileges.
EREMOTE	<i>spec</i> is not local and cannot be mounted.
EROFS	<i>spec</i> is write-protected, and <i>mflag</i> requests write permission.

Notes `mount()` may only be called under the effective user ID of a process with appropriate privileges.

As soon as a directory is mounted, it is treated as a subtree. In other words, files on the mounted file system can be accessed by processes without making allowances for the fact a mounted file system is involved. Links across file system boundaries with `link()` are not permitted, however, since that function checks the file system of a file.

The interface is intended only for the `mount` command.

See also `sysfs()`, `umount()`, and the commands `mount` and `fsck` in the manual "POSIX Commands (BS2000/OSD)".

## mprotect - modify access protection for memory mapping

Syntax `#include <sys/mman.h>`  
`int mprotect(void *addr, size_t len, int prot);`

Description The `mprotect()` function changes the access permissions for the mappings in the [*addr*, *addr* + *len*) area to the access permission specified in *prot*. The value specified in *len* is rounded to a multiple of the page size specified by `sysconf()`. All values that can also be specified in `mmap()` are permissible for *prot*:

The values for *prot* are defined in `sys/mman.h` as follows:

<code>PROT_READ</code>	Page can be read.
<code>PROT_WRITE</code>	Page can be written.
<code>PROT_EXEC</code>	Page can be executed.
<code>PROT_NONE</code>	Page cannot be accessed.

If `mprotect()` fails but the reason is not `EINVAL`, it may be that the access permissions of some pages were already changed in the specified area [*addr*, *addr* + *len*). If the error is in the address *addr2*, the access permissions of all whole pages in the area [*addr*, *addr2*] will be changed.

Return value

0	if successful.
-1	if an error occurs. <code>errno</code> is set to indicate the error.

Errors Under the conditions described below, the `mprotect()` function will fail and set `errno` to the following values:

<code>EACCES</code>	<i>prot</i> contains a value that does not match the access permissions of the process for the underlying file.
<code>EAGAIN</code>	<i>prot</i> contains the value <code>PROT_WRITE</code> for a mapping of type <code>MAP_PRIVATE</code> and a memory bottleneck occurs, i.e. the storage resources for reserving and locking the private page are not sufficient.
<code>EINVAL</code>	<i>addr</i> is not a multiple of the page size specified by <code>sysconf()</code> , or the <i>len</i> argument contains a value less than or equal to 0.
<code>ENOMEM</code>	Addresses in the area [ <i>addr</i> , <i>addr</i> + <i>len</i> ) are invalid for the address area of the process, or one or more pages have been specified which are not mapped.

See also `mmap()`, `sysconf()`, `sys/mman.h`.

## mrand48 - generate pseudo-random numbers between $-2^{31}$ and $2^{31}$

Syntax      `#include <stdlib.h>`  
             `long int mrand48 (void);`

Description See `drand48()`.

## msgctl - message control operations

**Syntax**      `#include <sys/msg.h>`

```
int msgctl(int msgid, int cmd, struct msqid_ds *buf);
```

**Description** `msgctl()` provides message control operations as specified by *cmd*. The possible values for *cmd*, and the message control operations they specify, are:

**IPC\_STAT**      Place the current value of each member of the data structure associated with *msgid* into the structure pointed to by *buf*. The contents of this structure are defined in `sys/msg.h`.

**IPC\_SET**      Set the value of the following members of the `msqid_ds` data structure associated with *msgid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode
msg_qbytes
```

**IPC\_SET** can only be executed by a process with appropriate privileges or a process that has an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` data structure associated with *msgid*. Only a process with appropriate privileges can raise the value of `msg_qbytes`.

**IPC\_RMID**      Remove the message queue identifier specified by *msgid* and destroy the message queue and the data structure associated with it. **IPC\_RMID** can only be executed by a process with appropriate privileges or one that has an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` data structure associated with *msgid*.

**Return value**

0                  if successful.

-1                 if an error occurs. `errno` is set to indicate the error.

**Errors**         `msgctl()` will fail if:

**EACCES**         *cmd* is **IPC\_STAT** and the calling process does not have read permission.

*Extension*

EFAULT	<i>buf</i> points to an invalid address. □
EINVAL	<i>msqid</i> is not a valid message queue identifier, or <i>cmd</i> is not a valid operation, or <i>cmd</i> is IPC_SET, and <i>msg_perm.uid</i> or <i>msg_perm.gid</i> is invalid.
EPERM	<i>cmd</i> is IPC_SET, and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and not equal to the value of <i>msg_perm.cuid</i> or <i>msg_perm.uid</i> in the data structure associated with <i>msqid</i> .
EPERM	<i>cmd</i> is IPC_SET, an attempt is being made to increase to the value of <i>msg_qbytes</i> , and the effective user ID of the calling process does not have appropriate privileges.

**Notes** The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

**See also** `msgget()`, `msgrcv()`, `msgsnd()`, `sys/msg.h`, section "Interprocess communication" on page 112.

## msgget - get message queue

**Syntax**      `#include <sys/msg.h>`  
                  `int msgget(key_t key, int msgflg);`

**Description** `msgget()` returns the message queue identifier associated with *key*.

A message queue identifier, associated message queue, and data structure (see `sys/msg.h`) are created for *key* if one of the following is true:

- *key* is `IPC_PRIVATE`.
- *key* does not already have a message queue identifier associated with it, and  $(msgflg \& IPC\_CREAT)$  is non-zero.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid` and `msg_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 `msg_perm.mode` are set equal to the low-order 9 bits of *msgflg*.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime` and `msg_rtime` are set equal to 0.
- `msg_ctime` is set equal to the current time.
- `msg_qbytes` is set equal to the system limit.

**Return value**  
 Non-negative integer (message queue identifier)  
                  if successful.

-1                if an error occurs. `errno` is set to indicate the error.

**Errors**        `msgget()` will fail if:

- |                     |   |
|---------------------|---|
| <code>EACCES</code> | A message queue identifier exists for the argument <i>key</i> , but the access permissions specified by the low-order 9 bits of <i>msgflg</i> are not granted (see section "Interprocess communication" on page 112). |
| <code>EEXIST</code> | A message queue identifier exists for the argument <i>key</i> , but the value of $((msgflg \& IPC\_CREAT) \&\& (msgflg \& IPC\_EXCL))$ is non-zero.   |
| <code>ENOENT</code> | A message queue identifier exists for the argument <i>key</i> and $(msgflg \& IPC\_CREAT)$ is 0.  |
| <code>ENOSPC</code> | A message queue identifier is to be created, but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.   |



**Notes**        The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

**See also**     `msgctl()`, `msgrcv()`, `msgsnd()`, `sys/msg.h`, section "Interprocess communication" on page 112.

## msgrcv - receive message from queue

Syntax `#include <sys/msg.h>`

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

Description `msgrcv()` reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user-defined buffer pointed to by *msgp*.

*msgp* points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg
{
    long int mtype;        /* Message type */
    char mtext[1];       /* Message text */
}
```

The structure member `mtype` is the type of the received message, as specified by the sending process.

The structure member `mtext` is the text of the message.

*msgsz* specifies the size in bytes of `mtext`. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and `(msgflg & MSG_NOERROR)` is non-zero. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

*msgtyp* specifies the type of message requested as follows:

- If *msgtyp* is 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message that is less than or equal to the absolute value of *msgtyp* is received.

*msgflg* specifies the action to be taken if a message of the desired type is not on the queue. The following actions are possible:

- If `(msgflg & IPC_NOWAIT)` is non-zero, the calling process will return immediately with a return value of -1, and `errno` set to `ENOMSG`.
- If `(msgflg & IPC_NOWAIT)` is 0, the calling process will suspend execution until one of the following events occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier *msqid* is removed from the system; when this occurs, `errno` is set equal to `EIDRM`, and -1 is returned.

- The calling process receives a signal that is to be caught; in this case a message is not received and the calling process resumes execution in the manner prescribed in `sigaction()`.

Upon successful completion, the following actions are taken with respect to the data structure associated with `msgid`:

- `msg_qnum` is decremented by 1.
- `msg_lrp_id` is set equal to the process ID of the calling process.
- `msg_rtime` is set equal to the current time.

If threads are used, then the function affects the process or a thread in the following manner: The `msgflg` parameter refers to the calling thread.

### Return value

number of bytes placed in `mtext` if successful.

-1 if an error occurs. `errno` is set to indicate the error.

### Errors

`msgrcv()` will fail if:

E2BIG The value of `mtext` is greater than `msgsz` and (`msgflg` & `MSG_NOERROR`) is 0.

EACCES Operation permission is denied to the calling process.

#### Extension

EFAULT `msgp` points to an invalid address. □

EIDRM The message queue identifier `msgid` is removed from the system.

EINTR `msgrcv()` was interrupted by a signal.

EINVAL `msgid` is not a valid message queue identifier; or the value of `msgsz` is less than 0.

ENOMSG The queue does not contain a message of the desired type and (`msgtyp` & `IPC_NOWAIT`) is non-zero.

### Notes

`msgp` should be converted to type `void *`.

The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

### See also

`msgctl()`, `msgget()`, `msgsnd()`, `sigaction()`, `sys/msg.h`, section "Interprocess communication" on page 112.

## msgsnd - send message to queue

Syntax `#include <sys/msg.h>`

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Description `msgsnd()` sends a message to the queue associated with the message queue identifier specified by *msqid*.

*msgp* points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg
{
    long int mtype;        /* Message type */
    char mtext[1];       /* message text */
}
```

The structure member `mtype` is a non-zero positive type `long int` that can be used by the receiving process for message selection.

The structure member `mtext` is any text of length *msgsz* bytes. The argument *msgsz* can range from 0 to a system-imposed maximum.

*msgflg* specifies the action to be taken if one or more of the following conditions are true:

- The number of bytes already on the queue is equal to `msg_qbytes` (see `sys/msg.h`).
- The total number of messages on all queues system-wide is already equal to the system-imposed limit.

These actions are as follows:

- If (*msgflg* & `IPC_NOWAIT`) is non-zero, the message will not be sent, and the calling process will return immediately.
- If (*msgflg* & `IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following events occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue identifier *msqid* is removed from the system; when this occurs, `errno` is set equal to `EIDRM`, and -1 is returned.
  - The calling process receives a signal that is to be caught; in this case, the message is not sent, and the calling process resumes execution in the manner prescribed in `sigaction()`.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set equal to the process ID of the calling process.
- `msg_stime` is set equal to the current time.

If threads are used, then the function affects the process or a thread in the following manner: The *msgflg* parameter refers to the calling thread.

#### Return value

- 0 if successful.
- 1 if an error occurs. `errno` is set to indicate the error.

#### Errors

`msgsnd()` will fail if:

- `EACCES` Operation permission is denied to the calling process.
- `EAGAIN` The message cannot be sent for one of the reasons cited above and (*msgflg* & `IPC_NOWAIT`) is non-zero.

#### Extension

- `EFAULT` *msgp* points to an invalid address. □
- `EIDRM` The message queue identifier *msqid* is removed from the system.
- `EINTR` `msgsnd()` was interrupted by a signal.
- `EINVAL` *msqid* is not a valid message queue identifier, or the value of `mtype` is less than 1; or the value of *msgsz* is less than 0 or greater than the system-imposed limit.

#### Notes

The value of the *msgp* argument should be converted to type `void *`.

The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

#### See also

`msgctl()`, `msgget()`, `msgrcv()`, `sigaction()`, `sys/msg.h`, section "Interprocess communication" on page 112.

## msync - synchronize memory

Syntax `#include <sys/mman.h>`  
`int msync(void *addr, size_t len, int flags);`

Description The `msync()` function writes all modified copies of pages in the `[addr, addr + len)` area back to the appropriate storage media or makes copies in the memory invalid so that later accesses to these pages will access the storage medium. The storage medium for a modified mapping of type `MAP_SHARED` is the file to which the page is mapped; the storage medium for a modified mapping of type `MAP_PRIVATE` is its paging area.

*flags* must have one of the following values:

<code>MS_ASYNC</code>	Perform asynchronous write accesses
<code>MS_SYNC</code>	Perform synchronous write accesses
<code>MS_INVALIDATE</code>	Mark mappings as invalid

If `MS_ASYNC` or `MS_SYNC` are set, `msync()` synchronizes the file contents with the current contents of the allocated storage area:

All write accesses to the storage area that have taken place before the `msync()` call are visible during read accesses to the file after `msync()`.

Before `msync()` is called, however, it is undefined whether write accesses to the corresponding file section will be visible during subsequent read accesses.

If `MS_ASYNC` is set, `msync()` returns as soon as all write operations have been initiated; if `MS_SYNC` is set, `msync()` does not return until all write operations are completed.

If `MS_INVALIDATE` is set, `msync()` synchronizes the memory area with the current contents of the assigned file section. Afterwards, all copies of data that are located in a cache memory are marked as invalid. Later references to these pages are handled by the system via the underlying storage medium.

All write accesses to the mapped file section that took place before the `msync()` call are visible during subsequent read accesses to the allocated memory area.

Before `msync()` is called, however, it is undefined whether write accesses to the corresponding file section will be visible during subsequent read accesses.

Return value

0	if successful.
-1	if an error occurs. <code>errno</code> is set to indicate the error.

- Errors** Under the conditions described below, the `msync()` function will fail and set `errno` to the following values:
- `EINVAL` *addr* is not a multiple of the page size defined by `sysconf()`.
  - `ENOMEM` Addresses in the [*addr*, *addr + len*) area are invalid for the address area of the process, or one or more pages have been specified which are not mapped.
  - `EIO` An I/O error occurred during read or write access to the file.
- Notes** `msync()` should be used if it is required that a memory object be in a known state, e.g. in transaction processing.
- Memory pages can also be written to disk in the course of normal system operations. Therefore it cannot be guaranteed that memory pages are only written to disk when `msync()` is called.
- See also** `mmap()`, `sysconf()`, `sys/mman.h`

## munmap - unmap memory pages

**Syntax**      `#include <sys/mman.h>`  
`int munmap(void *addr, size_t len);`

**Description** The `munmap()` function removes mappings from pages in the area [*addr*, *addr* + *len*). The value specified in *len* is rounded to a multiple of the page size defined by `sysconf()`. Further references to these pages result in a `SIGSEGV` signal to the process, provided that a new mapping of these pages was not established in the meantime.

Areas within the specified interval which are not `mmap` mappings are ignored.

**Return value**

0	if successful.
-1	if an error occurs. <code>errno</code> is set to indicate the error.

**Errors** `munmap()` will fail if:

<code>EINVAL</code>	<i>addr</i> is not a multiple of the page size defined by <code>sysconf()</code> , or addresses in the area [ <i>addr</i> , <i>addr</i> + <i>len</i> ) are not valid for the address area of the process, or the <i>len</i> argument contains a value less than or equal to 0.
---------------------	--

**See also** `mmap()`, `sysconf()`, `signal.h`, `sys/mman.h`.



# Functions and variables in alphabetical order (n - y)

## nanosleep - suspend current thread

**Syntax**      `#include <time.h>`

```
int nanosleep(const struct timespec * rqtp, struct timespec * rmtp);
```

**Description** The function `nanosleep()` suspends the current thread until the time period specified via *rqtp* expires or until a signal is sent to the calling thread that results in the calling of a signal handling routine or the termination of the process. The time in suspension may be longer than the time specified because the value was rounded up to be many times greater than the sleep resolution or because the system still needs to carry out other activities.

**Return value**

0                      if the specified time expires.

- 1                    if `nanosleep()` was interrupted by a signal. If *rmtp* is not a null pointer, the time remaining will be stored in this case in the structure pointed to by *rmtp*. If *rmtp* is NULL, the time remaining is not returned.

`errno` is set to indicate an error.

**Errors**            `nanosleep()` fail if:

EINTR                `nanosleep()` was interrupted by a signal.

EINVAL               a value was specified in nanoseconds that is less than 0 or greater than or equal to 1000 million in the *rqtp* argument.

ENOSYS               The function `nanosleep()` is not supported in this implementation.

**See also**          `sleep()`, `time.h`.

## nextafter - next displayable floating-point number

**Syntax**      `#include <math.h>`

```
double nextafter (double x, double y);
```

**Description** `nextafter()` returns the next displayable floating-point number that follows *x* in direction *y*. If *y* is less than *x*, the largest displayable floating-point number smaller than *x* is returned.

**Return value**

Next displayable floating-point number that follows *x* in direction *y* if successful.

If *x* is finite but the result of `nextafter(x, y)` would cause an overflow, the value `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

**Errors**      `nextafter()` will fail if:

`ERANGE`      the correct value would cause an overflow

**See also**      `math.h`.

## nftw - traverse file tree

Syntax `#include <ftw.h>`

```
int nftw (const char *path,
         int (*fn) (const char *, const struct stat *, in , struct FTW *),
         int depth, int flags);
```

Description `nftw()` recursively searches the directory hierarchy that begins with *path*. `nftw()` works similarly to `ftw()`, but also processes the *flags* argument, which is formed by bitwise inclusive ORing of the following values:

FTW_CHDIR	The searched directory becomes the current working directory. If FTW_CHDIR is not set, the current working directory remains unchanged.
FTW_DEPTH	Before the directory itself, all subdirectories are traversed. If FTW_DEPTH is not set, the directory is traversed first.
FTW_MOUNT	Only directories in the same file system as <i>path</i> are traversed. If FTW_MOUNT is not set, mounted directories are also traversed.
FTW_PHYS	The directory hierarchy is physically traversed; <code>nftw()</code> does not follow any symbolic links, but reports the links instead. If FTW_PHYS is not set, <code>nftw()</code> follows symbolic links. <code>nftw()</code> does not report the same file twice.

For every file or directory found, `nftw()` calls the user-defined function *fn* with the following four arguments:

1. Pathname of the object.
2. Pointer to the *stat* buffer containing information on the object.
3. Number of type integer, in which `nftw()` provides additional information.

FTW_F	The object is a file.
FTW_D	The object is a directory.
FTW_DP	The object is a directory; subdirectories have already been traversed (this situation can only occur if <i>flags</i> contains the value FTW_DEPTH).
FTW_SLN	The object is a symbolic link that points to a non-existent file (this situation can only occur if the <i>flags</i> does not contain the value FTW_PHYS).
FTW_DNR	The object is a directory which cannot be read. <i>fn()</i> is not called for any of the files in it or directories under it.

FTW\_NS            `stat()` cannot process the object because the access permissions are not sufficient. The `stat` buffer passed to `fn` is undefined. If `stat()` fails for other reasons, `nftw()` will fail and return -1.

#### 4. Pointer to a struct FTW which contains the following elements:

```
int base;
int level;
```

`nftw()` uses one file descriptor for each level in the file tree. The *depth* argument limits the number of file descriptors used. If *depth* is zero or negative, this has the same effect as the value 1. *depth* must not be greater than the number of file descriptors available at the specified time. If the `nftw()` function returns, it closes all file descriptors that it opened but none of the ones that were opened by `fn`.

`nftw()` descends the file tree from the highest hierarchy level onward until either the tree has been exhausted, an `fn` call returns a non-zero value, or an error is detected within `nftw()` (e.g. an I/O error).

#### Return value

0                    if the tree has been exhausted and `fn()` always returned the value 0.

#### Return value of the `fn()` function

if `fn()` returns a value  $\neq 0$ , `nftw()` stops traversing the file tree and returns the value that was returned by `fn`

-1                   if `nftw()` detects an error other than EACCES. `errno` is set to indicate the error.

#### Errors

`nftw()` will fail if:

EACCES            Search permission is denied for any component of *path*, or read permission is denied for *path* or `fn()` returns the value -1 and does not reset.

ENAMETOOLONG

The length of the *path* argument is greater than `{PATH_MAX}` or a component of the pathname is longer than `{NAME_MAX}`.

The resolving a symbolic link led to an interim result whose length exceeds `{PATH_MAX}`.

ENOENT            A component of the path prefix does not exist or *path* is an empty string.

ENOTDIR           A component of *path* is not a directory.

ELOOP            Too many symbolic links were encountered in resolving *path*.

EMFILE            `{OPEN_MAX}` file descriptors are already open.

ENFILE            Too many files are open.

`errno` can also be set if the function pointed to by *fn()* sets `errno`.

**Notes**            Since `nftw()` is recursive, it is possible for it to terminate with a memory error when applied to very deep file structures.

**See also**        `lstat()`, `opendir()`, `readdir()`, `stat()`, `ftw.h`.

## nice - change priority of process

Syntax      `#include <unistd.h>`  
`int nice(int incr);`

Description `nice()` adds the value of *incr* to the nice value of the calling process. Note that in the C runtime system, changing the nice value with *incr* has no effect on the priority of a process. The function is supported only for conformance with XPG4.

A process nice value is a non-negative integer for which a more positive value results in lower CPU priority. A maximum nice value of  $2^{\{NZERO\}}-1$  and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit. Only a process with appropriate privileges can lower the nice value.

If threads are used, then the function affects the process or a thread in the following manner: Changes the priority of a process. If the process is multithreaded, the scheduling priority affects all threads of the process throughout the scope of the system.

Return value  
New nice value minus  $\{NZERO\}$   
upon successful completion.  
-1            if an error occurs. The process nice value is not changed, and `errno` is set to indicate the error.

Errors        `nice()` will fail if:  
`EPERM`      *incr* is negative or greater than  $2^{\{NZERO\}}-1$ , and the calling process does not have appropriate privileges.

Notes        As -1 is a permissible return value in a successful situation, an application wishing to check for error situations should set `errno` to 0, then call `nice()`, and if it returns -1, check to see if `errno` is non-zero.

See also     `limits.h`, `unistd.h`.

## nl\_langinfo - get locale values

Syntax `#include <langinfo.h>`

```
char *nl_langinfo(nl_item item);
```

Description `nl_langinfo()` returns the value of the constant *item* in the current locale or environment. The available constants and values for *item* are defined in `langinfo.h`.

Return value

Pointer to a string of the locale

if no `langinfo` data is defined in an environment.

Null pointer if *item* is invalid.

Notes The array pointed to by the return value should not be modified by the program, but may be modified by further calls to `nl_langinfo()`. In addition, calls to `setlocale()` with a category corresponding to the category of *item* or to the category `LC_ALL` may overwrite the array.

If `setlocale()` is not called in an application, the current locale in the POSIX subsystem defaults to "POSIX". The return values of `nl_langinfo()` are based on the current locale. If the current locale does not contain any value for a given parameter, the corresponding value of the default is returned.

See also `setlocale()`, `langinfo.h`, `nl_types.h`, section "Locale" on page 53 and section "Environment variables" on page 71.

## nrand48 - generate pseudo-random numbers between 0 and $2^{31}$ with initialization value

Syntax `#include <stdlib.h>`

```
long int nrand48 (unsigned short int xsubi[3]);
```

Description See `drand48()`.

## offsetof - get offset of structure component from start of structure *(BS2000)*

Syntax	<pre>#include &lt;stddef.h&gt;  size_t offsetof(<i>type</i>, <i>component</i>);</pre>
Description	<p><code>offsetof()</code> returns the offset in bytes between the named structure <i>component</i> and the start of the structure of the specified <i>type</i>. <code>offsetof()</code> is a macro.</p> <p><i>type</i> is the name of the structure type (label). <i>component</i> is the name of the structure component.</p>
Return value	Offset of the structure component from the start of the structure in bytes if successful.
Notes	If the specified structure component is a bit field, the behavior is undefined.



## open - open file

**Name**        **open, open64**

**Syntax**      `#include <sys/types.h>`  
`#include <sys/stat.h>`  
`#include <fcntl.h>`

```
int open (const char *path, int oflag , .../* mode_t mode*/);  
int open64 (const char *path, int oflag , .../* mode_t mode*/);
```

**Description** If POSIX files are executed, the behavior of this function conforms to the XPG4 standard as described below:

The `open()` function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The `path` argument points to a pathname naming the file.

The `open64()` function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The `path` argument points to a pathname naming the file.

`open()` will return a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other process in the system. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor will be cleared (see `fcntl()`).

The file position indicator is set to the beginning of the file.

The file status byte and file access modes of the open file description will be set according to the value of `oflag`.

Values for `oflag` are constructed by a bitwise-inclusive-OR of flags from the following list, defined in `fcntl.h`. Applications must specify exactly one of the first three values (file access modes) below in the value of `oflag`:

<code>O_RDONLY</code>	Open for reading only.
<code>O_WRONLY</code>	Open for writing only.
<code>O_RDWR</code>	Open for reading and writing. The result is undefined if this flag is applied to a FIFO file.

Any combination of the following flags may be used:

- O\_APPEND** The file position indicator is set to the end of the file prior to each write.
- O\_CREAT** If the file exists, this flag has no effect, except under the conditions noted under **O\_EXCL** below. Otherwise, the file is created; the user ID of the file is set to the effective user ID of the process, and the group ID of the file is set to the effective group ID of the process or the group ID of the file's parent directory. The access permission bits (see `sys/stat.h`) of the file mode are set to the value of *mode* and then modified as follows: a bitwise-AND is performed on the individual file-mode bits and the corresponding bits in the complement of the process' file mode creation mask (see `umask()`). Thus, all bits in the file mode for which a corresponding bit is set in the file mode creation mask are cleared. When bits other than the file permission bits are set, the effect is unspecified. The *mode* argument does not affect whether the file is opened for reading, writing or for both.
- O\_EXCL** `open()` will fail if **O\_CREAT** and **O\_EXCL** are set and the file exists. If the file does not exist, the two actions, i.e. the check for the existence of the file and the creation of the file, are treated as a single action that is shielded from intervention by other processes executing `open()` for the same file name in the same directory with **O\_EXCL** and **O\_CREAT** set. If **O\_CREAT** is not set, the effect is undefined.
- O\_NOCTTY** If this flag is set and *path* identifies a terminal device, `open()` will not cause the terminal device to become the controlling terminal for the process.
- O\_NONBLOCK** When opening a FIFO for reading or writing (**O\_RDONLY** or **O\_WRONLY**):
- If **O\_NONBLOCK** is set:  
An `open()` for reading only will return without delay. An `open()` for writing only will return an error if no process currently has the file open for reading.
  - If **O\_NONBLOCK** is clear:  
An `open()` for reading only will block (i.e. wait) until a process opens the file for writing. An `open()` for writing only will block until a process opens the file for reading.
- When opening a block special or character special file that supports non-blocking opens:
- If **O\_NONBLOCK** is set:  
`open()` will return without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.
  - If **O\_NONBLOCK** is clear:  
The `open()` function will block until the device is ready or available before returning. Otherwise, the behavior of **O\_NONBLOCK** is undefined.

- `O_SYNC` If `O_SYNC` is set on a regular file, writes to that file will cause the process to block until the data is delivered to the underlying hardware.
- `O_TRUNC` If the file exists and is a regular file, and the file is successfully opened `O_RDWR` or `O_WRONLY`, its length is truncated to 0 and the mode and owner are unchanged. This has no effect on FIFO special files or terminal device files. The effect on other file types is implementation-dependent. The result of using `O_TRUNC` with `O_RDONLY` is undefined.
- `O_LARGEFILE` If specified, the offset maximum specified in the internal description of the open file is the highest value that can be correctly represented in an object of type `off64_t`.

If `O_CREAT` is set and the file did not previously exist, upon successful completion, `open()` will mark for update the `st_atime`, `st_ctime` and `st_mtime` fields of the file and the `st_ctime` and `st_mtime` fields of the parent directory.

If `O_TRUNC` is set and the file did previously exist, upon successful completion, `open()` will mark for update the `st_ctime` and `st_mtime` fields of the file.

There is no difference in functionality between `open()` and `open64()` except that `open64()` implicitly sets the `O_LARGEFILE` bit of the file status flag. The function `open64()` corresponds to using the function `open()` when `O_LARGEFILE` is set in *oflag*.

If threads are used, then the function affects the process or a thread in the following manner:

Opening a file; If `O_NONBLOCK` is not set in the parameter *oflag*, the following applies to FIFO: an `open()` for reading blocks the calling thread until a thread opens the file for writing. An `open()` for writing blocks the thread until a thread opens the file for reading. If a block-oriented or character-oriented device file is opened that supports non-waiting opens, the following applies: the `open()` function blocks the calling thread until the device has finished or is available.

#### Extension

If `O_CREAT` and `O_EXCL` are set and *path* is a symbolic link, the link is not followed.

#### BS2000

The following must be noted when executing BS2000 files:

`const char *path` is a string specifying the file to be opened. *path* can be:

any valid BS2000 file name.

- `link=linkname`  
*linkname* designates a BS2000 link name.
- (SYSDTA), (SYSOUT), (SYSLST), the corresponding system file
- (SYSTEMR), terminal I/O
- (INCORE), temporary binary file that is created in virtual memory only.

*mode* is an integer variable whose octal value specifies the desired access mode, i.e.:

0000	Open for reading. The file must already exist.
0001	Open for writing. The file must already exist. The previous contents are retained.
01001	Open for writing. If the file exists, the previous contents are deleted. If the file does not exist, it is created.
0002	Open for reading and writing. The file must already exist. The previous contents are retained.
01002	Open for reading and writing. If the file exists, the previous contents are deleted. If the file does not exist, it is created.
0003	Open for writing and reading. If the file exists, the previous contents are deleted. If the file does not exist, it is created.
0401	Open for appending to the end of the file. The file must already exist. The file is positioned to end of file, i.e. the original contents are preserved, and new text is appended to the end of the file.
0402	Open for appending to the end of the file and for reading. The file must already exist. The previous contents are preserved, and new text is appended to the end of the file. For KR functionality (only available with C/C++ versions lower than V3), existing files are positioned to end of file when opened; for ANSI functionality, they are positioned to the beginning of the file. □

#### Return value

Non-negative integer

indicating the the lowest numbered unused file descriptor, if successful.

-1 if an error occurs. No file is created or updated. `errno` is set to indicate the error.

#### Errors

`open()` and `open64()` fail if:

`EACCES` Search permission is denied on a component of the path.

The file does not exist, and the access permissions specified by *oflag* are denied.

The file does not exist, and write permission is denied by the parent directory of the file to be created.

`O_TRUNC` is set, and write permission is denied for the file.

*Extension*

EAGAIN	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see <code>chmod()</code> ).
EEXIST	<code>O_CREAT</code> and <code>O_EXCL</code> are set, and the named file exists.
EFAULT	<i>path</i> points beyond the assigned address space of the process.
EINTR	A signal was caught during the <code>open()</code> system call.
EINVAL	The value of the <i>oflag</i> argument is invalid.
EIO	A connection was cleared or an error occurred while opening a stream-oriented device.
EISDIR	The named file is a directory and <i>oflag</i> includes <code>O_WRONLY</code> or <code>O_RDWR</code> .
EMFILE	{ <code>OPEN_MAX</code> } file descriptors are currently open in the calling process.
EMULTIHOP	Components of <i>path</i> require hops to several remote computers, but the file system does not permit this.
ENAMETOOLONG	The length of the <i>path</i> argument exceeds { <code>PATH_MAX</code> } or a pathname component is longer than { <code>NAME_MAX</code> }.
ENFILE	The maximum allowable number of files is currently open in the system.
ENOENT	<code>O_CREAT</code> is not set and the named file does not exist, or <code>O_CREAT</code> is set and either the path prefix does not exist or <i>path</i> points to an empty string.
ENOLINK	<i>path</i> refers to a remote computer to which there is no active connection.
ENOSPC	The file does not exist, and <code>O_CREAT</code> is specified or the directory or file system that would contain the new file cannot be expanded.
ENOSR	A stream cannot be assigned.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist, or <code>O_NONBLOCK</code> is set, the named file is a FIFO, <code>O_WRONLY</code> is set and no process has the file open for reading.
EROFS	The named file resides on a read-only file system and either <code>O_WRONLY</code> , <code>O_RDWR</code> , <code>O_CREAT</code> (if file does not exist) or <code>O_TRUNC</code> is set in the <i>oflag</i> argument.
EOVERFLOW	<code>O_LARGEFILE</code> is not set for a file and the size of the file cannot be represented correctly in an object of type <code>off_t</code> .

Notes The program environment determines whether `open()` is executed for a BS2000 or POSIX file.

### *BS2000*

The BS2000 file name or link name can be written in both uppercase and lowercase. It is automatically converted to uppercase.

Non-existent files are created by default with the following attributes:

for KR functionality (only available with C/C++ versions lower than V3), as a SAM file with variable record length and standard block length;

for ANSI functionality, as an ISAM file with variable record length and standard block length.

SAM files are always opened as text files by `open()`.

If a link name is used, the following file attributes may be changed with the `SET-FILE-LINK` command: the access method, record length, record format, block length and block format. When the old contents of an existing file are deleted (0003, 01001), the catalog attributes of the file are preserved.

Location of the file position indicator in append mode:

If the file-position indicator of a file opened in append mode (0401, 0402) has been explicitly moved from the end of the file (`lseek()`), it is handled differently for KR and ANSI functionality as described below. KR functionality (only available with C/C++ versions lower than V3): the current file-position indicator is ignored only when writing with the elementary function `write()`, and the file is automatically positioned to end of the file.

ANSI functionality: the current file-position indicator is ignored for all write functions, and the file is automatically positioned to end of the file.

An attempt to open a non-existent file in the read (0000, 0002), update (0001), or append (0401, 0402) mode will result in an error. A file may be opened for different access modes simultaneously, provided these modes are mutually compatible within the BS2000 data management system. (INCORE) files can only be opened for writing (01001) or for writing and reading (0003). Data must first be written. To read in the written data again, the file must be positioned to beginning of file with the `lseek()` function.

When a program starts, the standard files for input, output, and error output are automatically opened with the following file descriptors:

```
stdin:    0
stdout:   1
stderr:   2
```

A maximum of `_NFILE` files may be open simultaneously. `_NFILE` is defined as 2048 in `stdio.h`.

See also `chmod()`, `close()`, `creat()`, `creat64()`, `dup()`, `fcntl()`, `fdopen()`, `lseek()`, `lseek64()`, `read()`, `umask()`, `write()`, `fcntl.h`, `sys/types.h`, `sys/stat.h`.

## opendir - open directory

Syntax `#include <dirent.h>`

*Optional*

`#include <sys/types.h>` □

`DIR *opendir(const char *dirname);`

**Description** `opendir()` opens a directory stream corresponding to the directory named by *dirname*. The directory stream is positioned at the first entry. The type `DIR`, which is defined in `dirent.h`, represents a directory stream that is an ordered sequence of all directory entries in a special directory. Since the type `DIR` is implemented in POSIX using a file descriptor, applications can only open a maximum of `{OPEN_MAX}` files and directories.

If *dirname* cannot be accessed or is not a directory, or if not enough memory to hold a `DIR` structure or a buffer for the directory entries can be allocated with `malloc()`, a null pointer is returned.

### Return value

Pointer to a `DIR` object  
if successful.

Null pointer if an error occurs. `errno` is set to indicate the error.

### Errors

`opendir()` will fail if:

`EACCES` Search permission is denied for the component of *dirname* or read permission is denied for *dirname*.

*Extension*

`EFAULT` *dirname* points outside the allocated address space of the process.

`ELOOP` Too many symbolic links were encountered in resolving *dirname*. □

`EMFILE` More than `{OPEN_MAX}` file descriptors are currently open in the calling process.

`ENAMETOOLONG`

The length of the *dirname* argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}`.

`ENFILE` Too many file descriptors are currently open in the system.

`ENOENT` *dirname* points to the name of a file that does not exist or to an empty string.

`ENOTDIR` A component of *dirname* is not a directory.

**Notes** `opendir()` should be used in conjunction with `readdir()`, `closedir()` and `rewinddir()` to examine the contents of the directory (see also `readdir()`). This method is recommended for portability.

`opendir()` is executed only for POSIX files

**See also** `closedir()`, `readdir()`, `rewinddir()`, `dirent.h`, `sys/types.h`, `limits.h`.

## openlog - system logging

**Syntax** `#include <syslog.h>`  
`void openlog(const char *ident, int logopt, int facility);`

**Description** See `closelog()`.

## optarg, opterr, optind, optopt - variables for command options

**Syntax** `#include <unistd.h>`  
`extern char *optarg;`  
`extern int optind, opterr, optopt;`

**Description** See `getopt()`.



## pathconf, fpathconf - get value of pathname variable

Syntax `#include <unistd.h>`

```
long int pathconf(const char *path, int name);
long int fpathconf(int fildes, int name);
```

Description `pathconf()` and `fpathconf()` provide a method of determining the current value of a configurable system variable *name* that is associated with a file or directory.

For `pathconf()`, *path* points to the pathname of a file or directory.

For `fpathconf()`, *fildes* is an open file descriptor.

The C runtime system supports the variables listed in the following table. Other X/Open-conformant implementations may support additional variables. The table below contains the system variables from the headers `limits.h` or `unistd.h` that can be queried with `pathconf()` or `fpathconf()`; the symbolic constants, which are defined in `unistd.h`, are the corresponding values used for the *name* argument:

System variable	Value of <i>name</i> (constant)	Notes
{LINK_MAX}	_PC_LINK_MAX	1.
{MAX_CANON}	_PC_MAX_CANON	2.
{MAX_INPUT}	_PC_MAX_INPUT	2.
{NAME_MAX}	_PC_NAME_MAX	3., 4.
{PATH_MAX}	_PC_PATH_MAX	4., 5.
{PIPE_BUF}	_PC_PIPE_BUF	6.
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7.
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3., 4.
_POSIX_VDISABLE	_PC_VDISABLE	2.

1. If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2. If *path* or *fildes* does not refer to a special file for a terminal, the {MAX\_CANON}, {MAX\_INPUT} and \_POSIX\_VDISABLE variables are ignored.
3. If *path* or *fildes* refers to a directory, the value returned applies to file names within the directory.
4. If *path* or *fildes* does not refer to a directory, no association of the variables {NAME\_MAX}, {PATH\_MAX} and \_POSIX\_VDISABLE with the specified file is supported.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.

6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the referenced object. If *path* or *fildes* refers to a directory, the value returned applies to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, the behavior is undefined.
7. If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, which are defined in this standard and exist or can be created within the directory.

## Return value

Current value of *name*

if successful.

The value returned will not be more restrictive than the corresponding value available to the application when it was compiled with the implementation's `limits.h` or `unistd.h`.

-1 if the variable corresponding to *name* has no limit for the *path* or file descriptor. `errno` is not set.

-1 if *name* has an invalid value, or if the implementation needs to use *path* or *fildes* to determine the value of *name*, and the implementation does not support the association of *name* with the file specified by *path* or *fildes*, or if the process did not have appropriate privileges to query the file specified by *path* or *fildes*, or if *path* does not exist, or if *fildes* is not a valid file descriptor.

In these cases, `errno` is set to indicate the error.

## Errors

`pathconf()` will fail if:

*Extension*

EACCES Search permission is denied for a component of the pathname.  $\square$

EINVAL The value of *name* is not valid, or an attempt was made to access a BS2000 file.

*Extension*

ELOOP Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG The length of the *path* argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}` and `_POSIX_NO_TRUNC` is set.

ENOENT The named file does not exist or *path* points to an empty string.  $\square$

ENOTDIR        A component of the path prefix is not a directory.

fpathconf() will fail if:

EINVAL        The value of *name* is not valid, or  
the implementation does not support an association of the variable *name*  
with the specified file.

EBADF        *fdes* is not a valid file descriptor.

**See also**    sysconf(), limits.h, unistd.h.

## pause - suspend process until signal is received

**Syntax**      `#include <unistd.h>`  
`int pause(void);`

**Description** `pause()` suspends the calling process until delivery of a signal whose action is either to execute a signal-handling function or to terminate the process.

If the action is to terminate the process, `pause()` will not return.

If the action is to execute a signal-handling function, `pause()` will return after the signal-handling function returns.

If threads are used, then the function affects the process or a thread in the following manner: Suspends the thread until it receives a signal.

**Return value**  
-1                      if an error occurs. `errno` is set to indicate the error.

Since `pause()` suspends process execution indefinitely unless interrupted by a signal, there is no successful completion return value.

**Errors**            `pause()` will fail if:

`EINTR`              A signal is caught by the calling process and control is returned from the signal-handling function.

**See also**        `sigsuspend()`, `sleep()`, `unistd.h`.

## pclose - close pipe stream

**Syntax**        `#include <stdio.h>`  
`int pclose(FILE *stream);`

**Description** `pclose()` closes the named *stream* that was opened by `popen()`, waits for the command started by `popen()` to terminate, and returns its exit status. However, if the exit status is unavailable to `pclose()`, then `pclose()` returns -1 and sets `errno` to `ECHILD` to report the situation. This may occur if the application has already obtained the exit status by one of the following functions:

- `wait()`
- `waitpid()` with a `pid` argument less than or equal to 0 or equal to the process ID of the command interpreter.

In any case, `pclose()` will not return before the child process created by `popen()` has terminated.

If the command interpreter cannot be executed, the child exit status returned by `pclose()` will be the same as if the command interpreter had terminated using `exit(127)` or `_exit(127)`.

**Return value**  
Exit status of the command interpreter  
if successful.  
-1                if *stream* was not created by `popen()`.

**Errors**        `pclose()` will fail if:  
`ECHILD`        The exit status of the child process could not be determined.  
*Extension*  
`EINVAL`        An attempt was made to access a BS2000 file. □

**Notes**        `pclose()` is executed only for POSIX files.

**See also**       `fork()`, `popen()`, `wait()`, `waitpid()`, `stdio.h`.

## perror - write error messages to standard error

**Syntax**      `#include <stdio.h>`  
`void perror(const char *s);`

**Description** `perror()` maps the error code in the external variable `errno` to a language-dependent *error\_message*, which is written to the standard error stream as follows:

*s* : *error\_message* \n

*s* is a string that should include at least the name of the program in which the error occurred. If *s* is a null pointer or the character to which *s* points is a null byte, the message portion is omitted ("*s* : ").

The contents of the error message strings depend on the environment variable `LANG`. All error codes and error messages are listed and explained in the header `errno.h`.

`perror()` will mark the file associated with the standard error stream as having been written (`st_ctime`, `st_mtime` marked for update) at some time between its successful completion and a call to `exit()`, `abort()`, or the completion of `fflush()` or `fclose()` on `stderr`.

**Notes**      The contents of the area in which the error code and error text are stored are not explicitly deleted, i.e. the previous contents are retained until they are overwritten with appropriate information when a fresh error occurs. `perror` calls are therefore only useful after a function has provided an error return value.

The program environment determines whether `perror()` is executed for a BS2000 or POSIX file.

### *BS2000*

In the case of I/O errors or when system commands are executed, *error\_message* contains the appropriate DMS error codes as supplementary information.

In KR mode (only available with C/C++ versions lower than V3), a value of type `char *` is returned. It contains a pointer to an internal C buffer with the error message. The contents are overwritten at each new call to `perror` (see also the manual "C Library Functions (BS2000/OSD)").

If the program is called in a BS2000 environment and the file does not exist, the following error message is printed on standard output:

```
Program fopen: dataset not found (cmd: OPEN), errorcode=DD33
```

DD33 is the DMS error code. □

**See also**      `strerror()`, `errno.h`, `stdio.h`, section "Selecting functionality" on page 40.

## pipe - create pipe

**Syntax**        `#include <unistd.h>`

```
int pipe(int filides[2]);
```

**Description** `pipe()` creates a pipe and places two file descriptors, which refer to the open file descriptions for the read and write ends of the pipe, into the arguments `filides[0]` and `filides[1]`. These integer values are the two lowest available at the time of the `pipe()` call. The `O_NONBLOCK` bit is not set for either of the two file descriptors (the `fcntl()` function can be used to set the `O_NONBLOCK` bit).

Data can then be written to the file descriptor `filides[1]` and read from file descriptor `filides[0]`. A read on the file descriptor `filides[0]` accesses the data written to file descriptor `filides[1]` on a first-in-first-out basis.

A process has the pipe open for reading if it has a file descriptor open that refers to the read end of the pipe, i.e. `filides[0]`; the same applies to writing and the write end, i.e. `filides[1]`.

Upon successful completion, `pipe()` will mark the `stat` structure components of the pipe, i.e. `st_atime`, `st_ctime` and `st_mtime`, for update.

The `FD_CLOEXEC` bit is not set for either of the two file descriptors.

**Return value**

0                if successful.

-1               if an error occurs. `errno` is set to indicate the error.

**Errors**        `pipe()` will fail if:

`EMFILE`        `{OPEN_MAX}` minus 2 file descriptors are already open for this process.

`ENFILE`        The number of simultaneously open files in the system would exceed a system-imposed limit.

**Notes**        `pipe()` is executed only for POSIX files.

**See also**     `fcntl()`, `read()`, `write()`, `unistd.h`.

## poll - multiplex STREAMs I/O

Syntax `#include <poll.h>`

```
int poll(struct pollfd fds[], nfd_t nfds, int timeout);
```

Description `poll()` provides applications with a mechanism for multiplexing input/output over a set of open file descriptors.

For each field element to which *fds* points, `poll()` checks whether one or more of the events listed in *events* has occurred for the corresponding file descriptor. The number of `pollfd` structures in the *fds* field is specified by the value *nfds*. `poll()` identifies the file descriptors which the application can read from or write to, or for which events have occurred.

*fds* defines the file descriptors to be checked as well as the events that are to be polled for the respective file descriptors. *fds* is a pointer to a field with one element each for every file descriptor to be checked. The elements of the field are `pollfd` structures, which contain the following:

```
int fd;           /* Open file descriptor */
short events;    /* Events to be queried */
short revents;   /* Events that have occurred */
```

*fd* identifies an open file descriptor, *events* and *revents* are bit masks which are formed from the following flags through bitwise ORing (any combinations are possible):

POLLIN	Data which does not have the highest priority can be read without blocking. For STREAMS this flag is set in <i>revents</i> even if the message has the length 0.
POLLRDNORM	Normal data (priority = 0) can be read without blocking. For STREAMS this flag is set in <i>revents</i> even if the message has the length 0.
POLLRDBAND	Data with priority $\neq$ 0 can be read without blocking. For STREAMS this flag is set in <i>revents</i> even if the message has the length 0.
POLLPRI	Data with the highest priority can be received without blocking. For STREAMS this flag is set in <i>revents</i> even if the message has the length 0.
POLLOUT	Normal data (priority = 0) can be written without blocking.
POLLWRNORM	As POLLOUT.
POLLWRBAND	Data with priority $\neq$ 0 can be written.
POLLMSG	An <code>M_SIG</code> or <code>M_PCSIG</code> message containing an <code>ASIGPOLL</code> signal has arrived at the beginning of the stream head queue.
POLLERR	An error has occurred for the STREAM or the special file. This flag is only valid in the <i>revents</i> bit mask; in the <i>events</i> bit mask it is ignored.



- POLLHUP** A hang-up has occurred in the **STREAM** (the connection to the device has been interrupted). **POLLHUP** and **POLLOUT** mutually exclude each other; data can never be written to a stream if a hang-up has occurred. However, the event and **POLLIN** or **POLLRDNORM**, **POLLRDBAND** or **POLLPRI** do not mutually exclude each other.  
The **POLLHUP** flag is only valid in the **revents** bit mask; in the **events** bit mask it is ignored.
- POLLNVAL** The specified *fd* value is invalid. This flag is only valid in the **revents** bit mask; in the **events** bit mask it is ignored.

If the value in **fd** is less than zero, **events** is ignored, and **revents** is set to 0 for this field entry when **poll()** returns.

The results of the **poll()** query are displayed in the **revents** field in the **pollfd** structure. **poll()** first sets all bits in **revents** to zero. If one or more of the events queried in **events** has occurred, **poll()** sets the corresponding bits in **revents**. The bits for **POLLHUP**, **POLLERR** and **POLLNVAL** are automatically set in **revents** when the corresponding events occur; they do not need to be set in **events**.

If the check reveals that none of the events queried for the file descriptors has occurred, **poll()** waits at least *timeout* milliseconds for an event to occur for one of the specified file descriptors. On a machine which does not offer precision in milliseconds, *timeout* is rounded up to the next permissible value available in this system.

If the value of *timeout* is 0, **poll()** returns immediately. If *timeout* has the value -1, **poll()** waits until one of the queried events occurs, or until the call is interrupted (blocking **poll()** call).

**poll()** is not affected by the **O\_NDELAY** and **O\_NONBLOCK** flags.

**poll()** supports text files, terminals, pseudoterminals, **STREAMS**-based files, **FIFO** files and pipes, sockets and **XTI**.

With text files, **poll()** always returns a **TRUE** for reading and writing.

#### Return value

- Value  $\geq$  0** if successful.  
A positive value indicates the total number of file descriptors for which the **revents** field is not equal to zero.  
0 means that the time for the call has expired and there are no file descriptors for which the **revents** field is not equal to zero.
- 1** if an error occurs. **errno** is set to indicate the error.

Errors	poll() will fail if:	
EAGAIN		The allocation of the internal data structures has failed but could succeed if repeated.
EFAULT		An argument points to a storage space outside the allocated address space.
EINTR		A signal was caught during the poll() system call.
EINVAL		The nfd <sub>s</sub> argument is less than zero or greater than OPEN_MAX, or one of the fd entries refers to a STREAM or multiplexer which is connected downstream via a multiplexer.

**See also** getmsg(), putmsg(), read(), select(), write(), poll.h, stropts.h.

## popen - initiate pipe stream to or from process

Syntax `#include <stdio.h>`

`FILE *popen (const char *command, const char *mode);`

Description `popen()` executes the command specified by the string *command*, creates a pipe between the calling program and the executed command, and returns a pointer to a stream that can be used to either read from (I/O mode *r*) or write to (I/O mode *w*) the pipe.

The environment of the executed command in an XPG4-conformant implementation will be as if a child process were created within the `popen()` call using `fork()`, and the child invoked the `sh` utility using the call:

```
exec1 (shell_path, "sh", "-c", command, (char *)0);
```

where *shell\_path* is an unspecified name for the `sh` utility.

`popen()` ensures that any streams from previous `popen` calls that remain open in the parent process are closed in the new child process.

*mode* is a string that specifies I/O mode:

1. If *mode* is *r* when the child process is started, the standard output of the command will be redirected to the pipe. The file descriptor `STDOUT_FILENO` will be the writable end of the pipe, and the file descriptor `fileno(stream)`, where *stream* is the stream pointer returned by `popen()`, will be the readable end of the pipe.
2. If *mode* is *w* when the child process is started, the standard output of the command will be redirected to the pipe. The file descriptor `STDIN_FILENO` will be the readable end of the pipe, and the file descriptor `fileno(stream)`, where *stream* is the stream pointer returned by `popen()`, will be the writable end of the pipe.

After `popen()`, both the parent and the child process will be capable of executing independently before either terminates.

Return value

Pointer to a stream  
if successful.

Null pointer if files or processes cannot be created.

Notes

If the parent process and the process created by `popen()` read or write a file simultaneously, neither of the processes may use buffered I/O. Problems with an output filter can be avoided by taking the precaution of flushing the buffers, e.g. with `flush()` (see also `fclose()`).

`popen()` is executed only for POSIX files.

See also

`pclose()`, `pipe()`, `sysconf()`, `system()`, `stdio.h`, the `sh` command in the manual "POSIX Commands (BS2000/OSD)".

## pow - power function

**Syntax**      `#include <math.h>`  
`double pow(double x, double y);`

**Description** `pow()` computes the value of  $x^y$ .  
*x* is the base of the exponential function (a floating-point number).  
*y* is the exponent (also a floating-point number).  
If *x* is 0, *y* must be positive;  
if *x* is negative, *y* must be an integer value.

**Return value**

Value of $x^y$	if <i>x</i> , <i>y</i> and the result lie in the permitted floating-point interval.
+/-HUGE_VAL	(depending on the sign) if an overflow occurs; <code>errno</code> is set to indicate the error.
1.0	if <i>x</i> and <i>y</i> are both 0.
-HUGE_VAL	if <i>x</i> is 0 and <i>y</i> is less than 0; <code>errno</code> is set to indicate the error.
undefined	if <i>x</i> is less than 0 and <i>y</i> is not an integer; <code>errno</code> is set to indicate the error.

**Errors**      `pow()` will fail if:

EDOM	The value of <i>x</i> is negative, and <i>y</i> is not an integer. The value of <i>x</i> is 0, and <i>y</i> is negative.
ERANGE	The value of <i>x</i> would cause an overflow.

**See also**      `exp()`, `hypot()`, `log()`, `log10()`, `sinh()`, `sqrt()`, `math.h`.

## printf - write formatted output on standard output stream

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

**Description** See `fprintf()`.

## ptsname - name of pseudoterminal

**Syntax**      `#include <stdlib.h>`  
`char *ptsname(int fildes);`

**Description** The `ptsname()` function returns the name of the slave pseudoterminal that is assigned to the master pseudoterminal. *fildes* is the file descriptor that references the master terminal. `ptsname()` returns a pointer to a string containing the pathname of the corresponding slave terminal. The name is terminated with the null byte.

The name has the format `/dev/pts/N`, where *N* is an integer between 0 and 255.

`ptsname()` is not thread-safe.

### Return value

Pointer to a string  
if successful. The string contains the name of the slave terminal.

Null pointer  
if an error occurs. This can happen if *fildes* is not a valid file descriptor or if the name of the slave terminal does not exist in the file system.

**Notes**      The pointer points to a static data area that is overwritten every time `ptsname()` is called.

**See also**    `grantpt()`, `ttyname()`, `unlockpt()`, `stdlib.h`.

## putc, putc\_unlocked - put byte on stream

Syntax `#include <stdio.h>`

```
int putc(int c, FILE *stream);
```

```
int putc_unlocked(int c, FILE *stream);
```

Description The function `putc()` is equivalent to `fputc()`, but is defined as a macro and a function. When it is used as a macro, it may evaluate *c* and *stream* more than once, so these argument should never be an expression with side-effects.

The function `putc_unlocked()` (see page 398 under `getc_unlocked( ) ...`) is functionally equivalent to `putc()` except that its implementation is not thread-safe. For this reason it can only be used safely in a multithreaded program if the thread that calls it owns the corresponding (FILE \*) object. This is the case after successfully calling the `flockfile()` or `ftrylockfile()` functions.

Return value

See `fputc()`.

Errors

See `fputc()`.

Notes

If `putc()` is used as a macro, it may handle the arguments *c* and *stream* with side-effects incorrectly. `putc(c, *f++)`, in particular, will usually not work correctly. It is therefore advisable to use `fputc()` instead.

The bytes are not written immediately to the external file but are stored in an internal C buffer (see section "Buffering streams" on page 77).

The program environment determines whether `putc()` is executed for a BS2000 or POSIX file.

*BS2000*

Control characters for white space (`\n`, `\t`, etc.) are converted to their appropriate effect when output to text files, depending on the type of text file (see section "White-space characters" on page 84). □

See also

`fputc()`, `getc_unlocked()`, `stdio.h`.

## putchar - put byte on standard output stream (thread-safe)

Syntax `#include <stdio.h>`

```
int putchar(int c);
```

```
int putchar_unlocked(int c);
```

Description The function call *putchar(c)* is equivalent to *putc(c, stdout)*. *putchar()* is implemented both as a function and as a macro.

The function *putchar\_unlocked()* (see page 398 under *getc\_unlocked( ) ...*) is functionally equivalent to *putchar()* except that its implementation is not thread-safe. For this reason it can only be used safely in a multithreaded program if the thread that calls it owns the corresponding (FILE \*) object. This is the case after successfully calling the *flockfile()* or *ftrylockfile()* functions.

Return value

See *fputc()*.

Notes The bytes are not written immediately to the external file but are stored in an internal C buffer (see section "Buffering streams" on page 77).

For further information on output to text files and on converting control characters for white space (`\n`, `\t`, etc.), see section "White-space characters" on page 84.

The program environment determines whether *putchar()* is executed for a BS2000 or POSIX file.

See also *getchar()*, *getchar\_unlocked()*, *putc()*, *putc\_unlocked()*, *stdio.h*.

## putchar\_unlocked - put byte on standard output stream (thread-safe)

Syntax `#include <stdio.h>`

```
int putchar_unlocked(int c);
```

Description see *getc\_unlocked()*.

## putenv - change or add environment variables

Syntax `#include <stdlib.h>`

```
int putenv (const char *string);
```

Description `putenv()` may be used to alter the value of an existing environment variable or to define a new one. *string* must point to a string of the form "*name=value*", where *name* is the name of an environment variable, and *value* is the value assigned to it. If *name* is identical to an existing environment variable, the associated value of the existing variable is updated with the new *value*. If *name* is a new environment variable, it is added to the environment. In either case, *string* becomes part of the environment and thus alters it.

The space occupied by *string* is no longer used when a new value is assigned to an existing environment variable with `putenv()`.

`putenv()` is not thread-safe.

Return value

0 if successful.

≠ 0 if an error occurs, e.g. because not enough memory is available. `errno` is set to indicate the error.

Errors `putenv()` will fail if:

ENOMEM There is not enough memory available.

Notes `putenv()` manipulates the environment to which `environ` points and may be used in connection with `getenv()`.

`putenv()` may use `malloc()` to expand the environment.

A potential source of error is to call `putenv()` with an automatic variable as an argument and then return from the calling function while *string* is still part of the environment.

*BS2000*

When a program is started from the POSIX shell, the SDF-P variable structure `SYSPOSIX` is evaluated as part of the environment definition in addition to the defaults for the environment (see also `environ`). `putenv()` does not, however, alter the SDF-P variables. The POSIX environment corresponds to the one pointed to be `environ`. `SYSPOSIX.name` is defined in BS2000, i.e. outside the POSIX subsystem. □

See also `environ`, `exec`, `getenv()`, `malloc()`, `stdlib.h`, section "Environment variables" on page 71.



## putmsg, putpmsg - send message to STREAMS file

Syntax `#include <stropts.h>`

```
int putmsg(int fildev, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flags);

int putpmsg(int fildev, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flags);
```

Description `putmsg()` creates a message from the specified buffers and sends it to a STREAMS file. The message can contain either a data section, a control section or both. The data and control sections to be sent are distinguished from each other by being written to different buffers (see below). The semantics of the sections is defined via the STREAMS module that receives the message.

The `putpmsg()` function has the same functionality as `putmsg()`, but it allows the user to send messages with different priorities.

All information described here for `putmsg()` also applies to `putpmsg()`, with exceptions being explicitly indicated.

*fildev* is a file descriptor that references an open stream. *ctlptr* and *dataptr* each point to an `strbuf` structure containing the following elements:

```
int maxlen;    /* Not used */
int len;       /* Length of data */
void *buf;     /* Pointer to buffer for data */
```

*ctlptr* points to the structure that describes the control section to be included in the message (if there is one). The `buf` field in the `strbuf` structure points to the buffer containing the control information, and the `len` field specifies the number of bytes to be sent. The `maxlen` field is not used in `putmsg()` (see `getmsg()`). In the same way, *dataptr* describes the data section which is to be included in the message. *flags* indicates what type of message is to be sent (see below).

For the data section of a message to be sent, *dataptr* must not be the same as the null pointer, and the `len` field of *dataptr* must contain a value  $\geq 0$ . For the control section of a message to be sent, the corresponding values must be set for *ctlptr*. A data (control) section is not sent if either *dataptr* (*ctlptr*) is the null pointer or the corresponding `len` field is set to -1.

If a control section is specified in `putmsg()`, and *flags* is set to `RS_HIPRI`, a message with high priority is sent.

If no control section is specified and *flags* is set to `RS_HIPRI`, `putmsg()` fails and sets `errno` to `EINVAL`.

If *flags* is set to 0, a normal message is sent (priority=0).

If neither a control section nor a data section is specified and *flags* is set to 0, no message is sent and the value 0 is returned.

The STREAM head guarantees that the control section of a message generated by `putmsg()` is at least 64 bytes long.

Other flags are used for `putpmsg()`: *flags* is a bit mask which contains either `MSG_HIPRI` or `MSG_BAND` or 0 (the values mutually exclude each other).

If *flags* is set to 0, `putpmsg()` fails and sets `errno` to `EINVAL`.

If a control section is specified and *flags* is set to `MSG_HIPRI` and *band* is set to 0, a message with a high priority is sent.

If *flags* is set to `MSG_HIPRI`, and either no control section is specified or *band*  $\neq$  0, `putpmsg()` fails and sets `errno` to `EINVAL`.

If *flags* is set to `MSG_BAND`, a message in the priority class specified by *band* is sent.

If no control section and no data section are specified, and *flags* is set to `MSG_BAND`, no message is sent and 0 is returned.

Normally, `putmsg()` blocks if the read/write queue of the stream is full because of internal control flow conditions. With high-priority messages, however, `putmsg()` does not block in this case.

With other messages, `putmsg()` does not block if the read/write queue is full if `O_NDELAY` or `O_NONBLOCK` is set. Instead, the call fails and `errno` is set to `EAGAIN`.

`putmsg()` or `putpmsg()` block regardless of the priority and `O_NDELAY` or `O_NONBLOCK` even if they are waiting for the availability of message blocks in the stream. Partial messages are not sent.

#### Return value

- |    |  |
|----|--|
| 0  | if successful.   |
| -1 | if an error occurs. <code>errno</code> is set to indicate the error. |

#### Errors

`putmsg()` and `putpmsg()` will fail if:

- |        |   |
|--------|---|
| EAGAIN | A message without priority was specified, the <code>O_NDELAY</code> or <code>O_NONBLOCK</code> flag is set, and the read/write queue of the STREAM is full because of internal control flow conditions<br>or<br>no buffer could be allocated for the message to be created. |
| EBADF  | <i>fildev</i> is not a valid file descriptor open for writing.  |
| EINTR  | A signal was caught during the <code>putmsg()</code> system call.   |
| EFAULT | <i>ctlptr</i> or <i>dataptr</i> point outside the allocated address space.  |

EINVAL	<p>An undefined value was specified in <i>flags</i>, or <i>flags</i> is set to <code>RS_HIPRI</code> or <code>MSG_HIPRI</code> and no control section was provided, or</p> <p>the STREAM or multiplexer referenced by <i>fildev</i> is connected downstream via a multiplexer.</p> <p>For <code>putpmsg()</code> only: <i>flags</i> is set to <code>MSG_HIPRI</code> and <i>band</i> <math>\neq</math> 0.</p>
ENOSR	No buffer could be allocated for the message to be created because there was not enough STREAMS storage space.
ENOSTR	No STREAM belongs to <i>fildev</i> .
ENXIO	A hang-up was generated downstream for the specified stream.
EPIPE or EIO	<i>fildev</i> references a STREAM-based pipe and the other end of the pipe is closed. The SIGPIPE signal is generated for the calling process.
ERANGE	<p>The data section of the message has a size that is not within the range defined by the maximum and minimum packet size of the highest stream module.</p> <p>ERANGE is also returned if the control section of the message is larger than the configured maximum size of the control section of a message, or if the data section of a message is larger than the configured maximum size of the data section of a message.</p>

`putmsg()` and `putpmsg()` also fail if an asynchronous STREAMS error message has reached the stream head before the `putmsg()` or `putpmsg()` call. In this case, `errno` refers to the error contained in the STREAMS error message.

**Notes** If two processes open a FIFO file, with one writing a high-priority message with `putmsg()` and the other reading a high-priority message with `getmsg()`, messages can be lost. This loss can be avoided by slowing down the send process with `sleep` between the individual `putmsg()` calls.

**See also** `getmsg()`, `poll()`, `read()`, `write()`, `stropts.h`.

## putpwent - enter user into user catalog *(extension)*

Syntax `#include <pwd.h>`

```
int putpwent(const struct passwd *p, FILE *f);
```

Description `putpwent()` writes the user data from the password structure `p` into the user catalog serially. The calling process must have appropriate privileges.

`p` is a password structure that was obtained with `getpwent()`, `getpwuid()` or `getpwnam()` and then modified.

`f` is supported only for compatibility reasons; it is not evaluated.

Return value

0 if successful.

≠0 if an error occurs. `errno` is set to indicate the error.

Errors `putpwent()` will fail if:

EINVAL The user data is invalid.

EFAULT The specified address of the `passwd` structure is invalid.

ENOENT The user is not recognized.

EPERM The calling process does not have appropriate privileges.

Notes There is no `/etc/passwd` password file in the POSIX subsystem. User data is stored internally in the user catalog (see also the manual "POSIX Basics (BS2000/OSD)").

See also `getpwent()` and the manual "POSIX Basics (BS2000/OSD)".

## puts - put string on standard output

**Syntax**      `#include <stdio.h>`  
`int puts(const char *s);`

**Description** `puts()` writes the string pointed to by `s`, followed by a newline character, to the standard output stream `stdout`. The terminating null byte is not written.

The structure components `st_ctime` and `st_mtime` of the file are marked for changing between successful execution of `puts()` and the next successful completion of a call to `fflush()` or `fclose()` for the same data stream or a call to `exit()` or `abort()` (see `sys/stat.h`).

**Return value**

Non-negative number  
if successful.

EOF  
if an error occurs. The error indicator for the stream is set, and `errno` is set to indicate the error.

**Errors**      See `fputc()`.

**Notes**      The `puts()` function appends a newline character, while `fputs()` does not.

The terminating null byte of `s` is not output.

The program environment determines whether `puts()` is executed for a BS2000 or POSIX file.

For further information on output to text files and on converting control characters for white space (`\n`, `\t`, etc.), see section "White-space characters" on page 84.

**See also**      `fputs()`, `fopen()`, `putc()`, `stdio`, `stdio.h`.

## pututxline - write utmpx entry

**Syntax**      `#include <utmpx.h>`  
                 `struct utmpx *pututxline (const struct utmpx *utmpx);`

**Description** See `endutxent()`.

**Return value**  
                 Pointer to a *utmpx* structure containing a copy of the added utmpx entry  
                 if successful.  
  
                 Null pointer      if an error occurs. `errno` is not set.

**Notes**        To be able to call `pututxline()`, the process must have the appropriate access permissions.

**See also**     `utmpx.h`.

## putw - put word on stream

**Syntax**      `#include <stdio.h>`  
`int putw(int w, FILE *stream);`

**Description** `putw()` writes the word *w* to the output stream *stream* at the position at which the file offset, if defined, is pointing. The size of a word corresponds to the size of a type `int` and varies from machine to machine. In the C runtime system, the size of a type `int` is 4 bytes. `putw()` neither assumes nor causes special alignment in the file.

The structure components `st_ctime` and `st_mtime` of the file are marked for changing between successful execution of `putw()` and the next successful completion of a call to `fflush()` or `fclose()` for the same data stream or a call to `exit()` or `abort()` (see `sys/stat.h`).

`putw()` is not thread-safe.

**Return value**

0	if successful.
≠ 0	if an error occurs. The error indicator for the stream is set, and <code>errno</code> is set to indicate the error.

*BS2000*

EOF	if an error occurs. □
-----	-----------------------

**Errors**      See `fputc()`.

**Notes**      Due to possible differences in word length and byte ordering, files written using `putw()` are machine-dependent, and may not be readable using `getw()` on a different processor.

Since `putw()` does not indicate errors explicitly (-1 is a valid integer value), it is advisable to also use `ferror()` to verify whether an error occurred before or after writing.

The bytes are not written immediately to the external file but are stored in an internal C buffer (see section "Buffering streams" on page 77).

Control characters for white space (`\n`, `\t`, etc.) are converted to their appropriate effect when output to text files, depending on the type of text file (see section "White-space characters" on page 84).

The program environment determines whether `putw()` is executed for a BS2000 or POSIX file.

**See also**      `fopen()`, `fputc()`, `fwrite()`, `getw()`, `stdio.h`, `sys/stat.h`.

## putwc - put wide character on stream

Syntax `#include <wchar.h>`

*Optional*

`#include <stdio.h>` □

`wint_t putwc(wint_t wc, FILE *stream);`

Description `putwc()` is equivalent to the `fputwc()` function, except that if it is implemented as a macro it may evaluate *stream* more than once, so the *stream* argument should never be an expression with side-effects.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

Return value

See `fputwc()`.

Errors

See `fputwc()`.

Notes

`putwc(wc, *f++)` may not work as expected. Therefore, use of this function is not recommended; `fputwc()` should be used instead.

See also

`putwc()`, `stdio.h`, `wchar.h`.



## putwchar - put wide character on standard output stream

**Syntax**        `#include <wchar.h>`  
                 `wint_t putwchar(wint_t wc);`

**Description** The function call `putwchar(wc)` is equivalent to `putwc(wc, stdout)`.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**Return value**  
See `putwc()`.

**See also**        `fputwc()`, `putwc()`, `wchar.h`.

## qsort - sort table of data

Syntax `#include <stdlib.h>`

```
void qsort (void * base, size_t nel, size_t width, int ( *compar) (const void *, const void *));
```

Description The `qsort()` function is an implementation of the quicksort algorithm. It sorts a table of data in place. The contents of the table are sorted in ascending order according to the user-supplied comparison function. *base* points to the element at the base of the table. *nel* is the number of elements in the table. *width* specifies the size of each element in bytes. *compar* is the name of the user-defined comparison function, which is called by `qsort()` with two arguments that point to the elements being compared. This function must return an integer less than, equal to, or greater than zero to indicate if the first argument is to be considered less than, equal to, or greater than the second.

The comparison function may be defined as follows:

```
/* Program fragment 1 compares two char values */
int comp(const void *a, const void *b)
{
    if(*((const char *)a) < *((const char *) b) )
        return(-1);
    else if(*((const char *)a) > *((const char *) b) )
        return(1);
    return(0);
}

/* Program fragment 2 compares two integer values */
int compare(const void *a, const void *b)
{
    return ( *((const int *) a) - *((const int *) b) );
}
```

Notes The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

### *Extension*

In contrast to XPG4, the order of array members that are considered equal by the comparison function is not changed. □

See also `stdlib.h`.

## raise - send signal to calling process

Syntax `#include <signal.h>`

```
int raise (int sig);
```

Description If the function is called with POSIX functionality, its behavior conforms with XPG4 as described below:

`raise()` sends the signal *sig* to the calling process. The defined signals are listed in `signal.h`.

If threads are used, then the function affects the process or a thread in the following manner:

- Sends a signal to the calling thread. The effect of `raise(sig)` is equivalent to calling `pthread_kill(pthread_self(), sig)`.

*BS2000*

- The following deviations in behavior must be noted if the function is called with BS2000 functionality:
- `raise()` can be used to simulate STXIT events as well as to send STXIT-independent signals (self-defined or predefined by the C runtime system).
- The following subset of the signals defined in `signal.h` may be used for *sig*:

Signal	STXIT class	Meaning
SIGHUP	ABEND	Disconnection of link to terminal
SIGINT	ESCPBRK	Interrupt from the terminal with <code>K2</code>
SIGILL	PROCHK	Execution of an invalid instruction
SIGABRT	–	raise signal for program abort with <code>_exit(-1)</code>
SIGFPE	PROCHK	Error in a floating-point operation
SIGKILL	–	raise signal for program abort with <code>exit(-1)</code>
SIGSEGV	ERROR	Memory access with invalid segment access
SIGALRM	RTIMER	A time interval has elapsed (real time)
SIGTERM	TERM	Signal at program termination
SIGUSR1	–	Defined by the user
SIGUSR2	–	Defined by the user
SIGDVZ	PROCHK	Division by 0
SIGXCPU	RUNOUT	CPU time has run out
SIGTIM	TIMER	A time interval has elapsed (CPU time)
SIGINTR	INTR	SEND-MESSAGE command

□

## Return value

- 0 if the signal was sent successfully.
- 1 if an error occurs. `errno` is set to indicate the error.

Errors `raise()` will fail if:*Extension*

- `EINVAL` The value of *sig* is an invalid signal number. □

## Notes

`raise(int sig)` uses the following call to `kill` to send the signal to the calling process:

```
kill(getpid(), sig);
```

A detailed list of error conditions can be found under `kill()`.

*BS2000*

With the exception of `SIGKILL` and `SIGSTOP`, the above signals can be intercepted with the `signal()` function (see `signal()`).

If the program does not provide for the handling of `raise` signals, the process is terminated with `exit(-1)` when a signal arrives, and the following messages are displayed:

```
"CCM0101 signal occurred: signal"
```

```
"CCM0999 Exit -1"
```

The `SIGABRT` signal causes the program to terminate with `_exit(-1)`. In contrast to `exit(-1)`, the termination routines registered with `atexit()` are not called and open files are not closed.

The `SIGKILL` signal causes the program to terminate with `exit(-1)`. In contrast to `SIGABRT`, `SIGKILL` cannot be intercepted, i.e. `signal` calls which specify the name of a self-defined function or `SIG_IGN` as the argument are not valid for `SIGKILL`. □

## See also

`atexit()`, `exit()`, `_exit()`, `kill()`, `sigaction()`, `signal()`, `signal.h`.

## rand - pseudo-random number generator (int)

**Syntax**     `#include <stdlib.h>`  
              `int rand(void);`  
              `void srand(unsigned int seed);`

**Description** `rand()` returns a positive random integer in the range  $[0, 2^{15}-1]$ .

A `rand` call selects values from a series of pseudo-random numbers by using a multiplicative, congruent random-number generator. The generator has a period of  $2^{32}$ .

`rand()` is not thread-safe. Use the reentrant function `rand_r()` when needed.

**Return value**  
Random number in the range  $[0, 2^{15}-1]$  if successful.

**Notes**       The random-number generator can be initialized or reset with `srand()`. If no initialization takes place, the random-number generator starts with its default value.

**See also**    `drand48()`, `rand_r()`, `random()`, `srand()`, `stdlib.h`.

## rand\_r - pseudo-random number generator (int, thread-safe)

**Syntax**     `#include <stdlib.h>`  
              `int rand_r(unsigned int *seed);`

**Description** The function `rand_r()` is the thread-safe version of `rand()`.

The function `rand_r()` returns an pseudo-random integer between 0 and  $2^{15}-1$ . If `rand_r()` is called with the same initial value for the object pointed to by `seed` and this object is not changed between sequential calls to `rand_r()`, the same series of pseudo-random numbers is created.

**Return value**  
The function `rand_r()` returns a pseudo-random number.

**See also**    `rand()`, `stdlib()`.

## random - create pseudo-random numbers

**Syntax**      `#include <stdlib.h>`  
                `long random(void);`

**Description** See `initstate()`.  
`random()` creates pseudo-random numbers in the range 0 through  $2^{31}-1$ .  
`random()` is not thread-safe. Use the reentrant function `rand_r()` when needed.

**Return value**  
Pseudo-random number (see `initstate()`).

**Example**      `/* Initialize an array and pass it to initstate. */`  
`static long state1[32] = { 3, 0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,`  
`0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd, 0x8c2e680f, 0xeb3d799f,`  
`0xb11ee0b7, 0x2d436b86, 0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,`  
`0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc, 0xde3b81e0, 0xdf0a6fb5,`  
`0xf103bc02, 0x48f340fb, 0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,`  
`0xf5ad9d0e, 0x8999220b, 0x27fb47b9 };`  
`main()`  
`{`  
`unsigned seed;`  
`int n;`  
`seed = 1;`  
`n = 128;`  
`initstate(seed, state1, n);`  
`setstate(state1);`  
`printf("%d\n", random());`  
`}`

**See also**      `drand48()`, `rand()`, `rand_r()`, `srand()`, `stdlib.h`

## read - read bytes from file

**Syntax**        `#include <unistd.h>`  
`ssize_t read(int fd, void *buf, size_t nbyte);`

**Description** `read()` reads *nbyte* bytes from the file associated with the open file descriptor, *fd*, into the buffer pointed to by *buf*.

*fd* is a file descriptor returned by a call to `creat()`, `open()`, `dup()`, `fcntl()` or `pipe()`.

If *nbyte* is 0, `read()` will return only the value 0 and *buf*.

On files that support seeking (for example, a regular file), `read()` starts at a position in the file given by the file offset associated with *fd*. The file offset is incremented by the number of bytes actually read.

Files that do not support seeking, for example, terminals, always read from the current position. The value of a file offset associated with such a file is undefined.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 will be returned.

The following occurs when attempting to read from an empty pipe or FIFO:

- If no process has the pipe open for writing, `read()` will return 0 to indicate end-of-file.
- If a process has the pipe open for writing and `O_NONBLOCK` is set, `read()` will return -1 and set `errno` to `EAGAIN`.
- If a process has the pipe open for writing and `O_NONBLOCK` is clear, `read()` will block until some data is written or the pipe is closed by all processes that had the pipe open for writing.

The following occurs when attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

- If `O_NONBLOCK` is set, `read()` will return a -1 and set `errno` to `EAGAIN`.
- If `O_NONBLOCK` is clear, `read()` will block until some data becomes available.
- The use of the `O_NONBLOCK` flag has no effect if there is some data available.

The `read()` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read()` returns null bytes. For example, `lseek()` allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return null bytes until data is written into the gap.

Upon successful completion, where *nbyte* is greater than 0, `read()` will mark for update the `st_atime` structure component of the file (see `sys/stat.h`), and return the number of bytes read. This number will never be greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte*, if the `read()` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a `read()` from a file associated with a terminal may return one typed line of data.

If a `read()` is interrupted by a signal before it reads any data, it will return -1 with `errno` set to `EINTR`.

If a `read()` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

If threads are used, then the function affects the process or a thread in the following manner: When an attempt is made to read from an empty pipe or FIFO, the following occurs: If a process has opened the pipe for writing and `O_NONBLOCK` is not set, `read()` blocks the calling process until data is written or until the pipe is closed by all processes that have opened it for reading. When an attempt is made to read from a file that is not a pipe or a FIFO that supports non-blocking reads and for which there is no data currently available, the following occurs: If `O_NONBLOCK` is not set, `read()` blocks the calling process until data becomes available.

#### Return value

- Number of bytes actually read upon successful completion.
- 0 at end-of-file.
- 1 if an error occurs. The contents of the buffer to which *buf* points are undefined. `errno` is set to indicate the error.

#### Errors

`read()` will fail if:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor, and the process would be delayed by the read operation.

#### *Extension*

`EAGAIN` The currently available amount of system memory for "raw" I/O is insufficient, or there is no data in a terminal device file waiting to be read, and `O_NONBLOCK` is set, or there is no message in a stream waiting to be read, and `O_NONBLOCK` is set. □

`EBADF` *fdes* is not a valid file descriptor open for reading.

`EFAULT` *buf* points outside the allocated address space of the process.



EINTR	The read operation was terminated due to the receipt of a signal, and no data was transferred.
EINVAL	An attempt was made to read from a stream linked with a multiplexer.
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.
ENXIO	A request was made for a non-existent device or the request exceeded the capabilities of the device.

**Notes**

The number of bytes actually read may be less than the value specified in *nbytes* if the end of the line is reached first (only for text files) and at end-of-file or the occurrence of an error.

The `sizeof()` function should be used to ensure that the number of bytes does not exceed the capacity of the buffer.

The program environment determines whether `read()` is executed for a BS2000 or POSIX file.

**See also** `fcntl()`, `lseek()`, `open()`, `pipe()`, `unistd.h`, and section "General terminal interface" on page 94.

## readdir - read directory

**Name**        **readdir, readdir64**

**Syntax**      `#include <dirent.h>`  
`#include <sys/types.h>`

```
struct dirent *readdir (DIR *dirp);  
struct dirent64 *readdir64 (DIR *dirp);
```

**Description** The data type `DIR`, which is defined in the header `dirent.h`, represents a directory stream, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of `readdir()`.

`readdir()` returns a pointer to a structure containing the next non-empty directory entry in the directory stream to which `dirp` points, and positions the directory stream at the next entry. It returns a null pointer upon reaching the end of the directory stream. The directory entry is described by the structure `dirent` (see `dirent.h`).

`readdir()` does not return directory entries containing empty names. If entries for dot (current directory) or dot-dot (parent directory) exist, one entry is returned for dot, and only one entry is returned for dot-dot.

The pointer returned by `readdir()` points to data which may be overwritten 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 was removed from or added to the directory after the most recent call to `opendir()` or `rewinddir()`, it is undefined whether a subsequent call to `readdir()` will return an entry for that file.

`readdir()` can buffer multiple directory entries in a single read operation; it updates the `st_atime` structure component of the directory each time the directory is actually read (see also `sys/stat.h`).

After a call to `fork()`, either the parent or child (but not both) may continue processing the directory stream by using `readdir()`, `rewind()` or `seekdir()`. If both the parent and child processes use these functions, the result is undefined.

There is no difference in functionality between `readdir()` and `readdir64()` except that `readdir64()` uses a `dirent64` structure.

The `dirent64` structure corresponds to the `dirent` structure except for the following components:

```
ino64_t d_ino
```

`readdir()` and `readdir64()` are not thread-safe. Use the reentrant function `readdir_r()` instead of `readdir()` if needed. There is currently no reentrant version of the `readdir64()` function.

**Return value**

`readdir()` and `readdir64()`:

Pointer to an object of type `struct dirent` upon successful completion.

Null pointer if the end of the directory is encountered. `errno` is not changed.

Null pointer if an error occurs. `errno` is set to indicate the error.

**Errors**

`readdir()` and `readdir64()` fail if:

`EBADF` The *dirp* argument does not point to an open directory stream.

`ENOENT` The current position of the directory stream is invalid.

`E_OVERFLOW` A value in the structure returned cannot be correctly represented.

**Notes**

`readdir()` should be used in conjunction with `opendir()`, `closedir()` and `rewinddir()` to examine the contents of the directory. As `readdir()` returns a null pointer both at the end of the directory and on error, an application wishing to check for error situations should set `errno` to 0 before calling `readdir()`, then check the value of `errno`, and if it is non-zero, assume that an error has occurred.

`readdir()` is executed only for POSIX files.

**See also**

`closedir()`, `opendir()`, `readdir_r()`, `rewinddir()`, `dirent.h`, `sys/stat.h`, `sys/types.h`.

## readdir\_r - read directory (thread-safe)

**Syntax**

```
#include <sys/types.h>
#include <dirent.h>
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

**Description** The function `readdir_r()` is the thread-safe version of the function `readdir()`.

The function `readdir_r()` initializes the `dirent` structure pointed to by `entry` with the next non-empty directory entry in the directory stream pointed to by `dirp`, stores a pointer to this structure at the location pointed to by `result` and positions the directory stream to point to the next entry.

The storage area pointed to by `entry` must be large enough to store `{NAME_MAX}` plus one character for the char array `d_name` from the `dirent` structure in the worst-case scenario.

If it returns successfully, the pointer returned for `*result` has the same value as the `entry` argument. If the end of the directory stream has been reached, this pointer contains the value `NULL`.

The function `readdir_r()` does not return any directory entries that contain empty names.

`readdir_r()` can temporarily store several directory entries for a single read operation; `readdir_r()` updates the `st_atime` structure component of the directory every time the directory is actually read.

**Return value**

0 if successful.

**Error number** otherwise, to indicate an error. `errno` is set to indicate the error.

**Errors** The function `readdir_r()` fails if:

`EBADF` The `dirp` argument does not point to an open directory stream.

**See also** `readdir()`, `dirent()`, `types()`.

## readlink - read contents of symbolic link

**Syntax**      `#include <unistd.h>`

```
int readlink(const char *path, char *buf, size_t bufsiz);
```

**Description** `readlink()` places the contents of the symbolic link referred to by *path* in the buffer *buf*, which has size *bufsiz*. The contents of the link are not terminated with a null byte when returned.

**Return value**

Number of bytes placed in the buffer upon successful completion.

-1 if an error occurs. `errno` is set to indicate the error. The contents of the buffer remain unchanged.

**Errors**      `readlink()` will fail if:

EACCES      Search permission is denied for a component of the path prefix of *path*.

EFAULT      *path* or *buf* are outside the allocated address space of the process.

EINVAL      *path* is not a symbolic link.

*Extension*

EINVAL      An attempt was made to access a BS2000 file. □

EIO      An I/O error occurred while reading from or writing to the file system.

ELOOP      Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG

The length of the *path* argument exceeds `{PATH_MAX}` or the length of a *path* component exceeds `{NAME_MAX}`.

ENOENT      The named file does not exist.

ENOSYS      The file system does not support symbolic links.

ENOTDIR      One of the component of the path prefix of *path* is not a directory.

**Notes**      `readlink()` will only access POSIX files.

**See also**    `stat()`, `symlink()`, `unistd.h`.

## readv - read array from file

Syntax `#include <sys/uio.h>`

```
ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

Description See `read()`.

`readv()` behaves like `read()` but reads the input data from the file belonging to *fildes* into the *iovcnt* buffers which are specified as elements of the *iov* field:

*iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1].

0 must be  $< \text{iovcnt} \leq \{ \text{IOV\_MAX} \}$

The *iovec* structure contains the following elements:

```
addr_t    iov_base;
size_t    iov_len;
```

Each *iovec* entry specifies the basic address and length of a storage area (buffer) in which data is to be put. `readv()` always fills a buffer completely before going on to the next one.

If successful, `readv()` returns the number of bytes that were actually read and written to the buffer. If the end of file is reached, 0 is returned.

Return value

integer >0	if successful. The number is the number of bytes that were actually read.
0	if the end of file (EOF) was reached during reading.
-1	if an error occurs. <code>errno</code> is set to indicate the error. The contents of the buffer are undefined.

Errors `readv()` will fail if:

EAGAIN	The <code>O_NONBLOCK</code> flag is set for the file descriptor and the process would be suspended by the read operation.
--------	---

*Extension*

EAGAIN	The currently available amount of system memory for "raw" I/O is insufficient, or there is no data in a terminal device file waiting to be read, and <code>O_NONBLOCK</code> is set, or there is no message in a stream waiting to be read, and <code>O_NONBLOCK</code> is set. □
--------	---

EBADF	<i>fildes</i> is not a valid file descriptor open for reading.
-------	--

EBADMSG	The file is a STREAM file in control-normal mode, but the message waiting to be read contains a control section.
---------	--

EFAULT	<i>iov</i> points outside the allocated address space of the process.
EINTR	The read operation was terminated due to the receipt of a signal, and no data was transferred.
EINVAL	An attempt was made to read from a stream linked with a multiplexer, or the sum of the <i>iov-len</i> values in the <i>iov</i> field caused a <code>ssize_t</code> overflow or <i>iovcnt</i> $\leq 0$ or <i>iovcnt</i> $>16$ .
EIO	A physical I/O error has occurred or the process is a member of a background process group attempting to read from its controlling terminal. The process is ignoring or blocking the SIGTTIN signal or the process group is orphaned.
EISDIR	<i>files</i> describes a directory that cannot be read with <code>readv()</code> . <code>readdir()</code> should be used instead.
ENXIO	A request was made for a non-existent device or the request exceeded the capabilities of the device.
ENOLINK	<i>files</i> is located on a remote computer to which the link is no longer active.

A `readv()` from a STREAMS file will also fail if an error message is received at the stream head. In this case, `errno` is set to the value that is returned in the error message. If a hang-up occurs in the stream currently being read, `readv()` continues running normally until the read queue of the stream head is empty. Thereafter 0 is returned.

**See also** `fcntl()`, `ioctl()`, `lseek()`, `open()`, `pipe()`, `stropts.h`, `sys/uio.h`, `unistd.h`.

## realloc - memory reallocator

**Syntax**      `#include <stdlib.h>`  
`void *realloc(void *ptr, size_t size);`

**Description** `realloc()` changes the size of the memory area pointed to by *ptr* to *size* bytes.

`realloc()` is part of a C-specific memory management package that internally administers memory areas which are requested and subsequently freed. As far as possible, all new requests are first satisfied from the areas that are already being managed, and only then from the operating system.

*ptr* is a pointer to the start of the memory area to be altered. It must be a pointer that was returned earlier by `malloc()` or `calloc()`.

*size* is an integer value that specifies the new size in bytes.

### Return value

**Pointer**            to the start of the reallocated memory area if successful.

**Null pointer**      if `realloc()` could not reallocate the space, e.g. because there was not enough memory available or because an error occurred. `errno` is set to indicate the error.

### Errors

`realloc()` will fail if:

**ENOMEM**            There is not enough memory available.

### Notes

Changing the size of a memory area with `realloc()` may cause the allocated block to be shifted. In such cases, the contents of the pointer passed as an argument are not identical to the return value.

The contents of the block are preserved up to the minimum of the old (when enlarging) and new (when reducing) sizes.

If `realloc()` returns a null pointer, the block to which *ptr* points may have been destroyed!

If *ptr* is a null pointer, `realloc()` has the same effect as a `malloc` call for the specified size.

**See also**      `calloc()`, `free()`, `malloc()`, `stdlib.h`.



## realpath - output real file name/pathname

Syntax `#include <stdlib.h>`

```
char *realpath (const char *file_name, char *resolved_name);
```

Description From the pathname specified in *file\_name*, `realpath()` derives an absolute pathname in which all symbolic links and references to `'.'` and `'..'` are resolved. This “real” pathname is stored in *resolved\_name* up to `{MAX_PATH}` bytes.

Both relative and absolute pathnames can be processed. With absolute pathnames and relative pathnames whose resolved name cannot be printed out relatively (e.g. `../.. /reldir`), the resolved absolute name is returned. For the other relative pathnames the resolved relative name is returned.  
*resolved\_name* must be large enough to incorporate the resolved pathname.

Return value

Pointer to *resolved\_name*  
if successful.

Null pointer otherwise. `errno` is set to indicate the error.

Errors

`realpath()` will fail if:

EACCES Read or search permission is denied for a component of *file\_name*.

EINVAL The *file\_name* or *resolved\_name* argument is a null pointer.

EIO An I/O error occurred during reading from the file system.

ENAMETOOLONG

The length of the *file\_name* argument exceeds `{PATH_MAX}`, or the length of a component of *file\_name* exceeds `{NAME_MAX}`.

In resolving a symbolic link, a interim result was produced whose length exceeds `{PATH_MAX}`.

ENOENT A component of the path prefix does not exist or *file\_name* is an empty string.

ENOTDIR A component of the path prefix is not a directory.

ENOMEM There is no longer enough memory available.

Notes

`realpath()` handles null-terminated strings.

You should have execution permission for all directories in the given and resolved path. In certain circumstances `realpath()` may not return to the current directory if an error occurs.

See also `getcwd()`, `sysconf()`, `stdlib.h`.

## re\_comp, re\_exec - compile and execute regular expressions

Syntax      `#include <re_comp.h>`  
  
`char *re_comp(const char *string);`  
`int re_exec(const char *string);`

Description `re_comp()` compiles a string into an internal format that is suitable for pattern matching. `re_exec` compares the string pointed to by *string* with the last regular expression that was passed to `re_comp()`.

If `re_comp()` is called with the value 0 or a null pointer, the current regular expression remains unchanged.

The strings that are passed to `re_comp()` and `re_exec()` must be null-terminated. They can contain terminating or embedded newline characters.

`re_comp()` and `re_exec()` support simple regular expressions. The rules which apply for the pattern matching are described below.

1. Regular one-character expressions match a character according to the following rules:
  - 1.1 An ordinary character (none of the special characters listed under 1.2) is a regular expression which matches itself.
  - 1.2 A backslash ( \ ) followed by a special character is a regular one-character expression that matches this special character. The following special characters are defined:
    - Period (.), asterisk (\*), opening square bracket ([) and backslash (\). These characters are special characters unless they occur in square brackets [ ] (see 1.4).
    - Circumflex (^) is a special character if it occurs at the beginning of a regular expression or if it occurs in square brackets and immediately follows the opening bracket ( [^ ] ) (see 1.4).
    - Dollar (\$) is a special character if it occurs at the end of a regular expression (see 3.2).
    - The character used to delimit a regular expression is a special character for this regular expression.
  - 1.3 A period (.) is a regular one-character expression which matches all characters except the newline character.

1.4 A non-empty string enclosed in square brackets is a regular one-character expression which matches every individual character in this string. If, however, the first character in the string is a circumflex (^), the regular expression matches all characters except for the remaining characters in the string and the newline character. But the ^ character only has this "power of exclusion" if it is the first character after the opening square bracket.

The minus sign (-) can be used to denote a range of consecutive ASCII characters, e.g. [0-9] and [0123456789] mean the same. The minus sign is not a special character if it is the first (possibly after a ^) or last character in the string.

The closing square bracket does not end such a string if it is the first character (possibly after a ^) in the string. For example, [a-f matches a closing square bracket ] or one of the characters a, b, c, d, e or f.

The four characters period (.), asterisk (\*), opening square bracket ([) and backslash (\) stand for themselves within such a string.

2. With the help of the following rules, regular expressions can be constructed from regular one-character expressions:

2.1 A regular one-character expression is a regular expression that matches everything that matches the regular one-character expression.

2.2 An asterisk (\*) followed by a regular one-character expression is a regular expression which matches 0 or several occurrences of the one-character expression.

If there is more than one possibility, the longest left-most substring that matches is selected.

2.3 A regular one-character expression followed by  $\{m\}$ ,  $\{m,\}$  or  $\{m,n\}$  is a regular expression that matches a multiple occurrence of the one-character expression.  $m$  and  $n$  must be non-negative integers less than 256.

$\{m\}$  matches exactly  $m$  occurrences,  $\{m,\}$  matches at least  $m$  occurrences and  $\{m,n\}$  matches occurrences between  $m$  and  $n$  (inclusive).

If there is more than one possibility, the highest number of occurrences that matches is selected.

2.4 The concatenation of regular expressions is a regular expression that matches a string which is produced from concatenation of the strings which match the corresponding components of the regular expression.

2.5 A regular expression which occurs between the strings \ ( and \) matches everything that matches the regular expression between these two strings.

2.6 The expression \n matches the same sequence of characters that earlier on in the same regular expression matched an expression enclosed in \ ( and \).  $n$  is a digit; the partial expression concerned begins with the  $n$ th occurrence of \, counting from the left. For example,  $^\(.\)1\$$  matches a line that consists of a string and its repetition.

3. In addition a regular expression can be restricted such that it matches only at the beginning of a line, the end of a line or both:
  - 3.1 A circumflex (^) at the beginning of a complete regular expression means that this expression only matches a string at the beginning of the line.
  - 3.2 A dollar sign (\$) at the end of a complete regular expression means that this expression only matches a string at the end of the line.  
For example, `^completeexpression$` means that the complete regular expression must match the entire line. The empty regular expression, i.e. `//`, is equivalent to the last regular expression that occurred.

## Return value

for `re_comp()`:

Null pointer     if `re_comp()` has compiled the passed string successfully.

String with error message  
                  otherwise.

for `re_exec()`:

1                if *string* matches the last compiled expression.

0                if *string* does not match the last compiled expression.

-1               if the compiled expression is invalid (in an internal error occurs).

Errors           In the event of an error, `re_comp()` returns one of the following strings:

No previous regular expression  
Regular expression too long  
unmatched \  
missing ]  
too many \  
(\)

Notes            A range contains all numbers that lie between the internal representation of the two range limits. This can be different in an EBCDIC and an ASCII environment.

For reasons of portability to implementations that comply with earlier versions of the X/Open standard, the `regcomp()` and `regexexec()` functions are recommended instead of the ones described here.

See also         `regcmp()`, `regexexec()`, `re_comp.h`.

## regcmp, regex - compile and execute regular expression

Syntax `#include <libgen.h>`

```
char *regcmp (const char *string1 [, char *string2, ...] / *, (char *) 0) */;
char *regex (const char *re, const char *subject [, char *ret0, ... ]);
extern char *__loc1;
```

Description `regcmp()` compiles the regular expression that is produced by concatenation of the arguments. The end of the argument chain is a null pointer. As the result, `regcmp()` returns a pointer to the expression which was compiled into an internal format. The memory for the compiled expression is provided via `malloc()`. The user is responsible for the release of the memory thus allocated if the space is no longer required.

The return of a null pointer by `regcmp()` indicates that an argument has an invalid value.

`regex()` searches for a pattern *re* compiled by `regcmp()` in the *subject* string. Additional arguments are passed to `regex()` to receive back matching partial expressions. If not enough arguments are specified for all returned hits, the behavior of `regex()` is undefined.

The global character pointer `__loc1` points to the first matching byte in *subject*.

`regcmp()` and `regex()` have been largely taken over by the editor `ed()`, although the syntax and semantics were changed slightly. The valid symbols and their respective meanings are as follows:

[ ]\* . ^ These symbols have the same meaning as in `ed()`.

\$ This symbol is equivalent to the end of the string (`\n` is equivalent to a newline character).

- A minus sign enclosed in brackets means *through*. So, for example, `[a-z]` means the same as `[abcd...xyz]`. The `-` can only mean 'minus' if it is used as the first or last character. So, for example, the expression `[ ]-` matches the characters `]` and `-`.

+ A regular expression followed by a `+` means *once or more*. So, for example, `[0-9]+` means the same as `[0-9] [0-9]*`.

{*m*} {*m*,} {*m*,*u*}

Integer values enclosed in `{ }` indicate the frequency with which the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is the maximum. *u* must be less than 256. If only *m* is present (e.g. `{m}`), this specifies exactly how often the regular expression is to be applied. The value `{m,}` is the same as `{m,infinite}`. The operations with the plus sign `+` and the asterisk `**` are equivalent to `{1,}` and `{0,}` respectively.

- ( ... )\$*n*     The value of the bracketed regular expression is to be returned. The value is stored in the (*n*+1)th argument after the *subject* argument. A maximum of ten bracketed regular expressions are permitted. `regex()` executes the assignments in all cases.
- ( ... )     Brackets are used for groupings. An operator, e.g. `*`, `+`, `{}`, can be applied to individual characters or to a regular expression enclosed in brackets. Example: `(a*(cb+)*)$0`.

All symbols defined above are special characters. They must therefore be preceded by a backslash `\` if they are to stand for themselves.

### Return value

for `regcmp()`:

Pointer to the compiled regular expression

if successful.

Null pointer     if an error occurs. `errno` is set to indicate the error.

for `regex()`:

Pointer to the next character in *subject* that does not match the pattern

if successful.

Null pointer     if an error occurs.

### Errors

`regcmp()` will fail if:

ENOMEM     There is no longer enough memory available.

### Notes

The user program may run out of memory if `regcmp()` is called iteratively without release of the arrays that are no longer required.

If you use one of these functions you must link the `libgen` library to it at compilation (`cc -lgen`).

**Example 1** The following example searches for a leading newline character in the string *subject* pointed to by *cursor*.

```
char *cursor, *newcursor, *ptr;
...
newcursor = regex((ptr = regcmp("^\\n", (char *)0)), cursor);
free(ptr);
```

**Example 2** The following example searches for a string `Testing3` and returns the address of the character after the last matching character (the character 4). The string `Testing3` is copied into the character field `ret0`.

```
char ret0[9];
char *newcursor, *name;
...
name = regcmp("([A-Za-z][A-Za-z0-9]{0,7})$0", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

**Example 3** In this example, a precompiled regular expression in `file.i` (see `regcmp(1)`) is checked against *string*.

```
#include "file.i"
char *string, *newcursor;
...
newcursor = regex(name, string);
```

**See also** `re_comp()`, `re_exec()`, `malloc()`.  
`ed()` in the manual "SINIX Commands V5.41".

## regex: advance, compile, step, loc1, loc2, locs - compile and match regular expressions

Syntax

```
#define INIT declarations
#define GETC () getc code
#define PEEKC() peekc code
#define UNGETC() ungetc code
#define RETURN(ptr) return code
#define ERROR(val) error code

#include <regex.h>

char *compile(char *instring, char *expbuf, const char *endbuf, int eof);
int step(const char *string, const char *expbuf);
int advance(const char *string, const char *expbuf);
extern char *loc1, *loc2, *locs;
```

Description These functions are general-purpose functions for handling regular expressions in programs that perform pattern matching for regular expressions. They are defined in the header `regex.h`.

Programs must have the following five macros declared before the `#include <regex.h>` statement. These macros are used by the `compile()` function. The macros `GETC()`, `PEEKC()` and `UNGETC()` operate on the regular expression given as input to `compile()`.

- `GETC()` returns the value of the next character (byte) in the regular expression pattern. The user must ensure that successive calls to `GETC()` return successive characters of the regular expression.
- `PEEKC()` returns the next character (byte) in the regular expression. The user must ensure that immediately successive calls to `PEEKC()` return the same byte, which should also be identical to the next character returned by `GETC()`.
- `UNGETC(c)` causes the argument `c` to be returned by the next call to `GETC()` and `PEEKC()`. No more than one character of pushback is ever needed, and this character is guaranteed to be the last character read by `GETC()`. The value of the macro `UNGETC(c)` is always ignored.
- `RETURN(ptr)` is used on normal exit of the `compile()` function. The value of the argument `ptr` is a pointer to the character after the last character of the compiled regular expression. This is useful to programs that have memory allocation to manage.



`ERROR (val)` corresponds to the abnormal termination of the `compile()` function. The argument *val* is an error number (see the Errors section below for the meanings of individual return values). The user must ensure that this call never returns.

The `step()` and `advance()` functions do pattern matching given a character string and a compiled regular expression as input.

The `compile()` function takes as input a simple regular expression and produces a compiled expression that can be used with `step()` and `advance()`.

The syntax of the `compile()` function is as follows:

```
char *compile(char *instring, char *expbuf, const char *endbuf, int eof);
```

- The first parameter, *instring*, is never used explicitly by `compile()` but is useful for programs that pass down different pointers to input characters. It is sometimes used in the `INIT` declaration (see below). Programs which invoke functions to input characters or which process characters in an external array can pass down the value `(char*)0` for this parameter.
- The next parameter, *expbuf*, is a pointer to `char`. It points to the place where the compiled regular expression will be placed.
- The parameter *endbuf* is one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in `(endbuf-expbuf)` bytes, a call to `ERROR(50)` is made.
- The parameter *eof* is the character which marks the end of the regular expression.

Each program that contains the `#include` statement for `regex.h` must also have a `#define` statement for the `INIT` macro. This macro is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for `GETC()`, `PEEKC()` and `UNGETC()`. Otherwise, it can be used to declare external variables that might be used by `GETC()`, `PEEKC()` and `UNGETC()`.

The `step()` and `advance()` functions have two parameters each:

- The first parameter, *string*, is a pointer to a string of characters to be checked against a regular expression. This string must be terminated with a null byte.
- The second parameter, *expbuf*, is the compiled regular expression which was obtained by a call to `compile()`.

`step()` returns a non-zero value if some substring of *string* matches the regular expression in *expbuf*; and it returns the value 0 if there is no match. If there is a match, two external character pointers are set as a side effect to the call to `step()`. The variable *loc1* points to the first character that matched the regular expression; the variable *loc2* points to the

character after the last character that matches the regular expression. Thus if the regular expression matches the entire input string, *loc1* will point to the first character of string and *loc2* will point to the null byte at the end of *string*.

`advance()` returns non-zero if the initial substring of *string* matches the regular expression in *expbuf*. If there is a match, an external character pointer, *loc2*, is set as a side effect. The variable *loc2* points to the next character in *string* after the last character that matched.

If the `advance()` function encounters an `*` character or the character sequence `\{ \}` in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, `advance()` will test whether the pattern sought is already contained in the previously matched substring by backing up along the string until it finds a match or reaches the point in the string that initially matched the `*` or `\{ \}`. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at some time during the backing up process, `advance()` will break out of the loop that backs up and will return 0.

The external variables *circf*, *sed* and *nbra* are reserved.

### Simple regular expressions (historical version)

A simple regular expression (SRE) specifies a set of character strings. A member of this set of strings is said to be matched by the SRE.

A pattern is constructed from one or more SREs. An SRE consists of ordinary characters or metacharacters.

Syntax elements for constructing patterns:

Regular expr.	Meaning	Example	Matching string
<i>r</i> +	One or more occurrences of the regular expression <i>r</i> . <i>r</i> must be in one of the following forms: <i>r</i> , <code>\r</code> , any character, <code>[r]</code> , <code>[rI-r2]</code> , <code>^[s]</code> , <code>^[rI-r2]</code> , <code>(r)</code> , <code>(rI r2)</code>	<code>u+</code>	<code>u</code> , <code>uu</code> , <code>uuu</code> , ...
<i>r</i> ?	Zero or one occurrence of the regular expression <i>r</i> . <i>r</i> must be in one of the following forms: <i>r</i> , <code>\r</code> , any character, <code>[r]</code> , <code>[rI-r2]</code> , <code>^[s]</code> , <code>^[rI-r2]</code> , <code>(r)</code> , <code>(rI r2)</code>	<code>u?</code>	none or <code>u</code>
<code>(r)</code>	Strings matching regular expression <i>r</i> . <i>r</i> can be any expression.	<code>(ok(abc))</code> <code>(au)*</code>	<code>okabc</code> none or <code>aus</code> , <code>auau</code> , ...
<code>(rI r2)</code>	Strings matching regular expression <i>rI</i> or <i>r2</i> .	<code>(ok ko)</code>	<code>ok</code> or <code>ko</code>

Within a pattern, all alphanumeric characters that are not part of a bracket expression, back-reference or duplication match themselves, i.e. the SRE pattern *abc*, when applied to a set of strings, will match only those strings containing the character sequence *abc* anywhere in them.

Only some of the characters, known as metacharacters, have a special meaning when used in regular expressions. The other characters match themselves. The regular expressions that may be used in `regex` functions are constructed as follows:

### Expression    Meaning

<i>c</i>	The character <i>c</i> , where <i>c</i> must not be a special character.
<code>\c</code>	The character <i>c</i> , where <i>c</i> is any character other than a digit in the range 1–9.
<code>^</code>	The beginning of the line being compared.
<code>\$</code>	The end of the line being compared.
<code>.</code>	Any character in the input.
<code>[s]</code>	Any character in the set <i>s</i> , where <i>s</i> is a sequence of characters. Ranges may be specified as <code>[c–c]</code> . The character <code>]</code> may be included in the set only in the first position; the character <code>-</code> may be included only in the first or last position, and the character <code>^</code> may be included by placing it anywhere other than first position in the set. Ranges in SREs are only valid if the <code>LC_COLLATE</code> category is set to the C locale.
<code>[^s]</code>	Any character not in the set <i>s</i> , where <i>s</i> is defined as above.
<i>r</i> *	Zero or more successive occurrences of the regular expression <i>r</i> . The longest leftmost matching string is used.
<i>rx</i>	The occurrence of regular expression <i>r</i> followed by the occurrence of regular expression <i>x</i> (concatenation).
<i>r</i> \{ <i>m</i> , <i>n</i> \}	Any number of <i>m</i> through <i>n</i> successive occurrences of the regular expression <i>r</i> . The regular expression <i>r</i> \{ <i>m</i> \} matches exactly <i>m</i> occurrences; <i>r</i> \{ <i>m</i> ,\} matches at least <i>m</i> occurrences. The maximum number of occurrences is matched.
<code>\(r\)</code>	The regular expression <i>r</i> . The <code>\(</code> and <code>\)</code> sequences are ignored.
<code>\n</code>	When <code>\n</code> is a number in the range 1–9 and appears in a concatenated regular expression, it stands for the regular expression <i>x</i> , where <i>x</i> is the <i>n</i> -th regular expression enclosed in <code>\(</code> and <code>\)</code> sequences that appeared earlier in the concatenated regular expression. For example, in the pattern <code>\(r\)x\(\(y</code> the <code>\2</code> matches the regular expression <i>y</i> , giving <i>rxzy</i> .

The following characters have special meaning when they do not appear within square brackets [ ] or are preceded by a \ (backslash): ., \*, [, \.

Other special characters, such as \$ have special meaning in more restricted contexts.

The character ^ at the beginning of an expression permits a successful match only immediately after a newline or at the beginning of each of the strings to which the match is applied, and the character \$ at the end of an expression requires a trailing newline.

Two characters have special meaning only when used within square brackets. The character - denotes a range, [c-c], unless it is just after the left square bracket or before the right square bracket, [-c] or [c-], in which case it has no special meaning. The character ^ has the meaning **complement of** if it immediately follows the left square bracket, [^c]. Elsewhere between brackets, [c^], it stands for the ordinary character ^. The right square bracket loses (]) its special meaning and represents itself in a bracket expression if it occurs first in the list after any initial circumflex (^) character.

The special meaning of the \ operator can be escaped only by preceding it with another \, that is, \\.

### SRE operator precedence

[...]            high precedence  
concatenation   low precedence

### Internationalized SREs

Character expressions within square brackets are constructed as follows:

*c*                A single character *c*, where *c* is not a special character.  
[[:*class*:]]      A *char class* expression. Any character of type *class*, as defined by category LC\_CTYPE in the program's locale (see the manual "POSIX Commands (BS2000/OSD)")

One of the following may be substituted for *class*:

*alpha*            a letter  
*upper*            an uppercase letter  
*lower*            a lowercase letter  
*digit*            a decimal digit  
*xdigit*           a hexadecimal digit  
*alnum*            an alphanumeric character (letter or digit)  
*space*            a blank

<i>punct</i>	a punctuation character
<i>print</i>	a printing character
<i>graph</i>	a character with a visible representation
<i>cntrl</i>	a control character
<code>[[=c=]]</code>	An equivalence class. Any collation element defined as having the same relative order in the current collation sequence as <i>c</i> . As an example, if <i>A</i> and <i>a</i> belong to the same equivalence class, then both <code>[[=A=]b]</code> and <code>[[=a=]b]</code> are equivalent to <code>[Aab]</code> .
<code>[[.cc.]]</code>	A collating symbol. Multi-character collating elements must be represented as collating symbols to distinguish them from single-character collating elements. As an example, if the string <i>ch</i> is a valid collating element, then <code>[[.ch.]]</code> will be treated as an element matching the same string of characters, while <i>ch</i> will be treated as a simple list of <i>c</i> and <i>h</i> . If the string <i>ch</i> is not a valid collating element in the current collating sequence definition, the symbol will be treated as an invalid expression.
<code>[c-c]</code>	Any collation element in the character expression range <i>c-c</i> , where <i>c</i> can identify a collating symbol or an equivalence class. If the character <code>-</code> appears immediately after an opening square bracket, for example, <code>[-c]</code> , or immediately prior to a closing square bracket, for example, <code>[c-]</code> , it has no special meaning.
<code>^</code>	Immediately following an opening square bracket, means the complement of, for example, <code>^[c]</code> . Otherwise, it has no special meaning.

In the case of expressions within square brackets, a `.` that is not part of a `[[.cc.]]` sequence, or a `:` that is not part of a `[[:class:]]` sequence, or an `=` that is not part of a `[[=c=]]` sequence, matches itself.

### Examples of regular expressions

<code>ab.d</code>	<i>ab</i> any character <i>d</i>
<code>ab.*d</code>	<i>ab</i> any sequence of characters (including none) <i>d</i>
<code>ab[xyz]d</code>	<i>ab</i> one of the characters <i>x y</i> or <i>z d</i>
<code>ab[^c]d</code>	<i>ab</i> any character, except <i>c d</i>
<code>^abcd\$</code>	a line containing only <i>abcd</i>
<code>a-d</code>	any one of the characters <i>a b c</i> or <i>d</i>

Return value

- RETURN()      when `compile()` is successful.
- ≠ 0            when `step()` and `advance()` are successful.
- ERROR         if `compile()` fails.
- 0               if `step()` and `advance()` fail.

Errors

- 11              Range endpoint too large
- 16              Invalid number
- 25              `\digit` out of range
- 36              Illegal or missing delimiter
- 41              No remembered search string in memory
- 42              `\(\)` imbalance
- 43              Too many `\(`
- 44              More than two numbers given in `\{\}`
- 45              `}` expected after `\`
- 46              First number exceeds second in `\{\}`
- 49              `[ ]` imbalance
- 50              Regular expression overflow

See also

`fnmatch()`, `glob()`, `regcomp()`, `regexec()`, `stlocale()`, `regex.h`, `regex.h`, and the manual "POSIX Commands (BS2000/OSD)".

## remainder - remainder from division

**Syntax**      `#include <math.h>`  
`double remainder (double x, double y);`

**Description** `remainder()` returns the floating-point remainder from dividing  $x$  by  $y$ . More precisely, it returns the value  $r = x - yn$  if  $y \neq 0$ , where  $n$  is the integer closest to the exact value  $x/y$ . If  $|n - x/y| = 1/2$ , the even value is chosen for  $n$ .

**Return value**  
Floating-point remainder =  $x - ny$   
if  $y \neq 0$ .  
**HUGE\_VAL** if  $y = 0$ . `errno` is set to `EDOM`.

**Errors**      `remainder()` will fail if:  
**EDOM**             $y = 0$ .

**See also**      `abs()`, `math.h`.

## remove - remove files

**Syntax**      `#include <stdio.h>`  
`int remove(const char *path);`

**Description** `remove()` causes the file or empty directory named by the pathname pointed to by *path* to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew.

`remove()` is identical to `unlink()` for files, and identical to `rmdir()` for directories.

*BS2000*

`remove()` can also be used for files with record I/O. □

**Return value**  
0                    if successful.  
-1                   if an error occurs. `errno` is set to indicate the error.

**Errors**          See `unlink()` and `rmdir()`.

**Notes**           The program environment determines whether `remove()` is executed for a BS2000 or POSIX file.

*BS2000*

*path* can be a fully or partially qualified file name. If a partially qualified file name is specified, `remove()` will delete all corresponding files without first asking for a (Y/N) confirmation. The response "Y" is assumed.

`remove()` performs only a logical deletion of the file(s), i.e. the catalog entry is deleted, and the assigned memory is released.

If a file has been opened by any program, it is not deleted. □

**See also**        `rmdir()`, `unlink()`, `stdio.h`.



## remque - remove element from queue

**Syntax**     `#include <search.h>`  
              `void remque(void *element);`

**Description** See `insque()`.

`insque()` and `remque()` modify queues that are created from double-concatenated elements.

`insque()` inserts the entry *element* in a queue. `remque()` removes *element* from a queue.

## rename - rename file

Syntax `#include <stdio.h>`

```
int rename(const char *old, const char *new);
```

Description `rename()` changes the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new pathname of the file.

If *old* and *new* both refer to the same existing file, `rename()` returns successfully and performs no other action.

If *old* points to the pathname of a file that is not a directory, *new* must not point to the pathname of a directory. If the link named by the *new* argument exists, it is removed, and *old* is renamed to *new*. In this case, a link named *new* will remain visible to other processes throughout the renaming operation and will refer either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If *old* points to the pathname of a directory, *new* must not point to the pathname of a file that is not a directory. If the directory named by the *new* argument exists, it is removed, and *old* is renamed to *new*. In this case, a link named *new* will exist throughout the renaming operation and will refer either to the file referred to by *new* or *old* before the operation began. Thus, if *new* names an existing directory, it must be an empty directory.

The pathname prefix of *new* must not be identical to *old*. Write access permission is required for the directory containing *old* and the directory containing *new*.

If *old* points to the pathname of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link named by *new* exists, and the file's link count becomes 0 when it is removed, and no process has the file open, the space occupied by the file will be freed, and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before `rename()` returns, but the removal of the file contents will be postponed until all references to the file are closed.

Upon successful completion, `rename()` will mark for update the `st_ctime` and `st_mtime` fields of the parent directory of each file.

*BS2000*

`rename()` can also be used without changes for files with record I/O. □

Return value

0	if successful
-1	if an error occurs; <code>errno</code> is set to indicate the error. Neither the file named by <i>old</i> nor the file named by <i>new</i> will be changed or created.

*BS2000*

errno is set to EMACRO.

If *old* and *new* point to files from different file systems, no changes are made.

errno is set to EXDEV. □

Errors rename() will fail if:

EACCES A component of either path prefix denies search permission; or one of the directories containing *old* or *new* denies write permissions; or write permission is required and is denied for a directory pointed to by the *old* or *new* arguments.

EBUSY One of the directories named by *old* or *new* is currently in use by the system or another process, and the implementation considers this an error.

*Extension*

EDQUOT The directory in which the entry for the new name is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted. □

EEXIST or ENOTEMPTY

The link specified by *new* is a non-empty directory.

*Extension*

EFAULT *old* or *new* points outside the allocated address space of the process.

EINTR A signal was caught during execution of the rename() system call. □

EINVAL The directory pathname *new* contains a path prefix that designates the directory *old* (see also "Notes").

*Extension*

EIO An I/O error occurred when creating or updating a directory entry. □

EISDIR The *new* argument points to a directory and the *old* argument points to a file that is not a directory.

*Extension*

ELOOP Too many symbolic links were encountered in resolving *old* or *new*. □

*BS2000*

EMACRO There is no existing file with the name *old*, or there is already a file cataloged under the name *old*, or the file to be renamed has been opened by a program. □

EMLINK *old* points to a directory, and the link count of the parent directory of *new* exceeds {LINK\_MAX}.

ENAMETOOLONG	The length of <i>old</i> or <i>new</i> exceeds <code>{PATH_MAX}</code> or a pathname component is longer than <code>{NAME_MAX}</code> .
ENOENT	The link named by <i>old</i> does not name an existing file, or either <i>old</i> or <i>new</i> points to an empty string.
ENOSPC	The directory that would contain <i>new</i> cannot be extended.
ENOTDIR	A component of either path is not a directory, or the <i>old</i> argument names a directory, and the <i>new</i> argument names a non-directory file.
EROFS	The requested operation requires writing in a directory on a read-only file system.
EXDEV	The links named by <i>new</i> and <i>old</i> are on different file systems.

**Notes** `rename()` cannot be used to relocate a file from the POSIX subsystem to BS2000 or vice-versa. The following statement, for example, will produce the error `EINVAL`:

```
rename(/BS2/hugo, *POSIX(hugo))
```

The program environment determines whether `rename()` is executed for a BS2000 or POSIX file.

**See also** `link()`, `rmdir()`, `unlink()`, `stdio.h`.

## rewind - reset file position indicator to start of stream

**Syntax**      `#include <stdio.h>`

```
void rewind(FILE *stream);
```

**Description** The call `rewind(stream)` is equivalent to:  
`(void) fseek(stream, 0L, SEEK_SET)`  
except that `rewind()` also clears the error indicator for *stream*.

**Errors**      See `fseek()` - with the exception of `EINVAL`.

**Notes**        Since `rewind()` does not return a value, an application wishing to detect errors should first set `errno` to 0, then call `rewind()`, and if `errno` is non-zero, assume that an error has occurred.

The program environment determines whether `rewind()` is executed for a BS2000 or POSIX file.

*BS2000*

`rewind()` can also be used without changes for files with record I/O. □

**See also**     `fseek()`, `fsetpos()`, `stdio.h`.

## rewinddir - reset file position indicator to start of directory stream

Syntax `#include <dirent.h>`

*Optional*

`#include <sys/types.h> □`

`void rewinddir(DIR *dirp);`

**Description** `rewinddir()` resets the position of the directory stream to which *dirp* refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to `opendir()` would have done. If *dirp* does not refer to a directory stream, the effect is undefined.

After a call to the `fork()` function, either the parent or child (but not both) may continue processing the directory stream using `readdir()`, `rewinddir()` or `seekdir()`. If both the parent and child processes use these functions, the result is undefined.

**Notes** `rewinddir()` should be used in conjunction with `opendir()`, `readdir()` and `closedir()` to examine the contents of the directory. This method is recommended for portability.

`rewinddir()` is executed only for POSIX files.

**See also** `closedir()`, `opendir()`, `readdir()`, `dirent.h`, `sys/types.h`.

## rindex - get last occurrence of character in string

**Syntax**      `#include <string.h>`  
`char *rindex(const char *s, int c);`

**Description** see `strrchr()`.

`rindex()` searches for the last occurrence of character `c` in string `s` and returns a pointer to the located position in `s` if successful.

The terminating null byte (`\0`) is also treated as a character.

**Return value**

Pointer to the (last) position of `c` in string `s`, if successful.

Null pointer if `c` is not contained in string `s`.

**Notes**      `index()` and `strchr()` are equivalent.

In BS2000, as in many other operating systems, you cannot use the null pointer to denote a null string. In this case a null pointer is an error and causes the process to abort. If you want to specify a null string, you must use a pointer which points to an explicit null string. With some implementations of the C programming language on many computers, a null pointer, when de-referenced, would result in a null string; this trick, which is portable only in very few cases, has been used in some programs. Programmers who use a null pointer to point to an empty string should be aware of this portability question; even with machines on which de-referencing a null pointer does not cause the program to abort, it need not necessarily result in a null string.

The moving of characters is performed differently in different implementations. Overlapping can therefore lead to unpredictable results.

**See also**      `index()`, `strchr()`, `strrchr()`.

## rint, rintf, rintl - round to nearest integer value

**Syntax**      `#include <math.h>`

`double rint(double x);`

`float rintf(float x);`

`long double rintl(long double x);`

**Description**    The functions return the integer value (displayed as a number of type `double`) nearest to *x*. `rint()` represents the result as a number of type `double`, `rintf()` as a number of type `float` and `rintl()` as a number of type `long double`.

The returned value is rounded according to the currently set rounding mode of the computer. If the default mode is set to 'round-to-nearest' and the difference between *x* and the rounded result is exactly 0.5, the next even integer is returned.

If the currently set rounding mode rounds infinitely in the positive direction, `rint()` is identical to `ceil()`. If the currently set rounding mode rounds infinitely in the negative direction, `rint()` is identical to `floor()`.

In this version the rounding mode is set to positive infinity.

**Return value**

**Integer value**    represented as a number of type `double`, `float` or `long double` if successful.

`HUGE_VAL`        if an overflow occurs. `errno` is set to `ERANGE` to indicate the error.

**Notes**            In this version the rounding mode is set to positive infinity.

**See also**        `abs()`, `ceil()`, `floor()`, `llrint()`, `llround()`, `lrint()`, `lround()`, `round.h`.



## rmdir - remove directory

**Syntax**        `#include <unistd.h>`  
`int rmdir(const char *path);`

**Description** `rmdir()` removes a directory whose name is given by *path*. The directory is removed only if it is an empty directory.

If *path* is a symbolic link, it is not followed.

If *path* is the root directory, then *path* is set to `EBUSY`; if *path* is the current directory of an active process, the behavior of `rmdir()` is unspecified.

If the directory link count becomes 0 and no process has the directory open, the space occupied by the directory will be freed and the directory will no longer be accessible. If one or more processes have the directory open when the last link is removed, the dot and dot-dot entries, if present, are removed before `rmdir()` returns and no new entries may be created in the directory, but the directory is not removed until all references to the directory are closed.

Upon successful completion, `rmdir()` marks the `st_ctime` and `st_mtime` fields of the parent directory for update.

### Return value

0                if successful.  
-1                if an error occurs. `errno` is set to indicate the error.

### Errors

`rmdir()` will fail if:

`EACCES`        Search permission is denied on a component of the path, or write permission is denied on the parent directory of the directory to be removed.

`EBUSY`        The directory to be removed is currently in use by the system or another process.

`EEXIST` or `ENOTEMPTY`  
*path* names a directory that is not an empty directory.

#### *Extension*

`EFAULT`        *path* points outside the allocated address space of the process.

`EINVAL`        The directory to be removed is the current directory.

`EIO`            An I/O error occurred when accessing the file system.

`ELOOP`        Too many symbolic links were encountered in resolving *path*. □

ENAMETOOLONG

The length of the *path* argument exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}` and `{_POSIX_NO_TRUNC}` is set.

ENOENT

*path* names a non-existent directory or points to an empty string .

ENOTDIR

A component of the path is not a directory. □

EROFS

The directory entry to be removed resides on a read-only file system.

**Notes** `rmdir()` is executed only for POSIX files

**See also** `mkdir()`, `remove()`, `unlink()`, `unistd.h`.

## round, roundf, roundl - round up to next integer value

**Syntax**

```
#include <math.h>

double round(double x);
float roundf (float x);
long double roundl (long double x);
```

**Description** The functions return the integer value represented as a floating-point value (displayed as a number of type `double`) nearest to  $x$ . `round()` represents the result as a number of type `double`, `roundf()` as a number of type `float` and `roundl()` as a number of type `long double`.

The returned value is rounded according to the currently set rounding mode of the computer. If the default mode is set to 'round-to-nearest' and the difference between  $x$  and the rounded result is exactly 0.5, the next even integer is returned.

**Return value**

Integer value	represented as a number of type <code>double</code> , <code>float</code> or <code>long double</code> if successful.
undefined	if an overflow occurs. <code>errno</code> is set to <code>ERANGE</code> to indicate the error.

**See also** `abs()`, `ceil()`, `floor()`, `llrint()`, `llround()`, `lrint()`, `lround()`, `rint()`

## sbrk - modify size of data segment

Syntax `#include <unistd.h>`  
`void *sbrk(int incr);`

Description See `brk()`.

## scalb - load exponent of base-independent floating-point number

Syntax `#include <math.h>`  
`double scalb (double x, double n);`

Description `scalb()` computes  $x * r^n$ , where  $r$  is the base of the machine-dependent floating-point arithmetic. For  $r=2$ , `scalb()` is equivalent to `ldexp()`.

### Return value

$x * r^n$  if `scalb()` is executed successfully.  
`+/-HUGE_VAL` depending on the sign of  $x$  if `scalb()` causes an overflow. `errno` is set to `ERANGE`  
`0` if `scalb()` causes an underflow. `errno` is set to `ERANGE`.

Errors `scalb()` will fail if:

`ERANGE` `scalb()` attempts an overflow or underflow.

Notes An application that wants to check the error situation should set `errno` to `0` before the `scalb()` function is called. If on the return `errno` is then not equal to zero, this signals an error.

For BS2000 the base is  $r=16$

See also `ldexp()`, `math.h`

## scanf - read formatted input from standard input stream

Syntax      #include <stdio.h>  
              int scanf(const char \**format*[, *arglist*]);

Description See `fscanf()`.

## seed48 - set seed (int) for pseudo-random numbers

Syntax      #include <stdlib.h>  
              unsigned short int \*seed48 (unsigned short int *seed16v*[3]);

Description See `drand48()`.

## seekdir - set position of directory stream

Syntax `#include <dirent.h>`

*Optional*

`#include <sys/types.h>`

`void seekdir(DIR *dirp, long int loc);`

Description `seekdir()` sets the position of the next `readdir()` operation on the directory stream pointed to by `dirp` to the position specified by `loc`. The value of `loc` should have been returned from an earlier call to `telldir()`. The new position reverts to the one associated with the directory stream at the time the `telldir()` operation was performed.

*Extension*

Values returned by `telldir()` are valid only if the directory has not changed because of compaction or expansion. This situation is not a problem with System V, but it may present a problem with some file system types.

Errors `seekdir()` will fail if:

*Extension*

EBADF           The stream associated with the directory is no longer valid. This error occurs if the directory has been closed.

Notes `seekdir()` is executed only for POSIX files

See also `opendir()`, `readdir()`, `telldir()`, `dirent.h`, `sys/types.h`

## select - synchronous I/O multiplexing

```
Syntax    #include <sys/time.h>

int select ( int nfds, fd_set *readfds, fd_set *writefds,
             fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);

void FD_SET(int fd, fd_set *fdset);

void FD_ZERO(fd_set *fdset);
```

**Description** `select` checks the I/O descriptor sets that are transferred in `readfds`, `writefds` and `exceptfds` to see whether one of their descriptors is ready for reading or writing or has an error condition pending. `nfds` is the number of bits to be checked in each bit mask that displays a file descriptor set. The descriptors of the descriptor sets are checked from 0 through `nfds-1`. On return, `select` replaces the given descriptor set with subsets comprising descriptors that are ready for the desired operation. The return value of the `select()` call is the number of descriptors that are ready.

The descriptor sets are stored as bit fields in ascending order. The following macros are available for the manipulation of such descriptor sets:

```
FD_ZERO(&fdset)      initializes a descriptor set fdset with the null set.
FD_SET(fd,&fdset)   inserts a descriptor fd in fdset.
FD_CLR(fd,&fdset)   removes fd from fdset
FD_ISSET(fd,&fdset) is not zero if fd is an element from fdset, otherwise it is zero.
```

The behavior of these macros is not defined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`. `FD_SETSIZE` is a constant that is defined in `sys/select.h` and is normally at least as high as the maximum number of descriptors available from the system.

If `timeout` is not a null pointer, it specifies a maximum time to be waited until the selection is complete. If `timeout` is a null pointer, the `select` blocks until one of the queried events occurs. `select` does not block if a structure containing only null values is transferred. `readfds`, `writefds` and `exceptfds` can be specified as null pointers if none of the descriptors is of interest.

## Return value

Number	ready descriptors in the descriptor sets
-1	if an error occurs
0	if the time limit was exceeded

## Errors

An error return from `select` can be:

EBADF	One of the I/O descriptor sets has an invalid I/O descriptor.
EINTR	A signal was issued before one of the desired events occurred, or the time limit was exceeded.
EINVAL	A component of the time limit that is referenced is outside the permitted range: <i>t_sec</i> must be between 0 and 10 inclusive. <i>t_usec</i> must be greater than or equal to 0 and less than 10.

## Notes

The default value for `FD_SETSIZE` (currently 2048) is the same as the default limit for the number of open files. To adjust programs which use a larger number of open files with `select`, it is possible to increase this size within a program by defining a higher value for `FD_SETSIZE` before including `<sys/types.h>`.

In future versions of the system, `select` could return the time remaining from the original time limit (if there is any) if the time value is changed at the right place. It is therefore not advisable to assume that the value of the time limit will remain unchanged as a result of the `select` call.

The descriptor sets are always changed on return, even if the call returns as the result of a time limit.

## See also

`poll()`, `read()`, `write()`.



## semctl - semaphore control operations

Syntax `#include <sys/sem.h>`

```
int semctl(int semid, int semnum, int cmd, ...);
```

Description `semctl()` provides a variety of operations for controlling semaphores, as specified by *cmd*.

*cmd* is used to specify one of the semaphore control operations listed below; *semid* and *semnum* are used to specify the semaphore for which the specified operation is to be performed. The access permissions required for a particular operation are shown under the relevant command (see also section "Interprocess communication" on page 112). The symbolic names for the values for *cmd* are defined in the header file `sys/sem.h`:

GETVAL            Return the value of `semval` (see also `sys/sem.h`). Requires read permission.

SETVAL            Set the value of `semval` to the value of the fourth argument of type `int`. Upon successful execution of this *cmd*, the `semadj` value corresponding to the specified semaphore is cleared in all processes. Requires alter permission (see also section "Interprocess communication" on page 112).

GETPID            Return the value of `sempid`. Requires read permission.

GETNCNT          Return the value of `semncnt`. Requires read permission.

GETZCNT          Return the value of `semzcnt`. Requires read permission.

The following commands affect every `semval` in the set of permissible semaphore:

GETALL            Return the value of `semval` and place into the array pointed to by *arg.array*. Requires read permission.

SETALL            Set `semval` to the value of the array of type `unsigned short` pointed to by 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 commands are also available:

IPC\_STAT          Place the current value of each member of the `semid_ds` data structure associated with *semid* into the `semid_ds` structure pointed to by the fourth argument to `semctl()`.

- IPC\_SET** Set the value of the following members of the `semid_ds` data structure associated with `semid` to the corresponding value found in the `semid_ds` structure pointed to by the fourth argument to `semctl()`:
- ```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* only the least-significant 9 bits */
```
- This command may be executed only by a process that has an effective user ID equal to that of a process with appropriate privileges or which matches the value of `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid`.
- IPC\_RMID** Remove the semaphore-identifier specified by `semid` from the system and destroy the set of semaphores and the data structure associated with it. This command can only be executed by a process that has an effective user ID equal to that of a process with appropriate privileges or which matches the value of `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid`.

### Return value

If successful, `semctl()` returns one of the values below, which depends on `cmd` as follows:

- Value of `semval`**  
if `GETVAL` was specified for `cmd`.
- Value of `sempid`**  
if `GETVAL` was specified for `cmd`.
- Value of `semncnt`**  
if `GETVAL` was specified for `cmd`.
- Value of `semzcnt`**  
if `GETVAL` was specified for `cmd`.
- 0** if other `cmd` values were specified.
- 1** if unsuccessful. `errno` is set to indicate the error.

### Errors

`semctl()` will fail if:

- EACCES** The calling process does not have the required access permission for the command to be executed (see section "Interprocess communication" on page 112).

#### Extension

- EFAULT** `msgp` points to an invalid address. □
- EINVAL** `semid` is not a valid semaphore ID, `semnum` has a value less than 0 or greater than `sem_nsems`, or `cmd` is not a valid command.

|        |                                                                                                                                                                                                                                                                                  |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EPERM  | <i>cmd</i> is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not that of a process with appropriate privileges and does not match <code>sem_perm.cuid</code> or <code>sem_perm.uid</code> in the data structure associated with <i>semid</i> . |
| ERANGE | <i>cmd</i> is equal to SETVAL or SETALL and the value to which <code>semval</code> is to be set exceeds the highest value permitted in the system.                                                                                                                               |

Notes The fourth argument in the "Syntax" section is identified in XPG4 as ... in order to avoid a clash with the ISO C standard. The fourth argument can be defined by the application programmer as follows:

```
union semun
{ int val;
  struct semid_ds *buf;
  unsigned short *array;
} arg;
```

The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

See also `semget()`, `semop()`, `sys/sem.h`, section "Interprocess communication" on page 112.

## semget - get semaphore ID

**Syntax**      `#include <sys/sem.h>`

```
int semget(key_t key, int nsems, int semflg);
```

**Description** `semget()` creates a semaphore identifier with its associated `semid_ds` data structure and its associated set of `nsems` semaphores (see `sys/sem.h`) for the argument `key` if one of the following is true:

- `key` has the value `IPC_PRIVATE`.
- No semaphore ID has been created yet for `key` and  $(semflg \& IPC\_CREAT)$  is not equal to 0.

When the new semaphore ID `key` is created, the corresponding data structure `semid_ds` is initialized as follows:

- The effective user ID and the effective group ID of the calling process are entered for the structure components `sem_perm.cuid`, `sem_perm.uid`, `sem_perm.cgid` and `sem_perm.gid`.
- The 9 low-order bits of `sem_perm.mode` are set equal to the 9 low-order bits of `semflg`.
- `sem_nsems` is set to the value of `nsems`.
- `sem_otime` is set to 0 and `sem_ctime` is set equal to the current time.
- The data structures associated with the individual semaphores are not initialized. The `semctl()` function with the command `SETVAL` or `SETALL` can be used to initialize each semaphore.

**Return value**

Semaphore ID

if successful. The semaphore ID is a non-negative integer.

-1 if unsuccessful. `errno` is set to indicate the error.

**Errors** `semget()` will fail if:

**EACCES**      There already exists a semaphore ID for `key`, but the permission specified in the 9 low-order bits of `semflg` was not granted.

**EEXIST**      A semaphore ID exists for the `key`, but  $((semflg \& IPC\_CREAT) \&\& (semflg \& IPC\_EXCL))$  is not equal to 0.

|        |                                                                                                                                                                                                                                                                                             |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EINVAL | The value of <i>nsems</i> is either less than or equal to 0 or exceeds the maximum value specified by the system, or a semaphore ID exists for the argument <i>key</i> , but the corresponding semaphore set contains less than <i>nsems</i> semaphores and <i>nsems</i> is not equal to 0. |
| ENOENT | No semaphore ID exists for <i>key</i> and ( <i>semflg</i> & IPC_CREAT) is equal to 0.                                                                                                                                                                                                       |
| ENOSPC | A semaphore ID is to be created, but this would exceed the maximum number of semaphores permitted in the system.                                                                                                                                                                            |

**Notes** The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

**See also** `semctl()`, `semop()`, `sys/sem.h`, section "Interprocess communication" on page 112.

## semop - semaphore operations

**Syntax** `#include <sys/sem.h>`

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

**Description** `semop()` permits the automatic execution of a user-defined list of semaphore operations on the semaphore set with the semaphore ID specified in the argument *semid*.

*sops* points to a user-defined array of semaphore operation structures.

*nsops* specifies the number of structures in the array.

Each `sembuf` structure contains the following members:

| Data type | Member name          | Description         |
|-----------|----------------------|---------------------|
| short     | <code>sem_num</code> | Semaphore number    |
| short     | <code>sem_op</code>  | Semaphore operation |
| short     | <code>sem_flg</code> | Operation flags     |

Each semaphore operation defined by `sem_op` is performed on the semaphore specified by *semid* and `sem_num`.

`sem_op` defines one of the following three semaphore operations:

1. If `sem_op` is a negative integer and the calling process has alter permission, one of the following occurs:
  - If `semval` is greater than or equal to the absolute value of `sem_op`, the absolute value of `sem_op` is subtracted from `semval`.
  - 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 (see `exit()`).
  - If `semval` is less than the absolute value of `sem_op`, and `(sem_flg & IPC_NOWAIT)` is non-zero, `semop()` will return immediately.
  - If `semval` is less than the absolute value of `sem_op` and `(sem_flg & IPC_NOWAIT)` is 0, `semop()` increments the `semncnt` value of the specified semaphore and suspends execution of the calling process 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 `semncnt` value of the specified semaphore is decremented by 1, the absolute value of `sem_op` is subtracted from `semval` and, 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.

- The `semid` for which the calling process is awaiting action is removed from the system. In this case, `errno` is set equal to `EIDRM` and the value `-1` is returned.
  - The calling process receives a signal that is to be caught. When this occurs, the `semncnt` value of the specified semaphore is decremented by 1, and the calling process resumes execution as described under the `sigaction()` function.
2. If `sem_op` is a positive integer and the calling process has write permission, the value of `sem_op` is added to `semval` and, if `(sem_flg & SEM_UNDO)` is non-zero, the value of `sem_op` is subtracted from the `semadj` value of the calling process for the specified semaphore.
  3. If `sem_op` is 0 and the calling process has read permission, one of the following will occur:
    - If `semval` is 0, `semop()` will return immediately.
    - If `semval` and `(sem_flg & IPC_NOWAIT)` are both non-zero, `semop()` will return immediately.
    - If `semval` is non-zero and `(sem_flg & IPC_NOWAIT)` is 0, `semop()` will increment the `semzcnt` value of the specified semaphore and suspend execution of the calling process until one of the following events occurs:
      - The value of `semval` becomes 0, at which time the `semzcnt` value of the specified semaphore is decremented by 1.
      - The identifier `semid` of the semaphore for which the calling process is awaiting action is removed from the system. When this occurs, `errno` is set to `EIDRM` and the value `-1` is returned.
      - The calling process receives a signal that is to be caught. When this occurs, the `semzcnt` value of the specified semaphore is decremented by 1, and the calling process resumes execution as described under `sigaction()`.

Upon 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.

When threads are used, the functionality of `semop` changes in the following aspects:

Execution of semaphore operations:

Regarding 1. If *semval* is smaller than the absolute value of *sem\_op* and (*sem\_flg* & `IPC_NOWAIT`) is equal to 0, `semop()` increments the value of *semncnt* of the specified semaphore by 1 and the calling thread is stopped until one of the following conditions is met:

- The value of *semval* is greater than or equal to the absolute value of *sem\_op*. When this occurs, the value of *semncnt* of the specified semaphore is decremented by 1, the absolute value of *sem\_op* is subtracted from *semval* and if (*sem\_flg* & `SEM_UNDO`) is not equal to 0, the absolute value of *sem\_op* is added to the *semadj* value of the calling process for the specified semaphore.
- The *semid* identifier for which the calling thread is waiting for an operation is deleted from the system. In this case `errno` is set to `EIDRM` and -1 is returned.
- The calling thread receives a signal that must be trapped. In this case the value of *semncnt* of the specified semaphore is decremented by 1 and the calling thread continues execution in the manner described for the `sigaction()` function.

Regarding 3. If *semval* is not equal to 0 and (*sem\_flg* & `IPC_NOWAIT`) is equal to 0, `semop()` increments the value of *semzcnt* of the specified semaphore by 1 and the calling thread is stopped until one of the following events occur:

- *semval* assumes the value 0. After that the value of *semzcnt* of the specified semaphore is decremented by 1.
- The *semid* identifier for which the calling thread is waiting for an operation is deleted from the system. In this case `errno` is set to `EIDRM` and -1 is returned.

The calling thread receives a signal that must be trapped. In this case the value of *semzcnt* of the specified semaphore is decremented by 1 and the calling thread continues execution in the manner described for the `sigaction()` function.

#### Return value

- |    |                                                                   |
|----|-------------------------------------------------------------------|
| 0  | if successful.                                                    |
| -1 | if unsuccessful. <code>errno</code> is set to indicate the error. |

#### Errors

`semop()` will fail if:

- |        |                                                                                                                                     |
|--------|-------------------------------------------------------------------------------------------------------------------------------------|
| E2BIG  | The value of <i>nsops</i> is greater than the system-imposed maximum value.                                                         |
| EACCES | The process does not have the required access permission for the command to be executed (see section "Error handling" on page 126). |
| EAGAIN | The operation would result in suspension of the calling process, but ( <i>sem_flg</i> & <code>IPC_NOWAIT</code> ) is non-zero.      |



|        |                                                                                                                                                                                                                   |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EFBIG  | The value of <code>sem_num</code> is less than 0 or greater than or equal to the number of semaphores in the set associated with <code>semid</code> .                                                             |
| EIDRM  | The semaphore identifier <code>semid</code> was removed from the system.                                                                                                                                          |
| EINTR  | <code>semop()</code> was interrupted by a signal.                                                                                                                                                                 |
| EINVAL | The value of <code>semid</code> is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a <code>SEM_UNDO</code> would exceed the system-imposed limit. |
| ENOSPC | The system-specific limit on the maximum number of individual processes requesting a <code>SEM_UNDO</code> would be exceeded.                                                                                     |
| ERANGE | An operation would cause a <code>semval</code> or a <code>semadj</code> to exceed the maximum value for the system.                                                                                               |

**Notes** The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

**See also** `exec`, `exit()`, `fork()`, `semctl()`, `semget()`, `sys/sem.h`, section "Interprocess communication" on page 112.

## setbuf - assign buffering to stream

Syntax `#include <stdio.h>`

```
void setbuf(FILE *stream, char *buf);
```

Description `setbuf()` may be used after the stream pointed to by *stream* has been assigned to an open file but before any other operation has been performed on the stream. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer.

The buffer size is not limited; however, the constant `BUFSIZ` (see `stdio.h`) is typically a good buffer size:

```
char buf[BUFSIZ];
```

If *buf* is not a null pointer, the following function calls are equivalent:

```
setbuf(stream, buf)
setvbuf(stream, buf, _IOFBF, BUFSIZ)
```

If *buf* is a null pointer, input and output are unbuffered, and the following calls are equivalent:

```
setbuf(stream, buf)
setvbuf(stream, buf, _IONBF, BUFSIZ)
```

*BS2000*

If *buf* is a null pointer, the buffer assigned by the system is used.

In contrast to `setvbuf()`, `setbuf()` has no return value.

Notes A common source of error is to use an "automatic" variable (i.e. a variable of storage class `auto`) as the buffer in a program block and then fail to close the file in the same block.

Since a portion of *buf* is required for internal administration data of the stream, *buf* will contain less than *size* bytes when full. It is therefore preferable to use `setvbuf()` with automatically assigned buffers.

`setbuf()` is executed for the file assigned to *stream*. This can be a POSIX file or a *BS2000* file.

*BS2000*

If the blocking factor is explicitly defined with the `BUFFER-LENGTH` parameter of the `SET-FILE-LINK` command, the size of the area must correspond to this defined blocking size. □

See also `fopen()`, `setvbuf()`, `stdio.h`, section "Streams" on page 77.

## setcontext - modify user context

Syntax        `#include <ucontext.h>`  
              `int setcontext(const ucontext_t *ucp);`

Description See `getcontext()`

## setgid - set group ID of process

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h>`

`int setgid(gid_t gid);`

Description If the process has appropriate privileges, `setgid()` 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 the saved set-group-ID, `setgid()` 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.

Return value

0 if successful.

-1 if unsuccessful. `errno` is set to indicate the error.

Errors `setgid()` will fail if:

EINVAL The value of *gid* is invalid and is not supported.

EPERM The process does not have appropriate privileges and *gid* does not match the real group ID or the saved set-group-ID.

Notes At login, the real user ID, effective user ID, and saved set-user-ID of the login process are set to the user ID of the user responsible for creating the process. The real group ID, effective group ID, and saved set-group-ID of the login process are likewise set to the group ID of the user responsible for creating the process.

When a process calls `exec()` to execute a file, the user and/or group IDs associated with the process may change. If the file executed is a 'set-user-ID' file, the effective user ID and saved set-user-ID of the process are set to the user of the file executed. If the file executed is a 'set-group-ID' file, the effective group ID and saved set-group-ID of the process are set to the group of the file executed. If the file executed is not a 'set-user-ID' or 'set-group-ID' file, the effective user ID, saved set-user-ID, effective group ID, and saved set-group-ID are not changed.

See also `exec`, `getgid()`, `setuid()`, `sys/types.h`, `unistd.h`.

## setgrent - reset file position indicator to beginning of group file

Syntax        `#include <grp.h>`  
              `void setgrent (void);`

Description See `endgrent()`.

## setitimer - set interval timer

Syntax        `#include <sys/time.h>`  
              `int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);`

Description See `getitimer()`.

## \_setjmp - set label for non-local jump (without signal mask)

Syntax        `#include <setjmp.h>`  
              `int _setjmp(jmp_buf env);`

Description See `_longjmp()`.

## setjmp - set label for non-local jump

Syntax `#include <setjmp.h>`

```
int setjmp(jmp_buf env);
```

Description `setjmp()` saves the current calling environment (address in the C runtime stack, program counter, register contents) in its *env* argument for later use by the `longjmp()` function. `setjmp()` is implemented as a macro in the POSIX subsystem; it may be implemented as a function in other X/Open-conformant systems.

If a macro definition is suppressed in order to access an existing function, or defines a program or an external identifier with the name `setjmp`, the behavior is undefined.

`setjmp()` is only meaningful in combination with the `longjmp()` function: these two functions can be combined to implement non-local jumps, i.e. jumps from any given function to any other active function. A `longjmp` call restores the calling environment saved by `setjmp()` and then resumes program execution (see also `longjmp()`).

*env* is the array in which `setjmp()` stores the current program state. The type `jmp_buf` is defined in the header `setjmp.h`.

All accessible objects will have the same values as when `longjmp()` was called, except for the values of "automatic" objects, which are undefined under the following conditions:

- They are local to the function containing the corresponding `setjmp` call.
- They are not of type `volatile`.
- They are changed between the `setjmp` and `longjmp` calls.

`setjmp()` should only be used in one of the following contexts:

- as the entire controlling expression of a selection or iteration statement, e.g.:

```
if (setjmp(env)) ...
```
- as one operand of a relational operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement, e.g.:

```
if (setjmp(env) == 0) ...
```
- as the operand of a unary "!" operator with the resulting expression being the entire controlling expression of a selection or iteration statement, e.g.:

```
if (!setjmp(env)) ...
```
- as the entire expression of an expression statement (possibly cast to `void`):

```
void: (void)setjmp(env);
```

## Return value

- 0 on successful return from a direct invocation of `sigset`.
- ≠ 0 if the return is from a call to `longjmp()`. In this case the return value corresponds to the value of the *val* argument of the `longjmp` call.

**Notes** In general, `sigsetjmp()` is more suitable than `setjmp()` for handling errors and signals which occur in low-level subroutines.

**See also** `longjmp()`, `sigsetjmp()`, `setjmp.h`.

## setkey - set encoding key

**Syntax** `#include <stdlib.h>`  
`void setkey(const char *key);`

**Description** `setkey()` provides access to an encoding algorithm. *key* is a character array of length 64 bytes containing only bytes with numerical values of 0 and 1. This string is divided into groups of 8, where the low-order bit in each group is ignored. This gives a 56-bit key that is recorded. This is the key that will be used by the algorithm to encode the string *block* passed to the `encrypt()` function.

**Notes** Since `setkey()` does not return a value, applications wishing to check for errors should set `errno` to 0, call `setkey()`, then test `errno` and, if it is non-zero, assume that an error has occurred.

**See also** `crypt()`, `encrypt()`, `stdlib.h`.

## setlocale - set or query locale

Syntax `#include <locale.h>`

```
char *setlocale(int category, const char *locale);
```

Description `setlocale()` can be used to change a part of the locale, as specified by *category* and *locale*, or to change or query the entire current locale or portions thereof. The following constant names, which are assigned to a database, may be specified for *category*:

LC\_ALL affects the entire locale (see section "Locale" on page 53).

*BS2000*

The locale component LC\_MESSAGES is not supported for BS2000 functionality (see section "Scope of the supported C library" on page 18). □

LC\_COLLATE affects the behavior of regular expressions and of string collation functions.

LC\_CTYPE affects the behavior of regular expressions, character-handling functions, and wide-character (multi-byte) functions.

LC\_MESSAGES affects the format of message strings.

*BS2000*

This component of the locale is not supported for BS2000 functionality (see section "Scope of the supported C library" on page 18). □

LC\_MONETARY affects the monetary formatting information returned by `localeconv()`.

LC\_NUMERIC affects the radix character for formatted input/output functions, string conversion functions, and of the non-monetary formatting information returned by `localeconv()`.

LC\_TIME affects the behavior of time conversion functions.

The behavior of `nl_langinfo()` is also affected by the settings for *category*.

*locale* is a pointer to a character string containing the required settings for *category*. In addition, the following preset values are defined for all settings of *category*:

"POSIX" specifies the minimal environment for the programming language C; this is called the **POSIX locale**. If `setlocale()` is not invoked, the POSIX locale is the default.

"C" same as "POSIX", but called the **C locale**.

" " specifies a language-dependent environment, which corresponds to the environment variables LC\_\* and LANG associated with the value of *category*.

Null pointer is used to instruct the `setlocale()` function to query the current locale and to return its name.



If threads are used, then the function affects the process or a thread in the following manner: If the process is multithreaded, then the change to the locale affects all threads of the process.

**BS2000**

- "V1CTYPE" In contrast to the C locale, the characters X'8B', X'8C', X'8D' are treated as lowercase letters, the characters X'AB', X'AC', X'AD' as uppercase letters, and the characters X'CO' and X'DO' as special characters. In the "C" locale, all these characters are treated as control characters.
- "V2CTYPE" In contrast to the C locale, the collating sequence is set to correspond to the values of the EBCDIC character set.
- "GERMANY" This setting specifies the usual conventions for German-speaking countries.
- "De.EDF04F" Country-specific locale whose conversion table is based on ASCII code ISO 8859-15 ASCII code or EDF04F EBCDIC code and that supports the "DM" currency in the category LC\_MONETARY.
- "De.EDF04F@euro" Country-specific locale whose conversion table is based on ASCII code ISO 8859-15 ASCII code or EDF04F EBCDIC code and that supports the "Euro" currency in the category LC\_MONETARY.

The strings are preset in the header file `locale.h` as follows:

| Symbolic constant  | Default value    |
|--------------------|------------------|
| LC_C_C             | "POSIX"          |
| LC_C_C             | "C"              |
| LC_C_DEFAULT       | " "              |
| LC_C_V1CTYPE       | "V1CTYPE"        |
| LC_C_V2CTYPE       | "V2CTYPE"        |
| LC_C_GERMANY       | "GERMANY"        |
| LC_C_DeEDF04F      | "De.EDF04F"      |
| LC_C_DeEDF04F@euro | "De.EDF04F@euro" |



**Return value**

String that indicates the current locale for *category*  
 if *locale* is not a null pointer and `setlocale()` is completed successfully, or  
 if *locale* is a null pointer. The locale is not changed.

Null pointer if `setlocale()` fails. The locale is not changed.

A subsequent call to `setlocale()` with the returned string and its associated category will restore that part of the locale. The string returned must not be modified by the program, but may be overwritten by a subsequent call to `setlocale()`.

**Notes** The following program statements show how a program can initialize the locale for a language, while selectively modifying it so that regular expressions and string operations can be applied to text recorded in a different language:

```
setlocale(LC_ALL, "De");
setlocale(LC_COLLATE, "Fr@dict");
```

Internationalized programs must call the `setlocale()` function to take a specific language into account. This can be done by calling `setlocale()` as follows:

```
setlocale (LC_ALL, "");
```

This call uses the settings of the environment variables to initialize the locale. Changing the setting of `LC_MESSAGES` has no effect on message catalogs that are already opened by calls to `catopen()`.

### *BS2000*

When a program is started, the pointer vector `environ` is constructed from the variables stored in `SYSPOSIX.name`. If `setlocale()` is called with the null string "" as the locale, the environment variables stored in this vector and their values are taken into account. If the queried environment variable is not present, the corresponding value from the POSIX locale applies. □

User-specific locales may be implemented in addition to the predefined locales and can be selected using `setlocale()` (see section "Locale" on page 53).

**See also** `catopen()`, `ctime()`, `ctype()`, `environ`, `exec`, `getdate()`, `gettext()`, `isalnum()`, `isalpha()`, `iscntrl()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `localeconv()`, `mblen()`, `mbstowcs()`, `mbtowc()`, `n1_langinfo()`, `printf()`, `scanf()`, `strcoll()`, `strerror()`, `strfmon()`, `strtime()`, `strtod()`, `strxfrm()`, `tolower()`, `toupper()`, `towlower()`, `towupper()`, `wscoll()`, `wctod()`, `wctombs()`, `wcsxfrm()`, `wctomb()`, `langinfo.h`, `locale.h`, section "Locale" on page 53.

## setlogmask - set log priority mask

**Syntax** `#include <syslog.h>`

```
int setlogmask(int maskpri);
```

**Description** See `closelog()`

## setpgid - set process group ID for job control

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h>`

`int setpgid(pid_t pid, pid_t pgid);`

**Description** `setpgid()` is used either to join an existing process group or create a new process group within the session of the calling process. If *pgid* is equal to *pid*, the process becomes a process group leader. If *pgid* is not equal to *pid*, the process becomes a member of an existing process group. The process group ID of the session leader does not change. Upon successful completion, the process group ID of the process with the process ID that matches *pid* is set to *pgid*.

If *pid* is 0, the process ID of the calling process is used.

If *pgid* is 0, the process group ID of the specified process is used.

**Return value**

0 if successful.

-1 if unsuccessful. `errno` is set to indicate the error.

**Errors** `setpgid()` will fail if:

**EACCES** The value of *pid* 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 *pgid* is less than 0 or not supported by the implementation.

**EPERM** The process specified by *pid* is a session leader, or the value of *pid* 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, or the value of *pgid* is valid but does not match the process ID of the process specified by *pid*, and there is no process with a process group ID that matches the value of *pgid* in the same session as the calling process.

**ESRCH** The value of *pid* does not match the process ID of the calling process or of a child process of the calling process.

**See also** `exec`, `getpgrp()`, `setsid()`, `tcsetpgrp()`, `sys/types.h`, `unistd.h`.

## setpgrp - set process group ID

Syntax `#include <unistd.h>`  
`pid_t setpgrp (void);`

Description If the calling process is not already a session leader, `setpgrp()` sets the process group ID and the session number of the calling process to the process ID of the calling process and releases the controlling terminal of the calling process.

The function does not have any effect if the calling process is a session leader.

Return value `setpgrp()` returns the value of the new process group ID.

See also `exec`, `fork()`, `getpid()`, `getsid()`, `kill()`, `setsid()`, `unistd.h`.

## setpriority - set process priority

Syntax `#include <sys/resource.h>`  
`int setpriority(int which, id_t who, int priority);`

Description See `getpriority()`.

## setpwent - delete pointer to search user catalog

Syntax `#include <pwd.h>`  
`void setpwent(void);`

Description See `endpwent()`.

## setregid - set real and effective group IDs

**Syntax**      `#include <unistd.h>`

```
int setregid(gid_t rgid, gid_t egid);
```

**Description** `setregid()` is used to set the real and the effective group IDs of the calling process. If *rgid* is -1, the real group ID (GID) is not changed; if *egid* is -1, the effective GID is not changed. The real and effective GIDs can be set to different values in the same call.

If the effective user ID of the calling process matches the superuser, the real GID and the effective GID can be set to any permissible value.

If the effective user ID of the calling process does not match the superuser, either the real GID can be set to the saved “set-GID” from `execv()`, or the effective GID can be set to either the saved “set-GID” or the real GID.

If a process for setting the GID sets its effective GID to its real GID, it can still reset its effective GID to the saved “set-GID”.

Both when the real GID is changed (i.e. if *rgid* is not -1) and when the effective GID is changed into a value that does not match the real GID, the saved “set-GID” is set to the same as the new effective GID.

If the current value of the real GID is changed, the old value from the group access list is deleted (see `getgroups()`), if it is entered in the list, and the new value is added to the group access list if it does not already exist and as long as this does not cause the number of groups in this NGROUPS list to be exceeded, as defined in the `/usr/include/sys/param.h` file.

**Return value**

|    |                                                                     |
|----|---------------------------------------------------------------------|
| 0  | if executed successfully                                            |
| -1 | if an error occurs. <code>errno</code> is set to indicate the error |

**Errors**      `setregid()` will fail if:

|        |                                                                                                                                                                                                                                                  |
|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EINVAL | The value of <i>rgid</i> or <i>egid</i> is invalid or outside the permitted value range.                                                                                                                                                         |
| EPERM  | The effective user ID of the calling process does not match the superuser, and a different modification was specified, i.e. something other than changing the real GID into the saved “set-GID” or the effective GID into the real or saved GID. |

**See also**      `exec()`, `getuid()`, `setuid()`, `setreuid()`, `unistd.h`.

## setreuid - set real and effective user IDs

Syntax `#include <unistd.h>`

```
int setreuid(uid_t ruid, uid_t euid)
```

Description `setreuid()` is used to set the real and the effective user IDs of the calling process. If *ruid* is -1, the real user ID is not changed; if *euid* is -1, the effective user ID is not changed. The real and effective user IDs can be set to different values in the same call.

If the effective user ID of the calling process matches the superuser, the real user ID and the effective user ID can be set to any permissible value.

If the effective user ID of the calling process does not match that of the superuser, either the real user ID can be set to the effective user ID, or the effective user ID can be set to either the saved "set-user-ID" from `execv` or the real user ID.

If a process for setting the user ID (UID) sets its effective user ID to its real user ID, it can still reset its effective user ID to the saved "set-user-ID".

Both when the real user ID is changed (i.e. if *ruid* is not -1) and when the effective user ID is changed to a value that does not match the real user ID, the saved "set-user-ID" is set to the same as the new effective user ID.

Return value

|    |                                                                     |
|----|---------------------------------------------------------------------|
| 0  | if executed successfully                                            |
| -1 | if an error occurs. <code>errno</code> is set to indicate the error |

Errors `setreuid()` will fail if:

|        |                                                                                                                                                                                                                                                                            |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EINVAL | The value of the <i>ruid</i> or <i>euid</i> argument is invalid or outside the permitted value range.                                                                                                                                                                      |
| EPERM  | The effective user ID of the calling process does not match that of the superuser, and a different modification was specified, i.e. something other than changing the real user ID to the effective user ID or the effective user ID into the real or saved "set-user-ID". |

See also `getuid()`, `setuid()`, `unistd.h`

## setrlimit - set resource limit

**Name**        **setrlimit, setrlimit64**

**Syntax**      `#include <sys/resource.h>`

```
int setrlimit (int resource, const struct rlimit *rlp);  
int setrlimit64 (int resource, const struct rlimit64 *rlp);
```

**Description** See `getrlimit()`.

## setsid - create session and set process group ID

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h>`

`pid_t setsid(void);`

**Description** The `setsid()` function creates a new session, unless the calling process is process group leader. Following the return of this function, the calling process will be the session leader of this new session, the process group leader of a new process group, and will have no controlling terminal. The process group ID of the calling process is set to the process ID of the calling process. The calling process will be the only process in the new process group and the only process in the new session.

**Return value**

Process group ID of the calling process  
if successful.

`(pid_t) -1` if unsuccessful. `errno` is set to indicate the error.

**Errors**

`setsid()` will fail if:

**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.

**Notes**

If the calling process is the last component of a pipeline started by a job control shell, the shell may make the calling process the process group leader. The other processes of the pipeline become members of that process group. In this case, the call to `setsid()` will fail. A process that calls `setsid()` and expects to be part of a pipeline should therefore always execute a `fork()` first; the parent process should exit, and the child process should call `setsid()`, thus ensuring that the process will work reliably regardless of whether or not it is called by a job-control shell (see the manual "POSIX Basics (BS2000/OSD)" and the manual "POSIX Commands (BS2000/OSD)").

**See also** `setpgid()`, `sys/types.h`, `unistd.h`.



## setstate - pseudo-random numbers

**Syntax**      `#include <stdlib.h>`  
                `char *setstate(const char *state);`

**Description** See `initstate()`.

## setuid - set user ID

**Syntax**      `#include <unistd.h>`  
                *Optional*  
                `#include <sys/types.h>`  
                `int setuid(uid_t uid);`

**Description** If the process has appropriate privileges, 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 the saved set-user-ID, `setuid()` sets the effective user ID to *uid*. The real user ID and saved set-user-ID remain unchanged.

**Return value**

|    |                                                                   |
|----|-------------------------------------------------------------------|
| 0  | if successful.                                                    |
| -1 | if unsuccessful. <code>errno</code> is set to indicate the error. |

**Errors**      `setuid()` will fail if:

|       |                                                                                                                           |
|-------|---------------------------------------------------------------------------------------------------------------------------|
| EPERM | The process does not have appropriate privileges and <i>uid</i> does not match the real user ID or the saved set-user-ID. |
|-------|---------------------------------------------------------------------------------------------------------------------------|

**Notes**      `setuid()` is frequently used to relinquish privileges that are no longer needed in programs that have the s-bit for the owner set (especially `root`). Such programs often need the privileges granted by the s-bit only for very specific tasks. When the privileges are no longer required, they can be relinquished by a call in the form given below:

```
erg = setuid(getuid());
```

**See also**      `setpgid()`, `sys/types.h`, `unistd.h`.

## setutxent - reset pointer to utmpx file

Syntax `#include <utmpx.h>`  
`void setutxent (void);`

Description See `endutxent()`.

## setvbuf - assign buffering to stream

Syntax `#include <stdio.h>`  
`int setvbuf(FILE *stream, char *buf, int type, size_t size);`

Description `setvbuf()` may be used after the stream pointed to by *stream* has been associated with an open file but before any other operation has been performed on the stream. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is a null pointer, all I/O is unbuffered.

*type* determines how *stream* is to be buffered, as follows:

|                     |                                    |
|---------------------|------------------------------------|
| <code>_IOFBF</code> | Full buffering of input and output |
| <code>_IOLBF</code> | Line buffering                     |
| <code>_IONBF</code> | Unbuffered input and output        |

If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by `setvbuf()`.

*size* specifies the size of the *buf* array.

The contents of the *buf* array at any given time are indeterminate.

Return value

|          |                                                                                                                                              |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------|
| 0        | if successful.                                                                                                                               |
| $\neq 0$ | if an invalid value was specified for <i>type</i> or if the request cannot be satisfied.<br><code>errno</code> is set to indicate the error. |

Errors `setvbuf()` will fail if:

|       |                                                            |
|-------|------------------------------------------------------------|
| EBADF | The file descriptor underlying <i>stream</i> is not valid. |
|-------|------------------------------------------------------------|

- Notes** A common source of error is to use an "automatic" variable (i.e. a variable of storage class `auto`) as the buffer in a program block and then fail to close the file in the same block.
- Since a portion of *buf* is required for internal administration data of the stream, *buf* will contain less than *size* bytes when full. It is therefore preferable to use `setvbuf()` with automatically allocated buffers.
- Allocating a buffer of *size* bytes with `setvbuf()` does not necessarily imply that all of *size* bytes will be used for the buffer area.
- Applications should note that many implementations only provide line buffering on input from terminal devices.
- `setvbuf()` is executed for the file that is assigned to `stream`. This file can be either a POSIX file or a BS2000 file.
- BS2000*
- If the blocking factor is explicitly defined with the `BUFFER-LENGTH` parameter of the `SET-FILE-LINK` command, the size of the area must correspond to this defined blocking size. □
- See also** `fopen()`, `setbuf()`, `stdio.h`, section "Streams" on page 77.

## shmat - shared memory attach operation

```
#include <sys/shm.h>
```

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

**Description** `shmat()` attaches the shared memory segment designated by the shared memory identifier `shmid` to the data segment of the calling process. The location at which the segment is attached is determined by the following criteria:

- If `shmaddr` is equal to 0, the segment is attached at the first free address found by the system.
- If `shmaddr` and  $(shmflg \& SHM\_RND)$  are not equal to 0, the segment is attached at the address given by  $(shmaddr - ((ptrdiff\_t)shmaddr \% SHMLBA))$ . (The character `%` is the C-language remainder operator.)
- If `shmaddr` is not equal to 0 and  $(shmflg \& SHM\_RND)$  is equal to 0, the segment is attached at the address specified with `shmaddr`.
- If  $(shmflg \& SHM\_RDONLY)$  is not equal to 0 and the calling process has read permission, the segment is attached for reading.
- If  $(shmflg \& SHM\_RDONLY)$  is not equal to 0 and the calling process has read and write permission, the segment is attached for reading and writing.

The following symbolic names are defined in the header file `sys/shm.h`:

| Name       | Description                                           |
|------------|-------------------------------------------------------|
| SHMLBA     | Multiple of the address of the lower segment boundary |
| SHM_RDONLY | Attach only for reading                               |
| SHM_RND    | Round up attachment address                           |

### Return value

Start address of the data segment for the shared memory area if successful. The value of `shm_nattach` is incremented in the data structure associated with the shared memory ID.

- 1 if an error occurs. The shared memory segment is not attached. `errno` is set to indicate the error.

### Errors

`shmat()` will fail if:

- EACCES The calling process is denied the access permissions required for the operation.

|        |                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EINVAL | The value of <i>shmid</i> is not a valid shared memory ID, or the value of <i>shmaddr</i> is not equal to 0 and the value of $(shmaddr - ((ptrdiff_t) shmaddr \% SHMLBA))$ is an invalid address for attaching shared memory, or the value of <i>shmaddr</i> is not equal to 0, $(shmflg \& SHM\_RND)$ is equal to 0 and the value of <i>shmaddr</i> is an invalid address for attaching shared memory. |
| EMFILE | The number of attached shared memory segments for the calling process would exceed the system-imposed limit.                                                                                                                                                                                                                                                                                            |
| ENOMEM | The available data space is not large enough to accommodate the shared memory segment.                                                                                                                                                                                                                                                                                                                  |

**Notes** The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

**See also** `exec`, `exit()`, `fork()`, `shmctl()`, `shmdt()`, `shmget()`, `sys/shm.h`, section "Inter-process communication" on page 112.

## shmctl - shared memory control operations

**Syntax**      `#include <sys/shm.h>`

```
int shmctl(int shmid, int cmd, struct shmctl_ds *buf);
```

**Description** `shmctl()` provides a number of shared memory control operations, as specified by *cmd*. The following values for *cmd* are available:

**IPC\_STAT**      Enter the current values of all members of the `shmctl_ds` data structure associated with *shmid* into the structure pointed to by *buf*. The format of the structure is defined in `sys/shm.h`.

**IPC\_SET**      Set the values of the following members of the `shmctl_ds` data structure associated with *shmid* to the corresponding values from the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode      /* only the low-order 9 bits */
```

`IPC_SET` can only be executed by a process that has an effective user ID equal to that of a process with appropriate privileges or to the value of `shm_perm.cuid` or `shm_perm.uid` in the `shmctl_ds` data structure associated with *shmid*.

**IPC\_RMID**      Remove the shared memory identifier specified by *shmid* from the system as well as the shared memory segment and the `shmctl_ds` data structure associated with it. `IPC_RMID` can only be executed by a process that has an effective user ID equal to that of a process with appropriate privileges or to the value of `shm_perm.cuid` or `shm_perm.uid` in the `shmctl_ds` data structure associated with *shmid*.

**Return value**

0                    if successful.

-1                   if an error occurs. `errno` is set to indicate the error.

**Errors**          `shmctl()` will fail if:

**EACCES**          *cmd* is equal to `IPC_STAT` and the calling process does not have read permission.

*Extension*

**EFAULT**          *msgp* points to an invalid address. □

**EINVAL** The value of *shmid* is not a valid shared memory identifier, or the value of *cmd* is not a valid command, or *cmd* is `IPC_SET` and `shm_perm.uid` or `shm_perm.gid` is invalid.

*Extension*

**ENOMEM** Not enough memory is available.

**EPERM** *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*.

**Notes** The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

**See also** `shmat()`, `shmdt()`, `shmget()`, `sys/shm.h`, section "Interprocess communication" on page 112.

## shmdt - shared memory detach operation

**Syntax**      `#include <sys/shm.h>`  
`int shmdt(const void *shmaddr);`

**Description** `shmdt()` detaches the shared memory segment located at the address specified with `shmaddr` from the data segment of the calling process.

### *Restriction*

In this version of the POSIX subsystem, a shared memory area can only exist if it is attached to a process. The behavior of `shmdt()` therefore deviates from XPG4 in the following respect: when the last process has detached itself from a shared memory area, the memory area is released. The administration data for the memory area is, however, retained by the POSIX kernel. If another process subsequently attaches itself to the same shared memory area, the earlier contents are lost. □

### **Return value**

0              if successful. `shmdt()` decrements the value of `shm_nattach` in the data structure associated with the shared memory ID.

-1             if an error occurs. The shared memory segment is not detached. `errno` is set to indicate the error.

**Errors**      `shmdt()` will fail if:

`EINVAL`      The value of `shmaddr` is not the data segment starting address of a shared memory segment.

**Notes**      The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

**See also**    `exec`, `exit()`, `fork()`, `shmat()`, `shmctl()`, `shmget()`, `sys/shm.h`, section "Inter-process communication" on page 112.



## shmget - create shared memory segment

**Syntax**      `#include <sys/shm.h>`  
                  `int shmget(key_t key, int size, int shmflg);`

**Description** `shmget()` returns the shared memory identifier associated with *key*.

A shared memory identifier, associated data structure and shared memory segment of at least *size* bytes (see `sys/shm.h`) are created for *key* if one of the following conditions is true:

- The argument *key* has the value `IPC_PRIVATE`.
- The argument *key* does not already have a shared memory identifier associated with it and  $(shmflg \ \& \ IPC\_CREAT)$  is not equal to 0.

Upon creation, the 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 to the effective user/group ID of the calling process.
- The 9 low-order bits of `shm_perm.mode` are set equal to the 9 low-order bits of *shmflg*. The argument `shm_segsz` is set to the value of *size*.
- The values of `shm_lpid`, `shm_nattch`, `shm_atime` and `shm_dtime` are set equal to 0.
- The current time is entered for `shm_ctime`.

### Return value

Shared memory identifier  
                  if successful. The shared memory ID is a non-negative integer.  
 -1                if an error occurs. `errno` is set to indicate the error.

### Errors

`shmget()` will fail if:

|                     |                                                                                                                                                                                                                                                                                   |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>EACCES</code> | A shared memory ID for the argument <i>key</i> exists, but the permissions specified in the 9 low-order bits of <i>shmflg</i> were not granted.                                                                                                                                   |
| <code>EEXIST</code> | A shared memory ID exists for the argument <i>key</i> , but $((shmflg \ \& \ IPC\_CREAT) \ \& \ (shmflg \ \& \ IPC\_EXCL))$ is not equal to 0.                                                                                                                                    |
| <code>EINVAL</code> | The value of <i>size</i> is less than the system-imposed minimum or greater than the system-imposed maximum, or a shared memory identifier exists for the argument <i>key</i> , but the size of the segment associated with it is less than <i>size</i> and <i>size</i> is not 0. |
| <code>ENOENT</code> | A shared memory identifier does not exist for <i>key</i> and $(shmflg \ \& \ IPC\_CREAT)$ is 0.                                                                                                                                                                                   |

|        |                                                                                                |
|--------|------------------------------------------------------------------------------------------------|
| ENOMEM | The amount of available physical memory is not sufficient to fill the request.                 |
| ENOSPC | The system-imposed limit on the maximum number of allowed shared memory IDs would be exceeded. |

**Notes** The IEEE 1003.4 Standards Committee is developing alternative interfaces for interprocess communication. Application developers who need to use interprocess communication (IPC) should design their applications so that modules using the IPC routines described here can be easily modified at a later date.

*BS2000*

Tasks with only read permission are not prevented from writing to the shared memory area using BS2000 resources. □

**See also** `shmat()`, `shmctl()`, `shmdt()`, `sys/shm.h`, section "Interprocess communication" on page 112.

## sigaction - examine and change signal handling

Syntax `#include <signal.h>`

```
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

Description `sigaction()` allows the calling process to examine and/or change the signal-handling action associated with the signal `sig`. The possible values for `sig` are defined in the header file `signal.h` (see `signal.h`).

The structure `sigaction`, which is used to describe the action to be taken, is defined in the header `signal.h` and contains at least the following members:

| Member type               | Member name             | Description                                                                               |
|---------------------------|-------------------------|-------------------------------------------------------------------------------------------|
| <code>void(*)(int)</code> | <code>sa_handler</code> | <code>SIG_DFL</code> , <code>SIG_IGN</code> or a pointer to a signal-handling function.   |
| <code>sigset_t</code>     | <code>sa_mask</code>    | Additional set of signals to be blocked during execution of the signal-handling function. |
| <code>int</code>          | <code>sa_flags</code>   | Special flags that can be used to affect the behavior of <code>sig</code> .               |

If `act` is not a null pointer, it points to a structure specifying the new action to be associated with `sig`, thus changing the current signal action. In this case, the argument `oact` must point to a structure in which the current signal action is to be stored on return from `sigaction()`.

If `act` is a null pointer, the current signal handling remains unchanged, so this call can be used to examine the current handling for a given signal. The argument `oact` may be a null pointer in this case,

`sa_handler` identifies the signal action for `sig` and may have any of the values defined as signal actions in `signal.h` (see `signal.h`).

If `sa_handler` specifies a signal-handling function, the `sa_mask` member identifies a set of signals that are added to the process signal mask before the signal-handling function is called. Note that the `SIGKILL` and `SIGSTOP` signals cannot be blocked (i.e. are not added to the signal mask by this mechanism) and that this restriction will be enforced by the system without causing an error to be indicated.

`sa_flags` can be used to change the behavior of the specified signal. The following flag bits, defined in the header `signal.h`, can be set in `sa_flags`:

`SA_NOCLDSTOP`

prevents `SIGCHLD` from being generated when a child process stops.

*Extension*

## SA\_NOCLDWAIT

If this flag bit is set and *sig* equals SIGCHLD, the system will not create zombie processes when children of the calling process exit. If the calling process subsequently executes successive `wait` calls, it will block until all of its children terminate; a value of -1 is then returned, with `errno` set to ECHILD.

## SA\_NODEFER

The signal is not automatically blocked by the system while being processed by the signal-handling function.

## SA\_RESETHAND

If this option is set and the signal is caught, the disposition of the signal will be reset to SIG\_DFL, and the signal will be blocked on entry to the signal handler (SIGILL and SIGTRAP cannot be automatically reset when delivered; the system silently enforces this restriction).

## SA\_RESTART

If this flag bit is set and the signal is caught, a system call that is interrupted by the execution of the signal-handling routine is transparently restarted by the system. Otherwise, that system call returns an EINTR error.

## SA\_SIGINFO

If this flag bit is cleared and the signal is caught, *sig* is passed as the only argument to the signal-catching function.  
If the flag is set and the signal is caught, blocked signals of type *sig* are reliably queued for the calling process, and two additional arguments are passed to the signal-catching function. If the second argument is not a null pointer, it points to a structure of type `siginfo_t` containing the reason for the signal; the third argument points to a structure of type `ucontext_t` containing the context of the receiving process at the time the signal was received. □

If *sig* is SIGCHLD and SA\_NOCLDSTOP is not set in `sa_flags`, then a SIGCHLD signal will be generated for the calling process whenever any of its child processes stop. If *sig* is SIGCHLD and SA\_NOCLDSTOP is set in `sa_flags`, no SIGCHLD signal is generated.

When a signal is caught by a signal-handling function defined by `sigaction()`, a new signal mask is calculated for the duration of the signal-handling 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 sent, including the sent signal itself. If the user-defined signal handler returns normally, the original signal mask is restored.

The current signal-handling action for *sig* remains in effect until `sigaction()` is called again or until one of the `exec` functions is called.

If the previous action (*oact*) for *sig* was established by `signal()`, the values of the structure components returned in the structure pointed to by *oact* are unspecified and, in particular, `oact->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()` via the *act* argument, handling of the signal will be as if the original call to `signal()` were repeated.

An attempt to set the action for a signal that cannot be caught or ignored to `SIG_DFL` causes an error with `errno` set to `EINVAL`.

## General notes on signal handling

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, or an invocation of `kill()`. In some circumstances, the same event generates signals for multiple processes.

Each process must ensure that a signal action is specified for each signal defined by the system (see section “Signal actions” on page 726). A signal is said to be **delivered** to a process when the prescribed 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 will remain 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 signal that is already pending is generated, it is undefined whether the signal will be delivered more than once. The order in which multiple, simultaneously pending signals are delivered to a process is unspecified.

When a **stop signal** (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`) is generated for a process, any pending signals of type `SIGCONT` for that process are discarded. Conversely, whenever `SIGCONT` is generated for a process, all pending stop signals for that process are likewise discarded. When `SIGCONT` is generated for a process that is stopped, the process is continued even if the `SIGCONT` signal is blocked or ignored. If `SIGCONT` is blocked and not ignored, it will remain pending until it is either unblocked or a stop signal is generated for the process.

## Signal actions

The following signal actions can be associated with a signal:

- `SIG_DFL`
- `SIG_IGN`
- a pointer to a signal-handling function

All signals are set to `SIG_DFL` or `SIG_IGN` (see `signal.h`) prior to entry of the `main()` routine. The signal actions prescribed by these values are as follows:

`SIG_DFL` - signal-specific default action:

- The default signal handling for supported signals is described in the `signal.h` section.
- If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a `SIGCHLD` signal will be 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 will not be delivered until the process is continued, except for `SIGKILL`, which always terminates the receiving process. A process that is a member of an orphaned process group will not be allowed to stop in response to the `SIGTSTP`, `SIGTTIN` or `SIGTTOU` signals. In cases where delivery of one of these signals would stop such a process, the signal will be 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`), will cause the pending signal to be discarded, whether or not it is blocked.

`SIG_IGN` - ignore signal:

- Delivery of the signal will have 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 will not allow the action `SIG_IGN` for the signals `SIGKILL` or `SIGSTOP`. Setting a signal action to `SIG_IGN` for a signal that is pending will cause 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 signal will be ignored.

Pointer to a signal-handling function - catch signal:

- On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. After returning from the signal-handling function, the receiving process will resume execution at the point at which it was interrupted.

- The signal-handling function is called in the form of a C function as follows:  

```
void func (int signo);
```
- *func* is the specified signal-handling function, and *signo* is the signal number of the signal being delivered.
- The behavior of a process is undefined after it returns normally from an error-handling function for a SIGFPE, SIGILL or SIGSEGV signal that was not generated by `kill()` or `raise()`.
- The system will not allow a process to catch the signals SIGKILL and SIGSTOP.
- If a process establishes a signal-handling 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.
- When signal-handling functions are invoked asynchronously with process execution, the behavior of some of the functions defined in this manual is unspecified if they are called from a signal-handling function. The following table defines a set of functions that are either **reentrant** or not interruptible by signals. These so-called **safe** functions may therefore be invoked by applications from signal-handling functions without restrictions:

|                            |                          |                            |                            |
|----------------------------|--------------------------|----------------------------|----------------------------|
| <code>access()</code>      | <code>free()</code>      | <code>raise()</code>       | <code>sysconf()</code>     |
| <code>alarm()</code>       | <code>fstat()</code>     | <code>read()</code>        | <code>tcdrain()</code>     |
| <code>calloc()</code>      | <code>getegid()</code>   | <code>rename()</code>      | <code>tcflow()</code>      |
| <code>cfgetispeed()</code> | <code>geteuid()</code>   | <code>rmdir()</code>       | <code>tcflush()</code>     |
| <code>cfgetospeed()</code> | <code>getgid()</code>    | <code>setgid()</code>      | <code>tcgetattr()</code>   |
| <code>cfsetispeed()</code> | <code>getgroups()</code> | <code>setpgid()</code>     | <code>tcgetpgrp()</code>   |
| <code>cfsetospeed()</code> | <code>getpgrp()</code>   | <code>setsid()</code>      | <code>tcsendbreak()</code> |
| <code>chdir()</code>       | <code>getpid()</code>    | <code>setuid()</code>      | <code>tcsetattr()</code>   |
| <code>chmod()</code>       | <code>getppid()</code>   | <code>sigaction()</code>   | <code>tcsetpgrp()</code>   |
| <code>chown()</code>       | <code>getuid()</code>    | <code>sigaddset()</code>   | <code>time()</code>        |
| <code>close()</code>       | <code>kill()</code>      | <code>sigdelset()</code>   | <code>times()</code>       |
| <code>creat()</code>       | <code>link()</code>      | <code>sigemptyset()</code> | <code>umask()</code>       |
| <code>dup2()</code>        | <code>lseek()</code>     | <code>sigfillset()</code>  | <code>uname()</code>       |
| <code>dup()</code>         | <code>malloc()</code>    | <code>sigismember()</code> | <code>unlink()</code>      |
| <code>execle()</code>      | <code>mkdir()</code>     | <code>signal()</code>      | <code>utime()</code>       |
| <code>execve()</code>      | <code>mkfifo()</code>    | <code>sigpending()</code>  | <code>wait()</code>        |
| <code>_exit()</code>       | <code>open()</code>      | <code>sigprocmask()</code> | <code>waitpid()</code>     |
| <code>fcntl()</code>       | <code>pathconf()</code>  | <code>sigsuspend()</code>  | <code>write()</code>       |
| <code>fork()</code>        | <code>pause()</code>     | <code>sleep()</code>       |                            |

|                          |                     |                     |
|--------------------------|---------------------|---------------------|
| <code>fpathconf()</code> | <code>pipe()</code> | <code>stat()</code> |
|--------------------------|---------------------|---------------------|

All functions not in the above table are considered to be **unsafe** with respect to signals. In the presence of signals, all X/Open-conformant functions behave as defined when called from or interrupted by a signal-handling function, with a single exception: when a signal interrupts an unsafe function and the signal-handling function calls an unsafe function, the behavior is undefined.

### Signal effects on other functions

Signals affect the behavior of the following functions if they are delivered to a process while it is executing any of these functions:

|                         |                         |                           |                          |
|-------------------------|-------------------------|---------------------------|--------------------------|
| <code>catclose()</code> | <code>fgetcwc()</code>  | <code>getgrnam()</code>   | <code>tcdrain()</code>   |
| <code>catgets()</code>  | <code>fopen()</code>    | <code>getpass()</code>    | <code>tcsetattr()</code> |
| <code>close()</code>    | <code>fputc()</code>    | <code>getpwnam()</code>   | <code>tmpfile()</code>   |
| <code>dup()</code>      | <code>fputcwc()</code>  | <code>getpwuid()</code>   | <code>wait()</code>      |
| <code>fclose()</code>   | <code>freopen()</code>  | <code>open()</code>       | <code>write()</code>     |
| <code>fcntl()</code>    | <code>fseek()</code>    | <code>pause()</code>      |                          |
| <code>fflush()</code>   | <code>fsync()</code>    | <code>read()</code>       |                          |
| <code>fgetc()</code>    | <code>getgrgid()</code> | <code>sigsuspend()</code> |                          |

This has the following consequences:

- If the action of the signal is to terminate the process, the process will be terminated and the function will not return.
- If the action of the signal is to stop the process, the process will stop until continued or terminated.
- The generation of a `SIGCONT` signal for a process causes the process to be continued at the point at which the process was stopped.
- If the associated action of the signal is to invoke a signal-handling function, the relevant signal-handling function will be invoked; in this case, the original function is said to be interrupted by the signal.
- If the signal-handling function executes a `return` statement, the behavior of the interrupted function will be as described for that function.
- Signals that are ignored will not affect the behavior of any function.
- Signals that are blocked will not affect the behavior of any function until they are delivered.



## Return value

- 0 if successful.
- 1 if an error occurs. `errno` is set to indicate the error. No new signal-handling function is defined.

Errors `sigaction()` will fail if:*Extension*

- EFAULT *act* and *oact* point outside the allocated address space of the process. □
- EINVAL *sig* is not a valid signal number, or  
an attempt was made to catch or ignore a signal that cannot be caught  
ignored, or  
an attempt was made to set the action to `SIG_DFL` for a signal that cannot  
be caught or ignored (or both).

## Notes

`sigaction()` supersedes `signal()` and should therefore be used with preference. In particular, `sigaction()` and `signal()` should not be used for the same signal in the same process.

If the same signal is registered two or more times, only the last one applies. This is especially true for signals mapped to one another. For example, the signal `SIGDVZ` is mapped to `SIGFPE`, and `SIGTIM` is mapped to `SIGVTALRM`. If a signal belonging to such a pair is registered first, and is then followed by the other, this will be treated as a repetition of the same signal.

Reentrant functions behave as described in this manual and may be used in signal-handling functions without restrictions. Applications should nonetheless consider all effects of such functions on data structures, files and process states. In particular, application writers need to consider the restrictions on interactions when interrupting `sleep()` and interactions among multiple file descriptors for a file description. `c`

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some semaphore. This manual does not address the more general problem of synchronizing access to shared data structures. Note that even the safe functions may modify the external variable `errno`; the signal-handling function may want to save and restore its value. Naturally, the same principles apply to reentrant application routines and asynchronous data access.

`siglongjmp()` is not in the list of reentrant functions. This is because the code executing after `siglongjmp()` can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use `longjmp()` and `siglongjmp()` from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either `malloc()`, `free()` or functions from `stdio.h`, all of which traditionally use

data structures in a non-reentrant manner. Since any combination of different functions using a common data structure can cause reentrancy problems, this manual does not define the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

If a signal occurs without `abort()`, `kill()` or `raise()` being called, the behavior is undefined if the signal handler calls an X/Open-conformant library function other than one of those listed in the table above or if an object of static storage duration other than a variable of type `volatile sig_atomic_t` is accessed. If such a call fails, the value of `errno` is indeterminate.

The association between the symbolic names of signal numbers and their numeric values has not been standardized. An application will be portable only if *sig* uses the symbolic names.

**See also**

`kill()`, `sigaddset()`, `sigdelset()`, `sigfillset()`, `sigemptyset()`, `sigismember()`, `sigprocmask()`, `sigsuspend()`, `signal.h`, and section "Signals" on page 111.

## sigaddset - add signal to signal set

**Syntax**      `#include <signal.h>`

```
int sigaddset(sigset_t *set, int sig);
```

**Description** `sigaddset()` adds the signal *sig* to the signal set pointed to by *set*.

**Return value**

0                    if successful.

-1                   if an error occurs. `errno` is set to indicate the error.

**Errors**        `sigaddset()` will fail if:

`EINVAL`        The value of *sig* is an invalid or unsupported signal number.

**Notes**        Applications should call `sigemptyset()` or `sigfillset()` for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` or `sigprocmask()`, the behavior is undefined.

**See also**     `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

## sigaltstack - set/read alternative stack of signal

Syntax `#include <signal.h>`

```
int sigaltstack(const stack_t *ss, stack_t *oss);
```

Description `sigaltstack()` is used to define an alternative stack in which signals can be processed. If `ss` is not zero, a pointer to a `stack_t` structure describing a stack on which the signals can be processed is expected. With `sigaction` you can specify which signals are to be handled on the alternative signal stack. The system then switches over to the signal stack for the duration of the signal-handling routine.

The `stack_t` structure contains the following components:

```
int      *ss_sp
long     ss_size
int      ss_flags
```

If `ss` is not zero, the `stack_t` structure describes an alternative signal stack, which becomes effective after the return of `sigaltstack()`. The components `ss_sp` and `ss_size` determine the base and the size of the stack. The `ss_flags` component indicates the status of the new stack and can have the following values:

`SS_DISABLE` The stack is deactivated and `ss_sp` and `ss_size` are ignored. If `SS_DISABLE` is not set, the stack will be activated.

If `oss` is not zero, on successful return from `sigaltstack` the structure contains the description of the alternative signal stack which was active before the `sigaltstack()` call. `ss_sp` and `ss_size` specify the base and the size of the stack.

The `ss_flags` component indicates the status of the stack and can have the following values:

`SS_ONSTACK` The process is currently executed with the alternative signal stack. Any attempts to modify the alternative signal stack during execution of the process will fail.

`SS_DISABLE` The alternative signal stack is currently deactivated.

The value `SIGSTKSZ` represents the number of bytes that are generally necessary for an alternative stack. The value `MINSIGSTKSZ` defines here the minimum stack size for a signal-handling routine. When computing the stack size the program should still set up this minimum value in addition, to take into account the operating system's own requirements. The constants `SS_ONSTACK`, `SS_DISABLE`, `SIGSTKSZ` and `MINSIGSTKSZ` are defined in `<signal.h>`.

## Return value

- 0 if executed successfully.
- 1 if an error occurs. `errno` is set to indicate the error.

## Errors

`sigaltstack()` will fail if:

- `EPERM` An attempt was made to modify an active stack (deactivate).
- `EINVAL` The `ss` argument is not zero and the `ss_flags` component to which `ss` points contains other flags than `SS_DISABLE`.
- `ENOMEM` The size of the alternative stack area is less than `MINSIGSTKSZ`.

## Notes

The following program excerpt is used to allocate an alternative stack area:

```
if ((sigstk.ss_sp = (char *)malloc(SIGSTKSZ)) == NULL)
    /* Error handling */;

sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, (stack_t *)0) < 0)
    perror("sigaltstack");
```

## See also

`sigaction()`, `sigsetjmp()`, `signal.h`

## sigdelset - delete signal from signal set

**Syntax**      `#include <signal.h>`  
`int sigdelset(sigset_t *set, int sig);`

**Description** `sigdelset()` deletes the signal *sig* from the signal set pointed to by *set*.

**Return value**  
0                    if successful.  
-1                   if an error occurs. `errno` is set to indicate the error.

**Errors**        `sigdelset()` will fail if:  
EINVAL            The value of *sig* is an invalid or unsupported signal number.

**Notes**        Applications should call `sigemptyset()` or `sigfillset()` for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` or `sigprocmask()`, the behavior is undefined.

**See also**      `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

## sigemptyset - initialize and empty signal set

**Syntax**      `#include <signal.h>`  
`int sigemptyset(sigset_t *set);`

**Description** `sigemptyset()` initializes the signal set pointed to by *set* in a manner that excludes all the signals defined in the system.

**Return value**

|    |                     |
|----|---------------------|
| 0  | if successful.      |
| -1 | if an error occurs. |

**Notes**      Applications should call `sigemptyset()` or `sigfillset()` for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` or `sigprocmask()`, the behavior is undefined.

**See also**      `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

## sigfillset - initialize and fill signal set

Syntax `#include <signal.h>`

```
int sigfillset(sigset_t *set);
```

Description `sigfillset()` initializes the signal set pointed to by *set* in a manner that includes all the signals defined in the system.

Return value

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

Errors `sigfillset()` will fail if:

*Extension*

EFAULT *set* specifies an invalid address.  $\square$

Notes Applications should call `sigemptyset()` or `sigfillset()` for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` or `sigprocmask()`, the behavior is undefined.

See also `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

## sighold, sigignore - add signal to signal mask / register SIG\_IGN for signal

Syntax `#include <signal.h>`

```
int sighold(int sig);
int sigignore(int sig);
```

Description See `signal()`.



## siginterrupt - change behavior of system calls in response to interrupts

**Syntax**      `#include <signal.h>`

```
int siginterrupt(int sig, int flag);
```

**Description** `siginterrupt()` is used to modify the restart behavior of system calls if the system call was interrupted by the specified signal. the function has the same effect as the following implementation:

```
siginterrupt(int sig, int flag) {
    int ret;
    struct sigaction act;
    (void) sigaction(sig, NULL, &act);
    if (flag)
        act.sa_flags &=~SA_RESTART;
    else
        act.sa_fags |= SA_RESTART;
    ret=sigaction(sig, &act, NULL);
    return ret;
}
```

**Return value**

0                    if successful.  
-1                   if an error occurs. `errno` is set to indicate the error..

**Errors**            `siginterrupt()` will fail if:

EINVAL              The *sig* argument specifies an invalid signal number.

**Notes**            `siginterrupt()` supports programs which use “historical” system interfaces. When a new portable application is written or an existing one rewritten, it should use the `sigaction()` function with the `SA_RESTART` flag instead of `siginterrupt()`.

**See also**          `sigaction()`, `signal.h`.

## sigismember - test for member of signal set

Syntax `#include <signal.h>`

```
int sigismember(const sigset_t *set, int sig);
```

Description `sigismember()` tests whether the signal *sig* is a member of the set pointed to by *set*.

Return value

- 1 upon successful completion, if the specified signal is a member of the specified set.
- 0 upon successful completion, if the specified signal is not contained in the specified set.
- 1 if an error occurs. `errno` is set to indicate the error.

Errors `sigismember()` will fail if:

`EINVAL` The value of *sig* is an invalid or unsupported signal number.

Notes Applications should call `sigemptyset()` or `sigfillset()` for each object of type `sigset_t` prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` or `sigprocmask()`, the behavior is undefined.

See also `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

## siglongjmp - execute non-local jump using signal

**Syntax**      `#include <setjmp.h>`

```
void siglongjmp(sigjmp_buf env, int val);
```

**Description** `siglongjmp()` restores the environment saved by the last invocation of `sigsetjmp()` in the same process with the same `sigjmp_buf` argument. If there is no prior invocation or if the function in which this macro was called has terminated in the interim, the behavior is undefined.

All accessible objects have the same values as of the time `siglongjmp()` was called, except that the values of automatic objects which are changed between the execution of `sigsetjmp()` and the `siglongjmp()` call are indeterminate.

Since `siglongjmp()` bypasses the usual function call and return mechanisms, it also executes correctly in contexts with interrupts, signals and their associated functions. However, if `siglongjmp()` is invoked from a nested signal handler (that is, from a function called as a result of a signal raised during another signal-handling function), the behavior is undefined.

`siglongjmp()` restores the saved signal mask if and only if the *env* argument was initialized by a call to `sigsetjmp()` with a *savemask* argument not equal to 0.

`siglongjmp()` is not thread-safe. The result of calling this function is undefined if the `jmp_buf` structure was not initialized in the calling thread.

**Return value**

0      After `siglongjmp()` is completed, program execution continues as if the corresponding execution of the `sigsetjmp()` macro had just returned the value specified by *val*. `siglongjmp()` cannot cause `sigsetjmp()` to return the value 0.

**Notes**      if *val* is 0, the corresponding `sigsetjmp()` macro returns the value 1. The distinction between `setjmp()` or `longjmp()` and `sigsetjmp()` or `siglongjmp()` is only significant for programs which use `sigaction()`, `sigprocmask()` or `sigsuspend()`.

**See also**      `longjmp()`, `setjmp()`, `sigprocmask()`, `sigsetjmp()`, `sigsuspend()`, `setjmp.h`.

## signal - examine or change signal handling

Syntax

```
#include <signal.h>

void ( *signal(int sig, void ( *func)(int)))(int);

int sighthold(int sig);

int sigignore(int sig);

int sigpause(int sig);

int sigrelse(int sig);

void ( *sigset(int sig, void ( *disp)(int)))(int);
```

Description `signal()` defines how the receipt of a signal is to be subsequently handled.

`sig` may be any signal defined by the system, except SIGKILL and SIGSTOP (see `signal.h`).

`func()` defines the signal action. The following values are possible:

- SIG\_DFL (default signal handling)
- SIG\_IGN (ignore the signal)
- Address of a signal-handling function (also called a signal handler)  
In this case, the system adds the signal `sig` to the signal mask of the calling process before the signal handler is executed. On exiting the signal-handler, the system restores the signal mask of the calling process to the existing state before the signal was received.

If `func()` points to a function, the following steps are performed in sequence when a signal occurs:

1. An equivalent of the following `signal` function is executed:

```
signal(sig, SIG_DFL);
```

If the value of `sig` in this example is SIGILL, a reset to SIG\_DFL occurs.

2. An equivalent of the following function is executed next:

```
( *func)(sig);
```

The signal-handling function `func()` may be terminated by a return statement or by an `abort()`, `exit()`, or `longjmp()` function. If `func()` executes a return statement and the value of `sig` is SIGFPE, SIGILL or SIGDVZ, the behavior is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If a signal occurs without `abort()`, `kill()` or `raise()` being called, the behavior is undefined if the signal handler calls an X/Open-conformant library function other than one of those listed in the table under `sigaction()` or if an object of static storage duration other than a variable of type `volatile sig_atomic_t` is accessed. If such a call fails, the value of `errno` is indeterminate.

At program startup, the equivalent of the following function is executed for some signals:

```
signal(sig, SIG_IGN);
```

An equivalent of the following function is executed for all other signals (see `exec`):

```
signal(sig, SIG_DFL);
```

The functions `sigset()`, `sighold()`, `sigignore()`, `sigpause()` and `sigrelse()` simplify signal management for application processes.

`sigset()` is used to modify signal handling. *sig* indicates the signal, which can be any one except `SIGKILL` and `SIGSTOP`. *disp* defines the handling of the signal, which can be `SIG_DFL`, `SIG_IGN` or the address of a signal-handling routine. If `sigset()` is used and *disp* is the address of a signal-handling routine, the system adds the signal *sig* to the signal mask of the calling process before the signal-handling routine is executed. When execution of the signal-handling routine terminates, the system resets the signal mask of the calling process to the status it had before the signal was received. If `sigset()` is used and *disp* equals `SIG_HOLD`, then *sig* is added to the signal mask of the calling process, and the signal handling remains unchanged.

`sighold()` adds *sig* to the signal mask of the calling process.

`sigrelse()` removes *sig* from the signal mask of the calling process.

`sigignore()` sets the handling of *sig* to `SIG_IGN`.

`sigpause()` removes *sig* from the signal mask of the calling process and deactivates the calling process until a signal is received.

If one of the above functions is used to set the handling of `SIGCHLD` to `SIG_IGN`, the child processes of the calling process will not generate any zombie processes when they are terminated. If the calling process waits for its child processes consecutively, it blocks until all its child processes are terminated. The value -1 is then returned and `errno` contains the error ID `ECHILD` (see `wait()`, `waitid()`, `waitpid()`).

## Return value

Value of *func()* on successful completion.

**SIG\_ERR** if an error occurs, e.g. if *sig* is not a valid signal number or *func()* points to an invalid address. *errno* is set to indicate the error.

**SIG\_HOLD** returned by *sigset()* on successful completion if the signal was blocked. If it was not blocked, *sigset()* returns the previous handling.

**SIG\_ERR** if an error occurs in *sigset()*, *errno* contains the relevant error ID.

All other functions return zero if successful. If an error occurs they return -1 and set *errno*.

## Errors

*signal()* will fail if:

**EINVAL** *sig* is an invalid signal number, or an attempt was made to catch a signal that cannot be caught, or to ignore a signal that cannot be ignored or to set the action to **SIG\_DFL** for a signal that can be neither caught nor ignored.

*BS2000*

**EFAULT** Invalid address. □

*sigset()*, *sighold()*, *sigrelse()*, *sigignore()* and *sigpause()* will fail if:

**EINVAL** *sig* is an invalid signal number, or with *sigset()* and *sigignore()* an attempt was made to catch a signal that cannot be caught or to ignore a signal that cannot be ignored.

## Notes

*sigaction()* provides a more comprehensive and reliable mechanism for controlling signals than *signal()*; new applications should therefore use *sigaction()*.

*sighold()* in conjunction with *sigrelse()* or *sigpause()* can be used to create critical program areas in which the receipt of a signal can be temporarily deactivated.

The *sigsuspend()* function can be used instead of *sigpause()* to increase the portability.

## See also

*exec*, *pause()*, *sigaction()*, *waitid()*, *signal.h*.

## siggam - variable for sign of lgamma

**Syntax**      `#include <math.h>`  
                `extern int siggam;`

**Description** See `lgamma()`.

## sigpause - remove signal from signal mask and deactivate process

**Syntax**      `#include <signal.h>`  
                `int sigpause(int sig);`

**Description** See `signal()`

**Notes**        If threads are used, then the function affects the process or a thread in the following manner: `sigpause()` deletes a signal from the signal mask and suspends the thread.

## sigpending - examine pending signals

**Syntax**      `#include <signal.h>`  
                `int sigpending(sigset_t *set);`

**Description** `sigpending()` stores the set of signals that are blocked from delivery and pending to the calling process, in the object pointed to by *set*.

**Return value**

|    |                                                                      |
|----|----------------------------------------------------------------------|
| 0  | if successful.                                                       |
| -1 | if an error occurs. <code>errno</code> is set to indicate the error. |

**Errors**        `sigpending()` will fail if:

*Extension*

|        |                                      |
|--------|--------------------------------------|
| EFAULT | <i>set</i> is not a valid pointer. □ |
|--------|--------------------------------------|

**See also**      `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `sigprocmask()`, `signal.h`.

## sigprocmask - examine or change blocked signals

Syntax `#include <signal.h>`

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Description `sigprocmask()` allows the calling process to examine and/or change its signal mask, i.e. the set of blocked signals.

If *set* is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

*how* indicates the way in which the set is to be changed, and can assume one of the following values (see also `signal.h`):

`SIG_BLOCK` The resulting set will be the union of the current set and the signal set specified by *set*.

`SIG_UNBLOCK` The resulting set will be the intersection of the current set and the complement of the signal set specified by *set*. The resulting set will be the signal set pointed to by *set*.

`SIG_SETMASK` The resulting set will correspond to the signal set specified by *set*.

If *oset* is not a null pointer, the previous mask is stored in the location pointed to by *oset*.

If *set* is a null pointer, the value of the argument *how* is not significant and the process signal mask is unchanged; thus the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to `sigprocmask()`, at least one of those signals will be delivered before the call to `sigprocmask()` returns.

It is not possible to block those signals which cannot be ignored (see `signal.h`). This is enforced by the system without causing an error to be indicated.

If any of the `SIGFPE`, `SIGILL` or `SIGSEGV` signals are generated while they are blocked, the result is undefined, unless the signal was generated by a call to `kill()` or `raise()`.

If `sigprocmask()` fails, the process signal mask is not changed.

`sigprocmask()` is not thread-safe. Use the function `pthread_sigmask()` when needed.

Return value

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error. The process signal mask is not changed.



**Errors**      `sigprocmask()` will fail if:

`EINVAL`      The value of *how* does not correspond to any of permitted value.

*Extension*

`EFAULT`      *set* or *oset* points beyond the allocated process address space. □

**See also**    `kill()`, `raise()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`,  
`sigfillset()`, `sigismember()`, `sigpending()`, `sigsuspend()`, `signal.h`.

## sigrelse - remove signal from signal mask

Syntax      `#include <signal.h>`  
`int sigrelse(int sig);`  
`void ( *sigset(int sig, void ( *disp)(int)))(int);`

Description `see signal()`.

## sigset - modify signal handling

Syntax      `#include <signal.h>`  
`void ( *sigset(int sig, void ( *func)(int)))(int);`

Description `sigset()` is used to modify signal handling.  
See `signal()`.

Notes        `sigset()` is not thread-safe.

## sigsetjmp - set label for non-local jump using signal

**Syntax**      `#include <setjmp.h>`  
                  `int sigsetjmp(sigjmp_buf env, int savemask);`

**Description** `sigsetjmp()` saves its calling environment in the argument *env* for later use by the `siglongjmp()` function. `sigsetjmp()` is implemented as a macro.

If the value of *savemask* is not equal to 0, `sigsetjmp()` will also save the current signal mask of the process as part of the calling environment. If `setjmp()` were used, this would be lost.

All accessible objects will have the same values as when `longjmp()` was called, except for the values of "automatic" objects, which are undefined under the following conditions:

- They are local to the function containing the corresponding `setjmp()` call.
- They are not of type `volatile`.
- They are changed between the `setjmp` invocation and the `longjmp` call.

`sigsetjmp()` may only be called in one of the following contexts:

- as the entire controlling expression of a selection or iteration statement, e.g.:  
     `if (sigsetjmp(env, mask)) ...`
- as one operand of a relational operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement, e.g.:  
     `if (sigsetjmp(env, mask)==0) ...`
- as the operand of a unary "!" operator with the resulting expression being the entire controlling expression of a selection or iteration statement, e.g.:  
     `if (!sigsetjmp(env, mask)) ...`
- as the entire expression of an expression statement (possibly cast to `void`), e.g.:  
     `(void) sigsetjmp(env, mask);`

If threads are used, then the function affects the process or a thread in the following manner: If the value of *savemask* is not equal to 0, `sigsetjmp()` also stores the current signal mask of the calling thread as part of the call environment.

**Return value**

|     |                                                                             |
|-----|-----------------------------------------------------------------------------|
| 0   | on successful return from a direct invocation of <code>sigsetjmp()</code> . |
| ≠ 0 | if the return is from a call to <code>siglongjmp()</code> .                 |

**Notes**      The distinction between `setjmp()` or `longjmp()` and `sigsetjmp()` or `siglongjmp()` is only significant for programs which use `sigaction()`, `sigprocmask()` or `sigsuspend()`.

**See also**    `siglongjmp()`, `signal()`, `sigprocmask()`, `sigsuspend()`, `setjmp.h`, **section "Signals"** on page 111.

## sigstack - set or query alternative stack for signal

```
#include <signal.h>
```

```
int sigstack (struct sigstack *ss, struct sigstack *oss);
```

**Description** `sigstack()` can be used to define an alternative stack, called a signal stack, in which the signals are processed. If the action of a signal indicates that the processing routine is to be executed in a signal stack (specified by a `sigaction()` call), the system checks whether the process is currently being executed in this stack. If the process is not being executed in the signal stack, the system switches over to the signal stack until the signal-handling routine terminates.

A signal stack is specified by a `sigstack` structure which contains the following elements:

```
char *ss_sp;          /* pointer of signal stack */
```

```
int ss_onstack;      /* current status */
```

`ss_sp` is the start address of the stack. If the `ss_onstack` field is non-zero, the signal stack is to be activated.

If `ss` is not a null pointer, `sigstack()` sets the status of the signal stack to the value in the `sigstack` structure to which `ss` points. The length of the stack must be at least `SIGSTKSZ` bytes. If `ss_onstack` is non-zero, the system assumes that the process is being executed in the signal stack. If `ss` is a null pointer, the status of the signal stack remains unchanged. If `oss` is not a null pointer, the current status of the signal stack is saved in the `sigstack` structure to which `oss` points.

### Return value

```
0          if successful.
-1         if an error occurs. errno is set to indicate the error.
```

### Errors

`sigstack()` will fail if:

```
EPERM     an attempt was made to modify an active stack
```

### Notes

Signal stacks are not automatically enlarged like normal stacks. Therefore, an overflow of the signal stack can cause unexpected results.

Portable applications should use `sigaltstack()` instead of `sigstack()`.

Programs should not terminate a signal-handling routine with `longjmp()` if it is executed in a stack that was set up with `sigstack()`. In certain circumstances this stack can become unusable. You are therefore advised to use the functions `siglongjmp()`, `setcontext()` or `swapcontext()` in this case.

## sigsuspend - wait for signal

Syntax `#include <signal.h>`

```
int sigsuspend(const sigset_t *sigmask);
```

Description `sigsuspend()` replaces the current signal mask of the process with the set of signals pointed to by `sigmask` and then suspends the process until delivery of a signal whose action is either to execute a signal-handling function or to terminate the process.

If the signal action is to terminate the process, then `sigsuspend()` will never return.

If the action is to execute a signal-handling function, the function will return on completion of the signal-handling function, with the signal mask restored to the set that existed prior to the `sigsuspend()` call.

It is not possible to block signals that cannot be ignored. (see `signal.h`). This is enforced by the system without causing an error to be indicated.

If threads are used, then the function affects the process or a thread in the following manner: `sigsuspend()` replaces the current signal mask of the calling thread with the signal set specified and then suspends the thread.

Return value

-1 if an error occurs. `errno` is set to indicate the error.

Since `sigsuspend()` suspends process execution indefinitely until it is interrupted by a signal, it cannot have a return value for successful completion.

Errors `sigsuspend()` will fail if:

EINTR A signal is caught by the calling process, and control is returned from the signal-handling function.

*Extension*

EFAULT `sigmask` points beyond the allocated address space of the process. □

See also `pause()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `signal.h`.

## sin - sine function

**Syntax**     `#include <math.h>`  
              `double sin(double x);`

**Description** `sin()` computes the sine of the floating-point number  $x$ , which specifies an angle in radians.

**Return value**  
      `sin(x)`           if successful. The return value is a floating-point number in the range  $[-1.0, +1.0]$ .

**See also**     `acos()`, `asin()`, `atan()`, `atan2()`, `cos()`, `sinh()`, `tan()`, `math.h`.

## sinh - hyperbolic sine function

**Syntax**     `#include <math.h>`  
              `double sinh(double x);`

**Description** `sinh()` computes the hyperbolic sine of the floating-point number  $x$ .

**Return value**  
      `sinh(x)`           if successful.  
      `+HUGE_VAL`       if an overflow occurs. `errno` is set to indicate the error.

**Errors**     `sinh()` will fail if:  
      `ERANGE`           The value of  $x$  causes an overflow.

**See also**     `acos()`, `asin()`, `atan()`, `cos()`, `cosh()`, `sin()`, `tanh()`, `math.h`.

## sleep - suspend process for fixed interval of time

**Syntax**      `#include <unistd.h>`  
`unsigned int sleep(unsigned int seconds);`

**Description** `sleep()` causes the current process to be suspended from execution until either the number of real-time seconds specified by *seconds* has elapsed or a signal is delivered to the calling process, and the action of that signal is to invoke a signal-handler or to terminate the process. The actual suspension time may be longer than *seconds* for priority reasons (i.e. due to the scheduling of other activity by the system).

If a SIGALRM signal is generated for the calling process during execution of `sleep()` and if the SIGALRM signal is being ignored or blocked from delivery, it is undefined whether `sleep()` will return when the signal is processed.

If the signal is blocked, it is likewise undefined whether it will still be pending after `sleep()` returns or whether it will be discarded.

If a SIGALRM signal is generated for the calling process during the execution of `sleep()`, except as a result of a prior call to `alarm()`, and if the SIGALRM signal is not being ignored or blocked from delivery, it is undefined whether that signal will have any effect other than forcing `sleep()` to return.

If `sleep()` is interrupted by a signal handler, the results are undefined under the following conditions:

- if the signal handler examines or changes the time at which a SIGALRM signal is to be generated
- if the signal handler changes the action associated with the SIGALRM signal
- if the signal handler changes whether the SIGALRM signal is to be blocked from delivery

If a signal handler interrupts `sleep()` and calls `siglongjmp()` or `longjmp()` to restore an environment saved prior to the `sleep()` call, both the action associated with the SIGALRM signal and the time at which the signal is to be generated are undefined. It is likewise undefined whether the SIGALRM signal will be blocked if the signal mask of the process is not also restored as part of the environment (see also `sigsetjmp()`).

If threads are used, then the function affects the process or a thread in the following manner: `sleep()` causes the current thread to be suspended until the specified time has expired or until a signal is sent to the thread.



## Return value

0 if `sleep()` returns because the specified time has elapsed.  
*seconds* minus the time already spent sleeping, i.e. the unslept time in seconds if `sleep()` returns because it was terminated prematurely by the delivery of a signal.

`sleep()` is always successful.

## Notes

Although the program is suspended by `sleep()`, time continues to run for a previously set alarm clock (see `alarm()`). This has the following effects:

1. If the previously set alarm time is less than the `sleep` time, e.g.:

```
alarm(2);  
sleep(30);
```

the alarm is triggered and the `sleep` call is ended after two "sleep" seconds have elapsed.

2. If the previously set alarm time is greater than the `sleep` time, e.g.:

```
alarm(30);  
sleep(5);
```

time continues to run on the alarm clock for 5 "sleeping" seconds. Following the `sleep` call, the alarm clock will be set at 25.

The time for which the program is actually suspended may also deviate from *seconds* for the following reasons:

- it may be up to one second shorter because "awakening" takes place at fixed 1-second intervals;
- it may be longer by any amount for priority reasons because the system has "more important" things to do.

**See also** `alarm()`, `pause()`, `sigaction()`, `unistd.h`.

## sprintf - write formatted output to string

Syntax `#include <stdio.h>`  
`int sprintf(char *s, const char *format[, arglist]);`

Description See `fprintf()`.

## sqrt - square root function

Syntax `#include <math.h>`  
`double sqrt(double x);`

Description `sqrt()` computes the square root of a non-negative floating-point number  $x$ .

Return value

|                      |                                                                         |
|----------------------|-------------------------------------------------------------------------|
| <code>sqrt(x)</code> | if $x \geq 0$ .                                                         |
| 0                    | if $x$ is negative.<br><code>errno</code> is set to indicate the error. |

Errors `sqrt()` will fail if:

|      |                               |
|------|-------------------------------|
| EDOM | The value of $x$ is negative. |
|------|-------------------------------|

See also `exp()`, `hypot()`, `log()`, `log10()`, `pow()`, `sinh()`, `math.h`.

## srand - generate pseudo-random numbers with seed

Syntax `#include <stdlib.h>`  
`void srand(unsigned int seed);`

Description `srand()` initializes the random number generator that is called by `rand()`.  
`seed` is any integer that sets the random number generator to a random number.  
The number 1 sets the random number generator to its default initial value.

See also `rand()`.

## **srandom - pseudo-random numbers**

Syntax      #include <stdlib.h>  
              void srand(unsigned int *seed*);

Description See `initstate()`.

## **srand48 - seed (double-precision) pseudo-random number generator**

Syntax      #include <stdlib.h>  
              void srand48(long int *seedval*);

Description See `drand48()`.

## **sscanf - read formatted input from string**

Syntax      #include <stdio.h>  
              int sscanf(const char \**s*, const char \**format*[, *arglist*]);

Description See `fscanf()`.

## stat - get file status

**Name** stat, stat64

**Syntax** #include <sys/stat.h>  
#include <sys/types.h>

```
int stat (const char *path, struct stat *buf);
int stat64 (const char *path, struct stat64 *buf);
```

**Description** `stat()` obtains information about the named file and writes it to the area pointed to by *buf*. *path* points to a pathname naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

*buf* is a pointer to a structure of type `stat`, as defined in the header file `sys/stat.h`, into which information concerning the file is placed.

`stat()` updates any time-related structure components, as described in the definition of "File times update" in the glossary, before writing into the `stat` structure.

The structure components `st_mode`, `st_ino`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime` and `st_mtime` will then have meaningful values for all file types. The value of the structure component `st_nlink` will be set to the number of links to the file.

There is no difference in functionality between `stat()` and `stat64()` except that `stat64()` uses a `stat64` structure.

The contents of the `stat` structure pointed to by *buf* include the following members:

```
mode_t    st_mode;    /* File mode (see mknod()) */
ino_t     st_ino;     /* Inode number (i-Node) */
dev_t     st_dev;     /* ID of device containing a
                        directory entry for this file */
dev_t     st_rdev;    /* Device ID, only defined for
                        character-special or block-special files */
nlink_t   st_nlink;   /* Number of links */
uid_t     st_uid;     /* User ID of the file's owner */
gid_t     st_gid;     /* Group ID of the file's group */
off_t     st_size;    /* File size in bytes */
time_t    st_atime;   /* Time of last access */
time_t    st_mtime;   /* Time of last data modification */
time_t    st_ctime;   /* Time of last file status change
                        The time is measured in seconds since
                        00:00:00 UTC, Jan 1, 1970 */
```

*Extension*

```
long      st_blksize; /* Preferred I/O block size */
blkcnt_t st_blocks; /* Number of st_blksize blocks allocated */
```

The `stat64` structure is defined like the structure for `stat()` with the exception of the following components:

```
ino64_t   st_ino
off64_t   st_size and
blkcnt64_t st_blocks
```

The elements of the structure have the following meanings:

|                       |                                                                                                                                                                                                                                                                               |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>st_mode</code>  | The mode of the file is defined in the system call <code>mknod()</code> .                                                                                                                                                                                                     |
| <code>st_ino</code>   | Uniquely identifies the file in a given file system. The pair <code>st_ino</code> and <code>st_dev</code> uniquely identifies regular files.                                                                                                                                  |
| <code>st_dev</code>   | Uniquely identifies the file system that contains the file.                                                                                                                                                                                                                   |
| <code>st_rdev</code>  | May be used only by administrative commands. This flag is valid only for block special or character special files and only has meaning on the system where the file was configured.                                                                                           |
| <code>st_nlink</code> | May be used only by administrative commands.                                                                                                                                                                                                                                  |
| <code>st_uid</code>   | The user ID of the file's owner.                                                                                                                                                                                                                                              |
| <code>st_gid</code>   | Group ID of the group to which the file is assigned.                                                                                                                                                                                                                          |
| <code>st_size</code>  | For regular files, this is the size of the file in bytes. It is undefined for block special or character special files.                                                                                                                                                       |
| <code>st_atime</code> | Time when file data was last accessed. Modified by the following system calls: <code>creat()</code> , <code>mknod()</code> , <code>utime()</code> and <code>read()</code> .                                                                                                   |
| <code>st_mtime</code> | Time when data was last updated. Modified by the following system calls: <code>creat()</code> , <code>mknod()</code> , <code>utime()</code> and <code>write()</code> .                                                                                                        |
| <code>st_ctime</code> | Time when the file status was last changed. Modified by the following system calls: <code>chmod()</code> , <code>chown()</code> , <code>creat()</code> , <code>link()</code> , <code>mknod()</code> , <code>unlink()</code> , <code>utime()</code> and <code>write()</code> . |

*Extension*

|                         |                                                                                                                                                       |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>st_blksize</code> | A hint as to the 'best' unit size for I/O operations. This field is not defined for block special or character special files.                         |
| <code>st_blocks</code>  | The total number of physical blocks of size 512 bytes currently used on disk. This field is not defined for block special or character special files. |

*BS2000*

With BS2000 files the following elements of the `stat` structure are set:

|                              |                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>mode_t st_mode</code>  | File mode containing the access permissions and file type.<br><br>Access permissions: here the Basic ACL is mapped to the file mode bits. The mode bits are all 0 if the file does not have Basic ACL protection.<br>File type: introduces a new file type <code>S_IFDVSBS2=X'40000000'</code> . This type, however, is not disjoint to <code>S_IFPOSIXBS2</code> . The <code>S_ISDVSBS2(mode)</code> macro can be used for querying. |
| <code>time_t st_atime</code> | Last access time, as is usual in BS2000 but in seconds since 1.1.1970 UTC).                                                                                                                                                                                                                                                                                                                                                           |
| <code>time_t st_mtime</code> | Last modification time.                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>time_t st_ctime</code> | Creation time.                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>long st_blksize</code> | Block size, 2K (i.e. 1 PAM page).                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>long st_blocks</code>  | Number of blocks occupied by the file on the disk.                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>dev_t st_dev</code>    | Contains the 4-byte <code>catid</code> .                                                                                                                                                                                                                                                                                                                                                                                              |

The two consecutive fields

|                           |                                    |
|---------------------------|------------------------------------|
| <code>uid_t st_uid</code> | and                                |
| <code>gid_t st_gid</code> | contain the 8-byte BS2000 user ID. |

All other fields are set to 0.

## Return value

|    |                                                                                     |
|----|-------------------------------------------------------------------------------------|
| 0  | if successful                                                                       |
| -1 | if an error occurs. <code>errno</code> is set to indicate the error for POSIX files |

## Notes

`stat()` is now also executed for BS2000 files.

## Errors

`stat()` and `stat64()` fail if:

`EACCES` Search permission is denied for a component of the path.

*Extension*

`EFAULT` `buf` or `path` points to an invalid address.

`EINTR` A signal was caught during the `stat()` or `lstat()` system call.

`EINVAL` The named file does not exist or the `path` argument points to an empty string.

|              |                                                                                                                                                                           |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EIO          | An I/O error occurred while reading the file system.                                                                                                                      |
| ELOOP        | Too many symbolic links were encountered in resolving <i>path</i> .                                                                                                       |
| EMULTIHOP    | Components of <i>path</i> require hops to several remote computers, but the file system does not permit this.                                                             |
| ENAMETOOLONG | The length of <i>path</i> exceeds <code>{PATH_MAX}</code> or a pathname component is longer than <code>{NAME_MAX}</code> and <code>{_POSIX_NO_TRUNC}</code> is in effect. |
| ENOLINK      | <i>path</i> refers to a remote computer to which there is no active connection.                                                                                           |
| ENOENT       | The specified file does not exist or the path is the null path.                                                                                                           |
| ENOTDIR      | A component of the path is not a directory.                                                                                                                               |
| E_OVERFLOW   | A component is too large to be stored in the structure pointed to by <i>buf</i> .                                                                                         |

**Notes** `stat()` is executed only for POSIX files

**See also** `chmod()`, `chown()`, `creat()`, `fstat()`, `lstat()`, `link()`, `mknod()`, `sys/stat.h`, `sys/types.h`.

## statvfs - read file system information

**Name** statvfs, statvfs64

**Syntax** #include <sys/statvfs.h>  
#include <sys/types.h>

```
int statvfs (const char *path, struct statvfs *buf);  
int statvfs64 (const char *path, struct statvfs64 *buf);
```

**Description** See `fstatvfs()`.

## \_\_STDC\_\_ - macro for ANSI conformance

**Syntax** \_\_STDC\_\_

**Description** This macro generates the value 1 for a compilation with `SOURCE-PROPERTIES=PARAMETERS(LANGUAGE-STANDARD=ANSI)` and is otherwise undefined.

**Notes** This macro need not be defined in a header file. Its name is recognized and replaced by the compiler.

## \_\_STDC\_VERSION\_\_ - Amendment 1 conformity?

**Syntax** \_\_STDC\_VERSION\_\_

**Description** Specifies which version of the ANSI standard is supported  
This macro expands to the decimal constant 199409L and shows that the implementation conforms to Amendment 1.

**Notes** The macro does not have to be defined in a header file. It's name is recognized by the compiler and replaced.



## stderr, stdin, stdout - variables for standard I/O streams

**Syntax**      `#include <stdio.h>`  
`extern FILE *stderr, *stdin, *stdout;`

**Description** A file with associated buffering is called a **stream** and is declared to be a pointer to a defined type `FILE`. The `fopen()` function creates certain descriptive data for a stream and returns a pointer to designate that stream in all further transactions.

There are three data streams which are predefined at program startup and need not be opened explicitly (see `stdio.h`):

`stdin`            Standard input, for reading conventional input.  
`stdout`           Standard output, for writing conventional output.  
`stderr`           Standard error, for the output of diagnostic and error messages.

When opened, the standard error stream is not fully buffered (see `setvbuf()`); the standard input and standard output streams are fully buffered if and only if the stream is not associated with an interactive device.

The following symbolic values in `unistd.h` define the file descriptors assigned to the data streams `stdin`, `stdout` and `stderr` when the application is started:

`STDIN_FILENO`  
File descriptor for standard input, `stdin`. Its value is 0.  
  
`STDOUT_FILENO`  
File descriptor for standard output, `stdout`. Its value is 1.  
  
`STDERR_FILENO`  
File descriptor for standard error, `stderr`. Its value is 2.

**See also** `fclose()`, `feof()`, `ferror()`, `fileno()`, `fopen()`, `fread()`, `fseek()`, `getc()`, `gets()`, `popen()`, `printf()`, `putc()`, `puts()`, `read()`, `scanf()`, `setbuf()`, `setvbuf()`, `tmpfile()`, `ungetc()`, `vprintf()`, `stdio.h`, `unistd.h`.

## step - compare regular expressions

Syntax      `#include <regex.h>`  
            `int step(const char *string, const char *exbuf);`

Description See `regex()`.

Notes        This function will not be supported by the X/Open standard in the future.

## strcasecmp, strncasecmp - non-case-sensitive string comparison

Syntax      `#include <strings.h>`  
            `int strcasecmp(const char *s1, const char *s2);`  
            `int strncasecmp(const char *s1, const char *s2, size_t n);`

Description The `strcasecmp()` function compares the string referenced by `s1` with the string referenced by `s2`. The strings to be compared must be terminated with the null byte. Uppercase and lowercase are not distinguished. `strncasecmp()` is used in the same way, except that no more than `n` bytes can be compared.

In the POSIX locale, `strcasecmp()` and `strncasecmp()` convert uppercase letters into lowercase letters before they do the comparison. The results are not specified in other locales.

Return value      Integer      On successful execution, `strcasecmp()` returns an integer which is greater than, equal to or less than zero, depending on whether the string identified by `s1` is greater than, equal to or less than the string referenced by `s2`. No distinction is made between uppercase and lowercase. `strncasecmp()` is used in the same way, except that no more than the first `n` characters of both strings can be compared.

See also      `strings.h`.

## strcat - concatenate two strings

**Syntax**      `#include <string.h>`

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

**Description** `strcat()` appends a copy of the string `s2` to the end of string `s1` and returns a pointer to `s1`. The terminating null byte (`\0`) at the end of string `s1` is overwritten by the first byte of string `s2`. `strcat()` terminates the concatenated string with a null byte (`\0`).

**Return value**  
Pointer to the result string `s1`.

**Notes**      Strings terminated with the null byte (`\0`) are expected as arguments.  
`strcat()` does not verify whether `s1` has enough space to accommodate the result!  
The behavior is undefined if memory areas overlap.

**See also**    `strncat()`, `string.h`.

## strchr - scan string for characters

**Syntax**      `#include <string.h>`

```
char *strchr(const char *s, int c);
```

**Description** `strchr()` searches for the first occurrence of character `c` in string `s` and returns a pointer to the located position in `s` if successful.  
The terminating null byte (`\0`) is considered to be part of the string.

**Return value**  
Pointer to the position of `c` in string `s` if successful.  
Null pointer if `c` is not contained in string `s`.

**Notes**      `strchr()` and `index()` are equivalent.

**See also**    `index()`, `rindex()`, `strrchr()`, `string.h`.

## strcmp - compare two strings

Syntax `#include <string.h>`

```
int strcmp(const char *s1, const char *s2);
```

Description `strcmp()` compares strings *s1* and *s2* lexically, e.g.:

"circle" is lexically less than "circular";

"bustle" is lexically greater than "bus".

Return value

Integer value:

< 0            *s1* is lexically less than *s2*.

= 0            *s1* and *s2* are lexically equal.

> 0            *s1* is lexically greater than *s2*.

Notes            Strings terminated with the null byte (`\0`) are expected as arguments. If this is not the case, the result is random.

The collating sequence is based on the EBCDIC character set.

See also        `strncmp()`, `string.h`.

## strcoll - compare strings using collating sequence

**Syntax**      `#include <string.h>`

```
int strcoll(const char *s1, const char *s2);
```

**Description** `strcoll()` returns an integer greater than, equal to, or less than 0, depending on whether string *s1* is greater than, equal to, or less than string *s2*, respectively. The strings are compared on the basis of the setting for the `LC_COLLATE` category of the current locale (see `setlocale()`).

`strcoll()` and `strxfrm()` can be used to sort strings based on the environment. `strcoll()` is intended for applications in which the number of comparisons per string is low. If strings are to be compared frequently, `strxfrm()` should be used together with `strcmp()` in a manner that allows the transformation process to be performed just once.

**Return value**

Integer value:

< 0            *s1* is lexically less than *s2*.  
= 0            *s1* and *s2* are lexically equal.  
> 0            *s1* is lexically greater than *s2*.

**Errors**

`strcoll()` will fail if:

`EINVAL`      The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

**Notes**

Strings terminated with the null byte (`\0`) are expected as arguments.

Since `strcoll()` has no return value to indicate an error, errors can only be detected as follows: by setting `errno` to 0, calling the function, and then checking `errno` after the function returns. If `errno` is not equal to 0, it can be assumed that an error occurred.

**See also**

`setlocale()`, `strcmp()`, `strxfrm()`, `string.h`.

## strcpy - copy string

**Syntax**     #include <string.h>

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

**Description** `strcpy()` copies the string `s2`, including the terminating null byte (`\0`), into the memory area pointed to by `s1`. The space pointed to by `s1` must be large enough to accommodate the string `s2` as well as the terminating null byte (`\0`).

**Return value**

Pointer to the result string `s1`.

**Notes**       A string terminated with the null byte (`\0`) is expected as the second argument. `strcpy()` does not verify whether `s1` is large enough to accommodate the result. The behavior is undefined if memory areas overlap.

**See also**    `strncpy()`, `string.h`.

## strcspn - get length of complementary substring

**Syntax**     #include <string.h>

```
size_t strcspn(const char *s1, const char *s2);
```

**Description** Starting at the beginning of string `s1`, `strcspn()` calculates the length of the segment that does not contain a single character from string `s2`. The terminating null byte (`\0`) is not treated as part of string `s2`.

The function is terminated and the segment length is returned on encountering a character in `s1` that matches a character in `s2`.

If the first character in `s1` already matches a character in `s2`, the segment length is equal to 0.

**Return value**

Integer value that indicates the segment length (number of non-matching characters) as of the beginning of string `s1`.

**Notes**       Strings terminated with the null byte (`\0`) are expected as arguments.

**See also**    `strspn()`, `string.h`.

## strdup - duplicate string

**Syntax**        `#include <string.h>`  
                 `char *strdup(const char *s1);`

**Description** `strdup()` returns a pointer to a new string, which is a duplicate of the string pointed to by `s1`. The space for the new string is allocated using `malloc()`. The returned pointer can be passed to the `free()` function. A null pointer is returned if the new string cannot be created.

**Return value**  
If successful, the function returns a pointer to a new string. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

**Errors**        `strdup()` will fail if:  
                 `ENOMEM`        There is not enough memory.

**See also**      `malloc()`, `free()`, `string.h`.

## strerror - get message string

Syntax `#include <string.h>`

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

Description `strerror()` maps the error number in *errnum* to a locale-dependent message string and returns a pointer to that string (see section "Error handling" on page 126). The returned string must not be modified by the program, but may be overwritten by a subsequent call to `strerror()` or `popen()`.

The contents of the message strings returned by `strerror()` should be determined by the setting of the `LC_MESSAGES` category in the current locale. A complete listing of error numbers and error messages as well as explanations can be found under the header `errno.h`.

Return value

Pointer to a message string  
if successful.

Null pointer if an error occurs. `errno` is set to indicate the error.

Errors

`strerror()` will fail if:

`EINVAL` The value of *errnum* is not a valid error number.

Notes

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `strerror()`, then check `errno`, and if it is not equal to 0, assume that an error has occurred.

See also `perror()`, `popen()`, `errno.h`, `string.h`, section "Error handling" on page 126.



## strfill - copy substring (BS2000)

**Syntax**      `#include <string.h>`  
`char *strfill(char *s1, const char *s2, size_t n);`

**Description** `strfill()` copies a maximum of  $n$  characters from string  $s2$  to the memory area pointed to by  $s1$ .

Copying takes place as described below, depending on the lengths and contents of strings  $s1$  and  $s2$  and the value specified for  $n$ :

1.  $n$  characters are always copied to  $s1$  (except in case 5), regardless of the length of string  $s1$ . In other words:
  - If  $s1$  contains more than  $n$  characters, the remaining characters on the right in  $s1$  are retained.
  - If  $s1$  contains less than  $n$  characters,  $s1$  is extended to the length of  $n$ . In this case,  $s1$  is not automatically terminated with a null byte (see Notes).
2.  $s2$  contains less than  $n$  characters:  
The required number of blanks are added to the copied characters from  $s2$  until a total of  $n$  characters have been written.
3.  $s2$  contains more than  $n$  characters:  
Only the leading  $n$  characters from  $s2$  are copied.
4.  $s2$  is empty:  
 $s1$  is filled with  $n$  blanks.
5.  $s2$  is passed as a null pointer:  
 $(n - \text{strlen}(s1))$  blanks are appended to string  $s1$ . If this subtraction yields a negative result or 0, i.e. if the number of characters in  $s1$  is greater than or equal to  $n$ , the contents of  $s1$  remain unchanged.

**Return value**  
Pointer to the result string  $s1$ .

**Notes**      Strings terminated with the null byte (`\0`) are expected as arguments. `strfill()` does not verify whether  $s1$  is large enough for the result and does not automatically terminate the result string with a null byte (`\0`). To avoid an unpredictable result, string  $s1$  should be explicitly terminated with the null byte after every call to `strfill()`.  
The behavior is undefined if memory areas overlap.

**See also**      `strncpy()`.

## strfmon - convert monetary value to string

Syntax `#include <monetary.h>`  
`ssize_t strfmon(char *s, size_t maxsize, const char *format, ...);`

Description `strfmon()` writes characters of type 'character' to the field pointed to by *s* in accordance with the *format* specification. No more than *maxsize* bytes are written to the field.

*format* is a string containing two types of object: simple characters that are copied into the output stream, and conversion specifications. Conversion specifications cause arguments (none, one or more) to be converted and formatted. If there are not enough arguments for the specified format, the result is undefined. If there are more arguments than allowed for by the format, the excess arguments are ignored.

A conversion specification consists of the following elements:

1. a % character
2. optional flags
3. an optional field size
4. an optional left-adjusted precision
5. an optional right-adjusted precision
6. a conversion character that determines how the arguments are converted (mandatory)

### Flags

To control the conversion, you can specify one or more of the flags listed below:

- `=f` An equals sign followed by a single *f*. This character is used as a filler for numeric values. The fill character must be representable in a single byte so that it does not clash with specifications on the field size and the alignment. The default fill character is the blank.  
This flag does not affect filling due to a field-size specification: the blank is always used as a filler in this case.  
The flag is ignored if no left-adjusted precision is specified.
- `^` Monetary values are formatted without grouping characters. By default, monetary values are formatted with the grouping characters that apply for the current locale.

- + or ( Controls how positive and negative monetary values are displayed. Only one of the two characters + and ( can be specified.  
If + is specified, the values defined in the current locale for + and - are used (in the USA, for example, the empty string for positive values and the - sign for negative values).  
If ( is specified, negative values are enclosed in brackets.  
The default value is +.
- ! Suppresses the currency symbol in the output.
- Controls the alignment. If this flag is set, values in the fields are left-aligned instead of right-aligned (i.e. padded to the right).

### Field size

- w* A sequence of decimal digits defining the minimum field size in bytes. The result of the conversion is right-aligned in the field and, if necessary, padded (the result is left-aligned if the - flag is set).  
The default field size is 0.

### Left-adjusted precision

- #n* A sequence of decimal digits prefixed by the # character. This value specifies the maximum number of digits expected to the left of the radix character (e.g. the period in \$ \*\*15.20).  
This option can be used to align the results of several `strfmon` calls in columns. It can also be used to fill up free positions with a special character, e.g. \$ \*\*\*123.45. This option causes a monetary value to be formatted as if it had *n* digits. If more than *n* digit positions are required, this conversion specification is ignored. Free digit positions are filled with the numeric filler character (see flag =f).  
If a grouping is defined in the current locale and is not suppressed (flag ^), the grouping characters are inserted before free positions are padded with filler characters. Filler characters are not grouped, even if they are numeric.  
To guarantee the alignment, all characters like currency symbols or minus signs before or after the number are positioned before or after the number in the formatted output using blanks so that their positive and negative formats have the same lengths.

### Right-adjusted precision

*.p* A sequence of decimal digits prefixed by the `.` character. This word specifies how many digits are to appear to the right of the radix character (e.g. the period in `$**15.20`). If *p* is 0, the radix character is also omitted. If right-adjusted precision is not specified, the right-adjusted precision defined in the current locale is used. The sum to be formatted is rounded to the specified number of digits before the formatting.

### Conversion characters

The following conversion characters are available:

- i* The argument of type `double` is formatted according to the international currency format defined in the locale (e.g. in the USA: USD 1,234.56).
- n* The argument of type `double` is formatted according to the national currency format defined in the locale (e.g. in the USA: \$1,234.56).
- %* Converted to a `%.`, no argument is converted. The complete conversion specification must be `%%`.

### Locale information

The behavior of the function is influenced by the `LC_MONETARY` category of the locale of the program. This applies particularly to the monetary radix character (which can be different from the numeric radix character which applies for the `LC_NUMERIC` category), the grouping character, the currency symbols and the currency formats. The international currency symbol should comply with the ISO 4217:1987 standard.

### Return value

- Number of bytes that was written to the field pointed to by *s* (without the terminating null byte) if the total number of bytes written, including the null byte, is not greater than *maxsize*.
- 1 otherwise. In the event of an error the contents of the field are undefined. `errno` is set to indicate the error.

### Errors

`strfmon()` will fail if:

- E2BIG The conversion was aborted due to lack of space in the buffer.

**Example** The following examples refer to a locale in the USA and the values 123.45, -123.45 and 3456.781:

| Conversion specification | Result                                      | Comment                                                                    |
|--------------------------|---------------------------------------------|----------------------------------------------------------------------------|
| %n                       | \$123.45<br>-\$123.45<br>\$3,456.78         | Default formatting                                                         |
| %11n                     | \$123.45<br>-\$123.45<br>\$3,456.78         | Right alignment within an 11-character field                               |
| %#5n                     | \$ 123.45<br>-\$ 123.45<br>\$ 3,456.78      | Values through 99.999 are aligned in a column                              |
| %= *#5n                  | \$***123.45<br>-\$***123.45<br>\$*3,456.78  | Specification of a filler character for free positions                     |
| %=0#5n                   | \$000123.45<br>-\$000123.45<br>\$03,456.78  | Filler characters are not grouped, even if the filler character is a digit |
| %^#5n                    | \$ 123.45<br>-\$ 123.45<br>\$ 3456.78       | Suppress grouping character                                                |
| %^#5.0n                  | \$ 123<br>-\$ 123<br>\$ 3456                | Round to integer                                                           |
| %^#5.4n                  | \$ 123.4500<br>-\$ 123.4500<br>\$ 3456.7800 | Increase right-adjusted precision                                          |
| %(#5n                    | \$ 123.45<br>( \$ 123.45)<br>\$ 3456.78     | Alternative representation for positive/negative values                    |
| %!(#5n                   | 123.45<br>( 123.45)<br>3456.78              | Suppress currency symbol                                                   |

**See also** localeconv(), monetary.h.

## strptime - convert date and time to string

Syntax `#include <time.h>`

```
size_t strptime(char *s, size_t maxsize, const char *format, const struct tm *timeptr);
```

Description `strptime()` formats the date and time as specified in the *format* string and places them in the array pointed to by *s*. The *format* string consists of zero or more conversion specifications and ordinary characters. All ordinary characters, including the terminating null byte, are copied unchanged into the array. If `strptime()` is used, no more than *maxsize* bytes are placed in the array.

If *format* is equal to `(char *)0`, the default format `"%c"` will be used for `strptime()`.

Each conversion specification is replaced by appropriate characters, as described in the following list. The appropriate characters are determined by the `LC_TIME` category of the locale and, in the case of `strptime()`, by the contents of *timeptr*:

|                 |                                                                    |
|-----------------|--------------------------------------------------------------------|
| <code>%%</code> | The character <code>%</code>                                       |
| <code>%a</code> | Abbreviated weekday name of the locale                             |
| <code>%A</code> | Full weekday name of the locale                                    |
| <code>%b</code> | Abbreviated month name of the locale                               |
| <code>%B</code> | Full month name of the locale                                      |
| <code>%c</code> | Appropriate date and time representation of the locale             |
| <code>%C</code> | Century (the year divided by 100, truncated to an integer) (00-99) |
| <code>%d</code> | Day of the month (01-31)                                           |
| <code>%D</code> | Date as <code>%m/%d/%y</code>                                      |
| <code>%e</code> | Day of the month (1-31; single digits are preceded by a space)     |
| <code>%f</code> | Date and time represented in accordance with <code>date()</code>   |
| <code>%h</code> | Abbreviated month name of the locale                               |
| <code>%H</code> | Hours (00-23), 24-hour representation                              |
| <code>%I</code> | Hours (01-12), 12-hour representation                              |
| <code>%j</code> | Day of the year (001-366)                                          |
| <code>%m</code> | Number of the month (01-12)                                        |
| <code>%M</code> | Minutes (00-59)                                                    |
| <code>%n</code> | Equivalent to <code>\n</code>                                      |
| <code>%p</code> | Locale's equivalent of either AM or PM                             |
| <code>%r</code> | Time in the form <code>%I:%M:%S [AMŠPM]</code>                     |

|    |                                                                                                                                                                                                                                                              |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| %R | Time in the form %H:%M                                                                                                                                                                                                                                       |
| %S | Seconds (00-61), allows leap seconds                                                                                                                                                                                                                         |
| %t | Inserts a tab character                                                                                                                                                                                                                                      |
| %T | Time in the form %H:%M:%S                                                                                                                                                                                                                                    |
| %u | Weekday as a number (1-7), Monday = 1                                                                                                                                                                                                                        |
| %U | Week number of the year (00-53). The first week begins with the first Sunday of the year. All days before the first Sunday of the year belong to week 0.                                                                                                     |
| %V | Week number of the year (01-53), with Monday as the first day of the week/ If the week containing January 1 has four or more days in the new year, then it is considered week 1. Otherwise, it is week 53 of the previous year, and the next week is week 1. |
| %w | Weekday as a number (0-6); Sunday = 0                                                                                                                                                                                                                        |
| %W | Week number of the year (01-53), with Monday as the first day of week 1. All days before the first Monday of the year belong to week 0.                                                                                                                      |
| %x | Appropriate date representation of the locale                                                                                                                                                                                                                |
| %X | Appropriate time representation of the locale                                                                                                                                                                                                                |
| %y | Year within the century (00-99)                                                                                                                                                                                                                              |
| %Y | Year in the form ccy (e.g. 1986)                                                                                                                                                                                                                             |
| %Z | Timezone name or abbreviation, or no bytes if no timezone exists.                                                                                                                                                                                            |

The difference between %U and %W is based on which day is considered the first weekday. Week 01 is the first week in January that begins with a Sunday (for %U) or a Monday (for %W). Week number 00 includes the days before the first Sunday (%U) or Monday (%W) in January.

### Modified conversion specifiers

Some conversion specifiers can be modified by the characters E or O to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier (see also the manual "SINIX 5.41 Guidelines for Programmers: Internationalization - Localization). If the alternative format or specification does not exist for the current locale, the behavior will be as if the unmodified conversion specification were used.

|     |                                                                            |
|-----|----------------------------------------------------------------------------|
| %Ec | The locale's alternative date and time representation.                     |
| %EC | 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.                              |

|                  |                                                                                                                                                                                 |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>%Ey</code> | Offset from <code>%EC</code> (year only) in the locale's alternative representation.                                                                                            |
| <code>%EY</code> | Alternative representation for the year.                                                                                                                                        |
| <code>%Od</code> | Day of the month, using the locale's alternative numeric symbols, padded as needed with leading zeros if an alternative symbol for zero exists; otherwise, with leading spaces. |
| <code>%Oe</code> | Day of month, using the locale's alternative numeric symbols, padded as needed with leading spaces.                                                                             |
| <code>%OH</code> | The hour (24-hour clock), using the locale's alternative numeric symbols.                                                                                                       |
| <code>%OI</code> | The hour (12-hour clock), using the locale's alternative numeric symbols.                                                                                                       |
| <code>%Om</code> | The month, using the locale's alternative numeric symbols.                                                                                                                      |
| <code>%OM</code> | The minutes, using the locale's alternative numeric symbols.                                                                                                                    |
| <code>%OS</code> | The seconds, using the locale's alternative numeric symbols.                                                                                                                    |
| <code>%Ou</code> | The weekday as a number in the locale's alternative representation (Monday = 1).                                                                                                |
| <code>%OU</code> | The week number of the year (Sunday is the first day of the week; rules correspond to <code>%U</code> ) using the locale's alternative numeric symbols.                         |
| <code>%OV</code> | The week number of the year (Sunday is the first day of the week, rules correspond to <code>%V</code> ) using the locale's alternative numeric symbols.                         |
| <code>%Ow</code> | Number of the weekday (Sunday = 0), using the locale's alternative numeric symbols.                                                                                             |
| <code>%OW</code> | The week number of the year (Monday is the first day of the week), using the locale's alternative numeric symbols.                                                              |
| <code>%Oy</code> | The year (offset from <code>%C</code> ) in the locale's alternative representation, and using the locale's alternative symbols.                                                 |

The default language for the output of `strptime()` is U.S. English. The user can select the output language for `strptime()` by using `setlocale()` to set the `LC_TIME` category for the locale.

The timezone is taken from the environment variable `TZ` (see `ctime()`).

#### Return value

- Number of bytes copied to `s` (without the terminating null byte)
  - if the number of resulting bytes, including the null byte, does not exceed *maxsize*.
- 0
  - if an error occurs. The contents of `s` are indeterminate.

**See also** `clock()`, `ctime()`, `getenv()`, `setlocale()`, `time.h`.



## strlen - get length of string

**Syntax**      `#include <string.h>`

`size_t strlen(const char *s);`

**Description** `strlen()` determines the length of string *s*, not including the terminating null byte (`\0`).

*BS2000*

Whereas the `sizeof` operator always returns the defined length, `strlen()` calculates the actual number of bytes in a string. The newline (`\n`) character, if present, is also included. □

**Return value**

Length of the string *s*

if successful. The terminating null byte is not included in the count.

**Notes**      A string terminated with the null byte (`\0`) is expected as the argument.

## strlower - convert a string to lowercase letters *(BS2000)*

**Syntax**      `#include <string.h>`

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

**Description** `strlower()` copies string *s2*, including the null byte (`\0`), to the memory area pointed to by *s1*, converting uppercase letters to lowercase letters in the process.

If string *s2* is passed as a null pointer, the copy operation is not performed, and the uppercase letters in *s1* are converted to lowercase.

*s1* is the result string into which *s2* is to be copied or in which uppercase letters are to be converted to lowercase.

If *s2* is not passed as a null pointer, *s1* must be large enough to accommodate *s2*, including the null byte (`\0`).

**Return value**

Pointer to the result string *s1*.

**Notes**      Strings terminated with the null byte (`\0`) are expected as arguments.

`strlower()` does not verify whether *s1* is large enough to accommodate the result.

The behavior is undefined if memory areas overlap.

**See also**    `strupper()`, `tolower()`, `toupper()`.

## strncasecmp - non-case-sensitive string comparisons

**Syntax**     #include <strings.h>  
              int strncasecmp(const char \*s1, const char \*s2, size\_t n);

**Description** See `strcasecmp()`.

## strncat - concatenate two substrings

**Syntax**     #include <string.h>  
              char \*strncat(char \*s1, const char \*s2, size\_t n);

**Description** `strncat()` appends a maximum of *n* characters from string *s2* to the end of string *s1* and returns a pointer to *s1*.

The null byte (`\0`) at the end of string *s1* is overwritten by the first character of string *s2*.

If string *s2* contains less than *n* characters, only the characters from *s2* are appended to *s1*. If string *s2* contains more than *n* characters, only the first *n* characters from *s2* are appended to *s1*.

`strncat()` terminates the string with a null byte (`\0`).

**Return value**  
Pointer to the result string *s1*.

**Notes**       Strings terminated with the null byte (`\0`) are expected as arguments.

`strncat()` does not verify whether *s1* has enough space to accommodate the result! The behavior is undefined if memory areas overlap.

**See also**    `strcat()`, `string.h`.

## strncmp - compare two substrings

**Syntax**      `#include <string.h>`

```
int strncmp(const char *s1, const char *s2, size_t n);
```

**Description** `strncmp()` compares strings *s1* and *s2* lexically up to a maximum length of *n*, e.g.

```
strncmp("Sie", "Siemens", 3)
```

returns 0 (equal), because the first three characters of both arguments match one another.

**Return value**

Integer value:

< 0            In the first *n* characters, *s1* is lexically less than *s2*.

= 0            In the first *n* characters, *s1* and *s2* are lexically equal.

> 0            In the first *n* characters, *s1* is lexically greater than *s2*.

**Notes**        Strings terminated with the null byte (`\0`) are expected as arguments.

The collating sequence is based on the EBCDIC character set.

**See also**     `strcmp()`, `string.h`.

## strncpy - copy substring

**Syntax**      `#include <string.h>`

```
char *strncpy(char *s1, const char *s2, size_t n);
```

**Description** `strncpy()` copies a maximum of  $n$  characters from string  $s2$  to string  $s1$ .

If string  $s2$  contains less than  $n$  characters, only the length of  $s2$  (`strlen + 1`) is copied, and  $s1$  is then padded to the length of  $n$  with null bytes.

If string  $s2$  contains  $n$  or more characters (excluding the null byte), string  $s1$  is not automatically terminated with the null byte.

If string  $s1$  contains more than  $n$  characters and the last character copied from  $s2$  is not the null byte, any data which may still remain in  $s1$  is retained.

`strncpy()` does not automatically terminate  $s1$  with the null byte.

**Return value**

Pointer to the result string  $s1$ .

**Notes**      `strncpy()` does not verify whether  $s1$  has enough space to accommodate the result!

Since `strncpy()` does not automatically terminate the result string with the null byte, it may often be necessary to explicitly terminate  $s1$  with a null byte. This is typically the case when only a part of  $s2$  is being copied, and  $s2$  does not contain a null byte either.

The behavior is undefined if memory areas overlap.

**See also**    `strcpy()`, `strlen()`, `string.h`.

## strpbrk - get first occurrence of character in string

**Syntax**      `#include <string.h>`

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

**Description** `strpbrk()` searches string *s1* for the first character matching any character in string *s2* and returns a pointer to the located position in *s1* if successful. The terminating null byte (`\0`) is not considered part of string *s2*.

**Return value**

Pointer to the first matching character found in *s1*  
if successful.

Null pointer      if not a single match is present.

**Notes**      Strings terminated with the null byte (`\0`) are expected as arguments.

**See also**      `strchr()`, `strrchr()`, `string.h`.

## strptime - convert string to date and time

Syntax `#include <time.h>`

```
char *strptime(const char *buf, const char *format, struct tm *tm);
```

Description In accordance with *format*, `strptime()` converts the string pointed to by *buf* into individual values that are stored in the structure pointed to by *tm*.

The *format* string consists of none, one or more conversion statements. Each conversion statement consists of one of the following elements:  
one or more white-space characters (as defined in `isspace()`),  
a regular character (neither % nor white-space characters)  
or a conversion specification.

Each conversion specification consists of a % character followed by a conversion character that specifies the desired conversion. Between two conversion specifications there must be a white-space character or a non-alphanumeric character.

The following conversion characters are supported:

|    |                                                                                                                  |
|----|------------------------------------------------------------------------------------------------------------------|
| %% | Replaced by %                                                                                                    |
| %a | Weekday, whereby the name from the locale is used. Either the full name or the abbreviated name can be specified |
| %A | Same meaning as %a                                                                                               |
| %b | Month, whereby the name from the locale is used. Either the full name or the abbreviated name can be specified   |
| %B | Same meaning as %b                                                                                               |
| %c | Date and time representation according to the definition in the locale                                           |
| %C | Century (the year divided by 100, truncated to an integer) (00-99)                                               |
| %d | Day of the month (01-31)                                                                                         |
| %D | Date as %m/%d/%y                                                                                                 |
| %e | Same meaning as %d                                                                                               |
| %h | Same meaning as %b                                                                                               |
| %H | Hours (00-23), 24-hour representation                                                                            |
| %I | Hours (01-12), 12-hour representation                                                                            |
| %j | Day of the year (001-366)                                                                                        |
| %m | Number of the month (01-12)                                                                                      |
| %M | Minutes (00-59)                                                                                                  |
| %n | Replaced by a white-space character                                                                              |

|    |                                                                                                                                                          |
|----|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| %p | Locale's equivalent of AM or PM                                                                                                                          |
| %r | Time in the form %I:%M:%S%p                                                                                                                              |
| %R | Time in the form %H:%M                                                                                                                                   |
| %S | Seconds (00-61), allows leap seconds                                                                                                                     |
| %t | Replaced by a white-space character                                                                                                                      |
| %T | Time in the form %H:%M:%S                                                                                                                                |
| %U | Week number of the year (00-53). The first week begins with the first Sunday of the year. All days before the first Sunday of the year belong to week 0. |
| %w | Weekday as a number (0-6), Sunday = 0                                                                                                                    |
| %W | Week number of the year (00-53), with Monday as the first day of week 1. All days before the first Monday of the year belong to week 1.                  |
| %x | Date representation of the locale                                                                                                                        |
| %X | Time representation of the locale                                                                                                                        |
| %y | Year within the century (00-99)                                                                                                                          |
| %Y | Year in the form <i>ccyy</i> (e.g. 1986)                                                                                                                 |

### Modified conversion specifications

Some conversion specifications can be modified by the characters E or O to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specification (see also the manual "SINIX 5.41 Guidelines for Programmers: Internationalization - Localization"). If the alternative format or specification does not exist for the current locale, the behavior will be as if the unmodified conversion specification were used.

|     |                                                                                                                                                                                 |
|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| %Ec | The locale's alternative date and time representation.                                                                                                                          |
| %EC | 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 | Offset from %EC (year only) in the locale's alternative representation.                                                                                                         |
| %EY | Alternative representation for the year.                                                                                                                                        |
| %Od | Day of the month, using the locale's alternative numeric symbols, padded as needed with leading zeros if an alternative symbol for zero exists; otherwise, with leading spaces. |
| %Oe | Same meaning as %Od                                                                                                                                                             |
| %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 week number of the year (Sunday is the first day of the week; rules correspond to %U) using the locale's alternative numeric symbols. |
| %OV | The week number of the year (Sunday is the first day of the week, rules correspond to %V) using the locale's alternative numeric symbols. |
| %Ow | Number of the weekday (Sunday = 0), using the locale's alternative numeric symbols.                                                       |
| %OW | The week number of the year (Monday is the first day of the week), using the locale's alternative numeric symbols.                        |
| %Oy | The year (offset from %C) in the locale's alternative representation, and using the locale's alternative symbols.                         |

A conversion specification consisting of white-space characters is executed by the input being read up to the first character that is not a white-space character (this character remains unread), or until there are no more characters left.

A conversion specification comprising a regular character is executed by the next character from the buffer being read. If the character read from the buffer does not match the character of the conversion specification, the latter fails and the deviating character plus all characters that follow it remain unread.

A sequence of conversion specifications consisting of %n, %t, white-space characters and combinations of all these is executed by being read up to the first character that is not a white-space character (this character remains unread), or until there are no more characters left.

All other conversion specifications are executed by characters being read in until a character which matches the next conversion specification is read (it remains in the buffer) or until there are no more characters left. The read characters are then compared with the values in the locale that correspond to the conversion specification. If the matching value is found in the locale, the corresponding structure elements of the `tm` structure are set to the values that correspond to this information.

The search is not case-sensitive if it is a comparison of elements such as weekdays and month names.

If no matching value is found in the locale, `strptime()` fails and no more characters are read.



**Return value**

Pointer to the character after the last character read  
if successful.

Null pointer otherwise.

**Notes**

The special handling of white-space characters and many “same formats” is designed to simplify the use of identical format strings with `strftime()` and `strptime()`.

**See also**

`scanf()`, `strftime()`, `time()`, `time.h`.

## strrchr - get last occurrence of character in string

**Syntax**

```
#include <string.h>
```

```
char *strrchr(const char *s, int c);
```

**Description**

`strrchr()` searches for the last occurrence of character `c` in string `s` and returns a pointer to the located position in `s` if successful.

The terminating null byte (`\0`) is also considered as a character.

**Return value**

Pointer to the position of `c` in string `s`  
if successful.

Null pointer if `c` is not contained in string `s`.

**Notes**

The functions `strrchr()` and `rindex()` are equivalent.

**See also**

`index()`, `rindex()`, `strchr()`, `string.h`.

## strspn - get length of substring

**Syntax**      `#include <string.h>`

`size_t strspn(const char *s1, const char *s2);`

**Description** Starting at the beginning of string *s1*, `strspn()` computes the length of the segment that contains only characters from string *s2*.  
The function is terminated, and the segment length is returned on encountering the first character in *s1* that does not match any character in *s2*.  
If the first character in *s1* matches none of the characters in *s2*, the segment length is equal to 0.

**Return value**

Integer value

that indicates the segment length (number of matching characters), starting at the beginning of string *s1*.

**Notes**      Strings terminated with the null byte (`\0`) are expected as arguments.

**See also**    `strcspn()`, `string.h`.

## strstr - find substring in string

**Syntax**      `#include <string.h>`

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

**Description** `strstr()` searches for the first occurrence of string *s2* (without the terminating null byte) in string *s1*.

**Return value**

Pointer      to the start of the string found in *s1*.

Null pointer    if *s2* is not contained in *s1*.

Pointer      to the start of *s1*, if *s2* has a length of 0.

**Notes**      Strings terminated with the null byte (`\0`) are expected as arguments.

**See also**    `strchr()`, `string.h`.

## strtod - convert string to double-precision number

**Syntax**      `#include <stdlib.h>`

```
double strtod(const char *s, char **endptr);
```

**Description** `strtod()` converts the string to which *s* points into a floating-point number of type `double`. The string to be converted may be structured as follows:

$$\left[ \left\{ \begin{array}{c} \textit{tab} \\ \lfloor \end{array} \right\} \dots \right] \left[ \left\{ \begin{array}{c} + \\ - \end{array} \right\} \right] [\textit{digit} \dots] [.] [\textit{digit} \dots] \left[ \left\{ \begin{array}{c} E \\ e \end{array} \right\} \right] \left[ \left\{ \begin{array}{c} + \\ - \end{array} \right\} \right] \textit{digit} \dots$$

Any white-space character may be used for *tab* (see definition under `isspace()`).

`strtod()` also recognizes strings that start with a digit but end with some other character. In such cases, `strtod()` first truncates the numeric part and converts it to a floating-point value.

`strtod()` returns a pointer (*\*endptr*) to the first non-convertible character in string *s* via the second argument *endptr* of type `char **`, but only if *endptr* is not passed as a null pointer.

If *endptr* is a null pointer, `strtod()` is executed like the `atof()` function:

`atof(s)` is equivalent to `strtod(s, (char **)NULL)` and `strtod(s, NULL)`.

If *endptr* is not a null pointer, a pointer (*\*endptr*) to the first character in *s* that completes the conversion is returned.

If absolutely no conversion is possible, *\*endptr* will be set to the start address of string *s*.

### Return value

|                                                   |                                                                                                                                        |
|---------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Floating-point number of type <code>double</code> | for strings which are structured as described above and represent a numeric value within the permissible floating-point range.         |
| 0                                                 | for strings that do not conform to the syntax described above or do not begin with convertible characters.                             |
| HUGE_VAL                                          | for strings whose numeric value lies outside the permissible floating-point range.<br><code>errno</code> is set to indicate the error. |

**Errors**      `strtod()` will fail if:

|        |                                                  |
|--------|--------------------------------------------------|
| ERANGE | The return value causes an overflow or underflow |
| EINVAL | No conversion could be performed.                |

**Notes** The radix character in the string to be converted is determined by `LC_NUMERIC` category of the locale. The default is a period.

**See also** `atof()`, `atoi()`, `atol()`, `isspace()`, `strtol()`, `strtoul()`, `stdlib.h`.

## strtok - split string into tokens

**Syntax** `#include <string.h>`  
`char * strtok(char *s1, const char *s2);`

**Description** `strtok()` can be used to split a complete string `s1` into substrings called "tokens", e.g. a sentence into individual words, or a source program statement into its smallest syntactical units. The pointer to `s1` may only be passed in the first call to `strtok()`; subsequent calls must be specified with a null pointer.

The start and end criterion for each token are separator characters (delimiters), which must be specified in a second string `s2`. Tokens may be delimited by one or more such separators or by the beginning and end of the entire string `s1`. Blanks, colons, commas, etc., are typical separators between the words of a sentence.

`strtok()` processes exactly one token per call. The first call returns a pointer to the beginning of the first token found. Each subsequent call returns a pointer to the beginning of the next token. `strtok()` terminates each token with the null byte (`\0`).

A different delimiter string `s2` may be specified in each call.

`strtok()` is not thread-safe. Use the reentrant function `strtok_r()` when needed.

**Return value** Pointer to the start of a token.

A pointer to the first token is returned at the first call; a pointer to the next token at the next call, and so on. `strtok()` terminates each token in `s1` with a null byte (`\0`) by overwriting the first found delimiter in each case with `\0`.

Null pointer, if no token, or no further token was found.

**See also** `string.h`, `strtok_r()`.

## strtok\_r - split string into tokens (thread-safe)

**Syntax**      `#include <string.h. >`  
`char *strtok_r(char *s, const char *sep, char **lasts);`

**Description** The function `strtok_r()` is the thread-safe version of `strtok()`.

The function `strtok_r()` can be used to split a complete string *s* terminated by a null into 0 or more substrings called "tokens". Tokens may be delimited by one or more separators that are specified in the *sep* string. The *lasts* argument points to a pointer provided by the user that `strtok_r()` uses to obtain the information necessary to continue processing this string.

The first time `strtok_r()` is called, *s* points to a string terminated with a null byte, and *sep* points to a string terminated with a null byte with delimiters. The value pointed to by *lasts* is ignored. The function `strtok_r()` returns a pointer to the beginning of the first token found, overwrites the first delimiter found with the NULL character (`\0`) and updates the pointer pointed to by *lasts*.

To get additional tokens, a null pointer is specified for *s* and the value from the last call is specified for *lasts* in the subsequent call. This can be continued until there are no more tokens. In this case a null pointer is returned.

A different delimiter string *sep* may be specified in each call.

The function `strtok_r()` returns a pointer to the token. If no token was found, a null pointer is returned.

### Return value

Pointer to the token found  
if successful.

Null pointer      if no token is found.

**See also**      `strtok()`, `string()`.

## strtol - convert string to long integer

Syntax `#include <stdlib.h>`

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

Description `strtol()` converts the string to which *s* points into an integer of type `long int`. The string to be converted may be structured as follows:

$$\left[ \left\{ \begin{array}{c} \text{tab} \\ \lfloor \quad \end{array} \right\} \dots \right] \left[ \left\{ \begin{array}{c} + \\ - \end{array} \right\} \right] \left[ \left\{ \begin{array}{c} 0 \\ 0X \end{array} \right\} \right] \text{digit} \dots$$

Any white-space character may be used for *tab* (see definition under `isspace()`).

Depending on the base (see *base*), the digits 0 to 9 and the letters a (or A) to z (or Z) may be used for *digit*.

`strtol()` also recognizes strings that begin with convertible digits (including octal and hexadecimal digits) but end with some other characters. In such cases, `strtol()` truncates the numeric part before converting it.

`strtol()` returns a pointer to the first non-convertible character in string *s* via the second argument *endptr* of type `char **`, but only if *endptr* is not passed as a null pointer.

If no conversion is possible at all, *\*endptr* is set to the start address of string *s*.

A third argument, *base*, defines the base (e.g. decimal, octal or hexadecimal) for the conversion.

*base* may be any integer from 0 to 36. For base 11 to base 36, the letters a (or A) to z (or Z) in the string to be converted are assumed to be digits, with corresponding values from 10 (a/A) to 35 (z/Z).

If *base* is equal to 0, the base will be determined from the structure of string *s* as shown below:

|                  |         |
|------------------|---------|
| leading 0        | base 8  |
| leading 0X or 0x | base 16 |
| otherwise        | base 10 |

If the parameter *base* = 16 is used for calculations, the characters 0X or 0x, which may optionally follow the sign in string *s*, if present, will be ignored.

**Return value**

Integer value of type `long int` for strings that have a structure as described above and which represent a numeric value.

0 for strings that do not conform to the syntax described above.

`LONG_MAX` or `LONG_MIN` if the result overflows, depending on the sign.

**Errors**

`strtol()` will fail if:

`ERANGE` The return value causes an overflow.

`EINVAL` The value of *base* is not supported.

**Notes**

If *endptr* is a null pointer and *base* is equal to 10, `strtol()` is executed like the `atol()` function:

`atol(s)` is equivalent to `strtol(s, NULL, 10)`.

**See also**

`atol()`, `atoi()`, `isalpha()`, `strtod()`, `strtoul()`, `stdlib.h`.

## strtol - convert string to long long integer

Syntax `#include <stdlib.h>`

`long long int strtoll(const char restrict *s, char ** restrict endptr, int base);`

Description `strtoll()` converts the EBCDIC string to which *s* points into an integer of type `long long int`. The string to be converted may be structured as follows:

$$\left[ \left\{ \begin{array}{c} \textit{tab} \\ \lfloor \quad \end{array} \right\} \dots \right] \left[ \left\{ \begin{array}{c} + \\ - \end{array} \right\} \right] \left[ \left\{ \begin{array}{c} 0 \\ 0X \end{array} \right\} \right] \textit{digit} \dots$$

Any white-space character may be used for *tab* (see definition under `isspace()`).

Depending on the base (see *base*), the digits 0 to 9 and the letters a (or A) to z (or Z) may be used for *digit*.

`strtoll()` also recognizes strings that begin with convertible digits (including octal and hexadecimal digits) but end with some other characters. In such cases, `strtoll()` truncates the numeric part before converting it.

`strtoll()` also returns a pointer to the first non-convertible character in string *s* via the second argument *endptr* of type `char **`, but only if *endptr* is not passed as a null pointer.

If no conversion is possible at all, *\*endptr* is set to the start address of string *s*.

A third argument, *base*, defines the base (e.g. decimal, octal or hexadecimal) for the conversion.

The function has the following parameters:

`const char *s`

Pointer to the EBCDIC string to be converted.

`char **endptr`

If *endptr* is not a null pointer, a pointer (*\*endptr*) to the first character in *s* is returned that terminates the conversion.

If no conversion is possible, *\*endptr* is set to the start address of the string *s*.

`int base`

Integer from 0 to 36 that is to be used as the base for the calculation.

For base 11 to base 36, the letters a (or A) to z (or Z) in the string to be converted are assumed to be digits, with corresponding values from 10 (a/A) to 35 (z/Z).



If *base* is equal to 0, the base will be determined from the structure of string *s* as shown below:

|                  |         |
|------------------|---------|
| leading 0        | base 8  |
| leading 0X or 0x | base 16 |
| otherwise        | base 10 |

If the parameter *base* = 16 is used for calculations, the characters 0X or 0x, which may optionally follow the sign in string *s*, if present, will be ignored.

### Return value

Integer value of type `long long int` for strings that have a structure as described above and which represent a numeric value.

0 for strings that do not conform to the syntax described above. The conversion is not executed. If the value of *base* is not supported, `errno` is set to `EINVAL`.

`LLONG_MAX` or `LLONG_MIN` if the result overflows, depending on the sign

`ULLONG_MAX` if the result overflows. `errno` is set to `ERANGE`.

**Notes** If *endptr* is a null pointer and *base* is equal to 10, `strtol()` differs from the `atoll()` function only in the error handling:

`atoll(s)` is equivalent to `strtol(s, (char **)NULL, 10)`.

**See also** `atol()`, `atoll()`, `atoi()`, `strtol()`, `stroul()`, `stroull()`, `wcstol()`, `wcstoll()`, `wcstoul()`, `wcstoull()`

## strtol - convert string to unsigned long integer

Syntax `#include <stdlib.h>`

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

Description `strtol()` converts the string to which *s* points into an integer of type `unsigned long int`. The string to be converted may be structured as follows:

$$\left[ \left. \begin{array}{c} \textit{tab} \\ \square \end{array} \right\} \dots \right] \left[ \begin{array}{c} 0 \\ 0X \end{array} \right] \textit{digit} \dots$$

Any white-space character may be used for *tab* (see definition under `isspace()`).

Depending on the base (see *base*), the digits 0 to 9 and the letters a (or A) to z (or Z) may be used for *digit*.

`strtol()` also recognizes strings that begin with convertible digits (including octal and hexadecimal digits) but end with some other characters. In such cases, `strtol()` truncates the numeric part before converting it.

`strtol()` returns a pointer to the first non-convertible character in string *s* via the second argument *endptr* of type `char **`, but only if *endptr* is not passed as a null pointer.

If no conversion is possible at all, *\*endptr* is set to the start address of string *s*.

A third argument, *base*, defines the base (e.g. decimal, octal or hexadecimal) for the conversion. *base* may be any integer from 0 to 36.

For base 11 to base 36, the letters a (or A) to z (or Z) in the string to be converted are assumed to be digits, with corresponding values from 10 (a/A) to 35 (z/Z).

If *base* is equal to 0, the base will be determined from the structure of string *s* as shown below:

|                  |         |
|------------------|---------|
| leading 0        | base 8  |
| leading 0X or 0x | base 16 |
| otherwise        | base 10 |

If the parameter *base* = 16 is used for calculations, the characters 0X or 0x, which may optionally follow the sign in string *s*, if present, will be ignored.

**Return value**

Integer value of type `unsigned long` for strings that have a structure as described above and which represent a numeric value.

0 for strings that do not conform to the syntax described above.

Largest possible value of type `unsigned long` if an overflow occurs; `errno` is set to indicate the error.

**Errors**

`strtoul()` will fail if:

`EINVAL` The value of *base* is not supported.

`ERANGE` The return value causes an overflow.

`EINVAL` The conversion could not be performed.

**See also** `atol()`, `atoi()`, `isalpha()`, `strtol()`, `stdlib.h`.

## strtoull - convert string to unsigned long long

Syntax `#include <stdlib.h>`

`unsigned long long int strtoull(const char restrict *s, char **restrict endptr, int base);`

Description `strtoull()` converts the string to which *s* points into an integer of type `unsigned long long int`. The string to be converted may be structured as follows:

$$\left[ \left\{ \begin{array}{c} \text{tab} \\ \_ \end{array} \right\} \dots \right] \left[ \left\{ \begin{array}{c} 0 \\ \text{OX} \end{array} \right\} \right] \text{digit} \dots$$

Any white-space character may be used for *tab* (see definition under `isspace()`).

Depending on the base (see *base*), the digits 0 to 9 and the letters a (or A) to z (or Z) may be used for *digit*.

`strtoull()` also recognizes strings that begin with convertible digits (including octal and hexadecimal digits) but end with some other characters. In such cases, `strtoull()` truncates the numeric part before converting it.

`strtoull()` also returns a pointer to the first non-convertible character in string *s* via the second argument *endptr* of type `char **`, but only if *endptr* is not passed as a null pointer.

If no conversion is possible at all, *\*endptr* is set to the start address of string *s*.

A third argument, *base*, defines the base (e.g. decimal, octal or hexadecimal) for the conversion.

The function has the following parameters:

`const char *s`

Pointer to the EBCDIC string to be converted.

`char **endptr`

If *endptr* is not a null pointer, a pointer (*\*endptr*) to the first character in *s* is returned that terminates the conversion.

If no conversion is possible, *\*endptr* is set to the start address of the string *s*.

`int base`

Integer from 0 to 36 that is to be used as the base for the calculation.

For base 11 to base 36, the letters a (or A) to z (or Z) in the string to be converted are assumed to be digits, with corresponding values from 10 (a/A) to 35 (z/Z).

If *base* is equal to 0, the base will be determined from the structure of string *s* as shown below:

|                  |         |
|------------------|---------|
| leading 0        | base 8  |
| leading 0X or 0x | base 16 |
| otherwise        | base 10 |

If the parameter *base* = 16 is used for calculations, the characters 0X or 0x, which may optionally follow the sign in string *s*, if present, will be ignored.

### Return value

Integer value of type `unsigned long long int` for strings that have a structure as described above and which represent a numeric value.

0 for strings that do not conform to the syntax described above. No conversion is performed. If the value of *base* is not supported, `errno` is set to `EINVAL`.

`LLONG_MAX` or `LLONG_MIN` depending on the sign

`ULLONG_MAX` if the result overflows. `errno` is set to `ERANGE`.

**See also** `atol()`, `atoll()`, `atoi()`, `strtol()`, `strtoll()`, `stroul()`, `wcstol()`, `wcstoll()`, `wcstoul()`, `wcstoull()`

## strupper - convert string to uppercase letters *(BS2000)*

**Syntax**      `#include <string.h>`

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

**Description** `strupper()` copies string `s2`, including the null byte (`\0`), to the memory area pointed to by `s1`, converting lowercase letters to uppercase in the process.

If string `s2` is passed as a null pointer, the copy operation is not performed, and the lowercase letters in `s1` are converted to uppercase.

If `s2` is not passed as a null pointer, `s1` must be large enough to accommodate `s2`, including the null byte (`\0`).

**Return value**

Pointer to the result string `s1`.

**Notes**      Strings terminated with the null byte (`\0`) are expected as arguments.

`strupper()` does not verify whether `s1` is large enough to accommodate the result.

The behavior is undefined if memory areas overlap.

**See also**    `strlower()`, `tolower()`, `toupper()`.

## strxfrm - string transformation based on LC\_COLLATE

**Syntax**      `#include <string.h>`

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

**Description** `strxfrm()` transforms the string `s2` and places the resulting string into the array `s1`. The transformation is such that if `strcmp()` were applied to two transformed strings, it would return a value corresponding to the result of `strcoll()` applied to the same two original strings. The transformation is based on the collating sequence defined by the local setting of the program's `LC_COLLATE` category (see `setlocale()`).

A maximum of `n` bytes are placed into the resulting array pointed to by `s1`, including the terminating null byte. If `n` is 0, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

**Return value**

**Length**            of the transformed string (excluding the terminating null byte) if successful.

**Value  $\geq n$**       The contents of the array `s1` are indeterminate.

Since no return value is reserved to indicate an error, errors can only be detected as follows: by setting `errno` to 0, calling the `strxfrm()` function, and then checking `errno` after the function returns. If `errno` is non-zero, it may be assumed that an error occurred.

**Errors**            `strxfrm()` will fail if:

**EINVAL**            The `s2` argument contains characters outside the domain of the collating sequence.

**Notes**            A string terminated with the null byte (`\0`) is expected as argument `s2`.

String `s2` is not modified by `strxfrm()`. The transformation is performed in a work area.

If the return value is greater than or equal to `n`, the contents of string `s1` will be indeterminate, since no null byte was written.

If the hexadecimal value 0 has been assigned to one of the characters in string `s2` in the current locale, the transformed string will be terminated using that character as the null byte.

**See also**        `setlocale()`, `strcoll()`, `strcmp()`, `string.h`.

## swab - swap bytes

**Syntax**      `#include <unistd.h>`  
`void swab(const void *src, void *dest, ssize_t nbytes);`

**Description** `swab()` copies *nbytes* bytes, which are pointed to by *src*, to the object pointed to by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even and not negative. If *nbytes* is odd and positive, `swab()` copies and exchanges *nbytes-1* bytes, and the disposition of the last byte is unspecified. If *nbytes* is negative, `swab()` does nothing. If arguments overlap, the behavior of `swab` is undefined.

**See also**      `unistd.h`.

## swapcontext - swap user context

**Syntax**      `#include <ucontext.h>`  
`int swapcontext (ucontext_t *oucp, const ucontext_t *ucp);`

**Description** See `makecontext()`.



## swprintf - output formatted wide characters

Syntax `#include <wchar.h>`  
`int swprintf(wchar_t *s, size_t n, const wchar_t *format [, arglist]);`

Description See `fwprintf()`.

## swscanf - formatted read

Syntax `#include <wchar.h>`  
`int swscanf(const wchar_t *s, const wchar_t *format [, arglist]);`

Description See `makecontextfwscanf()`.

## symlink - make symbolic link to file

Syntax `#include <unistd.h>`

```
int symlink(const char *path1,, const char *path2);
```

Description `symlink()` creates a symbolic link. Its name is the pathname referenced by *path2*. This pathname must not be the same as the name of an existing file or a symbolic link. The content of the symbolic link is the string referenced by *path1*. A symbolic link can refer to another file system. The file identified by *path1* need not be present.

The file to which the symbolic link points is used when an `open()` operation is performed on the link. A `stat()` on a symbolic link returns the referenced file, whereas an `lstat()` returns information about the link itself. This can lead to surprising results when a symbolic link is made to a directory. To avoid undesirable side effects in programs, the `readlink()` call can be used to read the contents of a symbolic link.

Return value

0 if successful.  
-1 if an error occurs; `errno` is set to indicate the error.

Errors

`symlink()` will fail if:

**EACCES** Search permission is denied for the directory in which the symbolic link was created. Search permission is denied for a component of the path prefix of *path2*.

**EEXIST** The file or symbolic link specified using *path2* already exists.

**ENOTDIR** A component of the path prefix of *path2* is not a directory.

**EIO** An I/O error occurred while reading from or writing to the file system.

**ELOOP** Too many symbolic links were encountered in resolving *path2*.

**ENAMETOOLONG** The length of the *path1* or *path2* argument exceeds `{PATH_MAX}` or a component of *path1* or *path2* is longer than `{NAME_MAX}`.  
`symlink()` could also fail if the resolving of a symbolic link produces a result whose length exceeds `{PATH_MAX}`.

**ENOENT** A component of the pathname prefix of *path2* does not exist or *path2* is an empty string.

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ENOSPC           | <p>The directory in which the entry for the new symbolic link is to be created cannot be extended because there is no space left on the file system containing the directory.</p> <p>The new symbolic link cannot be created because there is no space left on the file system which will contain the link.</p> <p>There are no free inodes on the file system on which the file is to be created.</p>                                                                                  |
| EROFS            | <p>The new symbolic link would reside on a read-only file system.</p>                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <i>Extension</i> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| EDQUOT           | <p>The directory in which the entry for the new symbolic link is to be placed cannot be extended because the maximum number of disk blocks allocated to the user (i.e. the user's quota) on the file system was exceeded.</p> <p>The new symbolic link cannot be created because the user's quota of disk blocks on the file system that is to contain the link was exceeded.</p> <p>The user's quota of inodes on the file system on which the file is to be created was exceeded.</p> |
| EFAULT           | <p><i>path1</i> or <i>path2</i> points outside the allocated address space for the process.</p>                                                                                                                                                                                                                                                                                                                                                                                         |
| ENOSYS           | <p>The file system does not support symbolic links. □</p>                                                                                                                                                                                                                                                                                                                                                                                                                               |

**Notes**     `symlink()` is executed for POSIX files only

**See also**   `lchown()`, `link()`, `lstat()`, `open()`, `readlink()`, `unistd.h`, and the command `cp` in the manual "POSIX Commands".

## sync - update superblock

**Syntax**      `#include <unistd.h>`  
`void sync(void);`

**Description** `sync()` causes all information in memory that updates file systems to be scheduled for writing out to all file systems. This includes modified superblocks, modified inodes and delayed block-special I/O files.

`sync()` should be used by programs which check a file system, for example `fsck()` or `df()`. `sync` is mandatory before a new system is loaded.

When `sync()` returns, the writing is not necessarily finished. The system call `fsync()` finishes writing before it returns.

**Return value**  
The function does not return any values.

**See also**      `fsync()`, `unistd.h`.

## sysconf - get numeric value of configurable system variable

Syntax `#include <unistd.h>`

```
long int sysconf(int name);
```

Description `sysconf()` can be used to determine the current value of a configurable system variable specified by *name*. These values represent the configurable limits of the operating system.

The table below lists the system variables from the headers `limits.h`, `unistd.h` or `time.h` for which values can be queried using `sysconf()`. The symbolic constants containing the corresponding values used for *name* are defined in `unistd.h`:

| System variable       | Value of <i>name</i> (constant) |
|-----------------------|---------------------------------|
| ARG_MAX               | _SC_ARG_MAX                     |
| AIO_LISTIO_MAX        | _SC_AIO_LISTIO_MAX              |
| AIO_MAX               | _SC_AIO_MAX                     |
| AIO_PRIO_DELTA_MAX    | _SC_AIO_PRIO_DELTA_MAX          |
| BC_BASE_MAX           | _SC_BC_BASE_MAX                 |
| BC_DIM_MAX            | _SC_BC_DIM_MAX                  |
| BC_SCALE_MAX          | _SC_BC_SCALE_MAX                |
| BC_STRING_MAX         | _SC_BC_STRING_MAX               |
| CHILD_MAX             | _SC_CHILD_MAX                   |
| CLK_TCK               | _SC_CLK_TCK                     |
| COLL_WEIGHTS_MAX      | _SC_COLL_WEIGHTS_MAX            |
| DELAYTIMER_MAX        | _SC_DELAYTIMER_MAX              |
| EXPR_NEST_MAX         | _SC_EXPR_NEST_MAX               |
| LINE_MAX              | _SC_LINE_MAX                    |
| LOGIN_NAME_MAX        | _SC_LOGIN_NAME_MAX              |
| NGROUPS_MAX           | _SC_NGROUPS_MAX                 |
| MQ_OPEN_MAX           | _SC_MQ_OPEN_MAX                 |
| MQ_PRIO_MAX           | _SC_MQ_PRIO_MAX                 |
| OPEN_MAX              | _SC_OPEN_MAX                    |
| POSIX_ASYNCHRONOUS_IO | _SC_ASYNCHRONOUS_IO             |
| _POSIX_FSYNC          | _SC_FSYNC                       |
| _POSIX_MAPPED_FILES   | _SC_MAPPED_FILES                |
| _POSIX_MEMLOCK        | _SC_MEMLOCK                     |
| _POSIX_MEMLOCK_RANGE  | _SC_MEMLOCK_RANGE               |

| <b>System variable</b>            | <b>Value of <i>name</i> (constant)</b> |
|-----------------------------------|----------------------------------------|
| _POSIX_MEMORY_PROTECTION          | _SC_MEMORY_PROTECTION                  |
| _POSIX_MESSAGE_PASSING            | _SC_MESSAGE_PASSING                    |
| _POSIX_PRIORITIZED_IO             | _SC_PRIORITIZED_IO                     |
| _POSIX_PRIORITY_SCHEDULING        | _SC_PRIORITY_SCHEDULING                |
| _POSIX_REALTIME_SIGNALS           | _SC_REALTIME_SIGNALS                   |
| _POSIX_SEMAPHORES                 | _SC_SEMAPHORES                         |
| _POSIX_SHARED_MEMORY_OBJECTS      | _SC_SHARED_MEMORY_OBJECTS              |
| _POSIX_SYNCHRONIZED_IO            | _SC_SYNCHRONIZED_IO                    |
| _POSIX_THREADS                    | _SC_THREADS                            |
| _POSIX_THREAD_ATTR_STACKADDR      | _SC_THREAD_ATTR_STACKADDR              |
| _POSIX_THREAD_ATTR_STACKSIZE      | _SC_THREAD_ATTR_STACKSIZE              |
| _POSIX_THREAD_PRIORITY_SCHEDULING | _SC_THREAD_PRIORITY_SCHEDULING         |
| _POSIX_THREAD_PRIO_INHERIT        | _SC_THREAD_PRIO_INHERIT                |
| _POSIX_THREAD_PRIO_PROTECT        | _SC_THREAD_PRIO_PROTECT                |
| _POSIX_THREAD_PROCESS_SHARED      | _SC_THREAD_PROCESS_SHARED              |
| _POSIX_THREAD_SAFE_FUNCTIONS      | _SC_THREAD_SAFE_FUNCTIONS              |
| _POSIX_TIMERS                     | _SC_TIMERS                             |
| _POSIX2_C_BIND                    | _SC_2_C_BIND                           |
| _POSIX2_C_DEV                     | _SC_2_C_DEV                            |
| _POSIX2_C_VERSION                 | _SC_2_C_VERSION                        |
| _POSIX2_CHAR_TERM                 | _SC_2_CHAR_TERM                        |
| _POSIX2_FORT_DEV                  | _SC_2_FORT_DEV                         |
| _POSIX2_FORT_RUN                  | _SC_2_FORT_RUN                         |
| _POSIX2_LOCALEDEF                 | _SC_2_LOCALEDEF                        |
| _POSIX2_SW_DEV                    | _SC_2_SW_DEV                           |
| _POSIX2_UPE                       | _SC_2_UPE                              |
| _POSIX2_VERSION                   | _SC_2_VERSION                          |
| _POSIX_JOB_CONTROL                | _SC_JOB_CONTROL                        |
| _POSIX_SAVED_IDS                  | _SC_SAVED_IDS                          |
| _POSIX_VERSION                    | _SC_VERSION                            |
| PTHREAD_DESTRUCTOR_ITERATIONS     | _SC_THREAD_DESTRUCTOR_ITERATIONS       |
| PTHREAD_KEYS_MAX                  | _SC_THREAD_KEYS_MAX                    |
| PTHREAD_STACK_MIN                 | _SC_THREAD_STACK_MIN                   |

| System variable                                                                   | Value of <i>name</i> (constant)  |
|-----------------------------------------------------------------------------------|----------------------------------|
| PTHREAD_THREADS_MAX                                                               | _SC_THREAD_THREADS_MAX           |
| PTHREAD_DESTRUCTOR_ITERATIONS                                                     | _SC_THREAD_DESTRUCTOR_ITERATIONS |
| PTHREAD_KEYS_MAX                                                                  | _SC_THREAD_KEYS_MAX              |
| PTHREAD_STACK_MIN                                                                 | _SC_THREAD_STACK_MIN             |
| PTHREAD_THREADS_MAX                                                               | _SC_THREAD_THREADS_MAX           |
| RE_DUP_MAX                                                                        | _SC_RE_DUP_MAX                   |
| RTSIG_MAX                                                                         | _SC_RTSIG_MAX                    |
| SEM_NSEMS_MAX                                                                     | _SC_SEM_NSEMS_MAX                |
| SEM_VALUE_MAX                                                                     | _SC_SEM_VALUE_MAX                |
| STREAM_MAX                                                                        | _SC_STREAM_MAX                   |
| SIGQUEUE_MAX                                                                      | _SC_SIGQUEUE_MAX                 |
| TIMER_MAX                                                                         | _SC_TIMER_MAX                    |
| TTY_NAME_MAX                                                                      | _SC_TTY_NAME_MAX                 |
| TZNAME_MAX                                                                        | _SC_TZNAME_MAX                   |
| Maximal size of the data buffer of the functions<br>getgrgid_r() and getgrnam_r() | _SC_GETGR_R_SIZE_MAX             |
| Maximal size of the data buffer of the functions<br>getpwnam_r() and getpwuid_r() | _SC_GETPW_R_SIZE_MAX             |

## Return value

Current numeric value of *name*

if successful.

The returned value will not be lower than the corresponding value in the application if it were compiled with the implementation's `limits.h` or `unistd.h`. The value will not change during the lifetime of the calling process.

-1

if *name* is an invalid value.

`errno` is set to indicate the error.

if *name* does not have a defined value.

In this case, the value of `errno` is not changed.

## Errors

`sysconf()` will fail if:

`EINVAL`

The value of the *name* argument is invalid.

**Notes** Since all return values are permitted in a successful situation, applications wishing to check for error situations should set `errno` to 0, then call `sysconf()`, and if it returns -1, check to see if `errno` is non-zero.

If the value of `sysconf(_SC_2_VERSION)` is not equal to the value of the symbolic constant `_POSIX2_VERSION`, the commands available via `system()` or `popen()` might not behave in conformance with XPG4. The interfaces described in this manual will, however, continue to operate in conformance with XPG4 even if `sysconf(_SC_2_VERSION)` reports that the commands no longer perform as defined in the standard.

**See also** `pathconf()`, `limits.h`, `time.h`, `unistd.h`.



**sysfs - get information on file system type** (*extension*)

**Syntax**        `#include <sys/fstyp.h>`  
                  `#include <sys/fsid.h>`

```
int sysfs(int opcode[], const char *fsname][ ], int fs_index, char *buf]);
```

**Description** `sysfs()` returns information on the file system types configured in the system. The number of arguments accepted by `sysfs()` depends on the value of *opcode*.

The following values for *opcode* are accepted in the C runtime system:

|           |                                                                                                                                                                                                                                             |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| GETFSIND  | Translates <i>fsname</i> , a null-terminated file-system identifier, into a file-system type index.                                                                                                                                         |
| GETFSTYP  | Translates <i>fs_index</i> , a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by <i>buf</i> . This buffer must be at least of size FSTYPSZ (see <code>sys/fstyp.h</code> ). |
| GETNFSYTP | Returns the total number of file system types configured in the system.                                                                                                                                                                     |

**Return value**

|                                        |                                                                   |
|----------------------------------------|-------------------------------------------------------------------|
| File system type index                 | if <i>opcode</i> is GETFSIND; upon successful completion.         |
| 0                                      | if <i>opcode</i> is GETFSTYP; upon successful completion.         |
| Number of file system types configured | if <i>opcode</i> is GETNFSYTP; upon successful completion.        |
| -1                                     | if unsuccessful. <code>errno</code> is set to indicate the error. |

**Errors** `sysfs()` will fail if:

|        |                                                                                                                                                                             |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EINVAL | <i>fsname</i> points to an invalid file-system identifier; <i>fs_index</i> is zero or invalid, or <i>opcode</i> is invalid, or an attempt was made to access a BS2000 file. |
| EFAULT | <i>buf</i> or <i>fsname</i> points outside the allocated address space for the process.                                                                                     |

**Notes**        `sysfs()` is executed only for POSIX files

**See also**     `sys/fstyp.h`, `sys/fsid.h`.

## syslog - log message

Syntax      #include <syslog.h>  
              void syslog(int *priority*, const char \**message*, .../ \* *argument* \*/);

Description See `closelog()`.

## system - execute system command

Syntax `#include <stdlib.h>`

```
int system(const char *command);
```

Description `system()` passes the system command given in the string *command* to a command interpreter for execution. Depending on which functionality is selected, *command* is interpreted as a POSIX or BS2000 command (see section "Scope of the supported C library" on page 18).

If *command* is a POSIX command, the environment of the executed command will be as if a child process were created using `fork()`, and the child process invoked the `sh` command using `execl()` as follows:

```
execl(shell_path, "sh", "-c", command, (char *)0);
```

where *shell\_path* must be replaced by the pathname of the `sh` command.

`system()` will not return until the child process has terminated and will not affect its termination status.

### BS2000

If *command* is a BS2000 command, it will be executed in the same task in which the program that invokes `system()` is running. Note that if programs or procedures are started in the `system` call, the calling program will be unloaded (see "Notes"). □

### Return value

Exit status of the command interpreter

if *command* is not a null pointer and the command was successfully executed. The exit status of the command interpreter is returned in the format specified by `waitpid()`. It corresponds to the exit status of the `sh` command, except that if some error prevents the command interpreter from executing after the child process is created, the return value from `system()` will be as if the command interpreter had terminated using `exit(127)` or `_exit(127)`.

≠ 0 if *command* is a null pointer and a command interpreter exists.

-1 if a child process cannot be created, or if the command interpreter has no exit status. `errno` is set to indicate the error.

*BS2000*

- 0 if *command* was executed successfully (return value of the BS2000 command: 0)
- 1 if the BS2000 command was not executed successfully (return value of the command: error code  $\neq$  0)
- undefined if control is not returned to the program following the BS2000 command (see "Notes"). □

## Errors

`system()` will fail if:

- EAGAIN The system does not have the resources required to create a further process or the system-specific limit for the maximum number of simultaneously executing processes for the system or an individual user ID {CHILD\_MAX} would be exceeded.

*Extension*

- EINTR `system()` was interrupted by a signal.
- ENOMEM Not enough memory is available.

## Notes

If the return value of `system()` is not -1, its value can be decoded by using the macros that are defined in both `sys/wait.h` as well as `stdlib.h`.

The following function can be used to determine whether or not an XPG4-conformant environment is present: `sysconf(_SC_2_VERSION)`.

Note that, while `system()` must ignore `SIGINT` and `SIGQUIT` and block `SIGCHLD` while waiting for the child to terminate, the handling of signals in the executed command is as specified by `fork()` and `exec`. For example, if `SIGINT` is being caught or is set to `SIG_DFL` when `system()` is called, then the child will be started with `SIGINT` handling set to `SIG_DFL`.

Ignoring `SIGINT` and `SIGQUIT` in the parent process prevents coordination problems (two processes reading from the same terminal, for example) when the executed command ignores or catches one of the signals. It is also usually the correct action when the user has given a command to the application to be executed synchronously (as in the "!" command in many interactive applications). In either case, the signal should be delivered only to the child process, not to the application itself. There is one situation where ignoring the signals might have less than the desired effect. This is when the application uses `system()` to perform some task invisible to the user. If the user typed the interrupt character (^C, for example) while `system()` is being used in this way, one would expect the application to be killed, but only the executed command will be killed. Applications that use `system()` in this way should carefully check the return status from `system()` to see if the executed command was successful, and should take appropriate action when the command fails.

Blocking `SIGCHLD` while waiting for the child to terminate prevents the application from catching the signal and obtaining status from `system()`'s child process before `system()` can get the status itself.

The context in which the command is ultimately executed may differ from that in which `system()` was called. For example, when file descriptors that have the `FD_CLOEXEC` flag set are closed, the process ID and parent process ID of `system()` and the command will be different. Furthermore, if the executed command changes its environment variables or its current working directory, that change will not be reflected in the caller's context.

`sh` may not be available following a call to `chroot`.

There is no defined way for an application to find the specific path for the shell. However, `confstr()` can provide a value for `PATH` that is guaranteed to find the `sh` command.

#### *BS2000*

The BS2000 command must not exceed a maximum length of 2048 characters and need not be specified with the system slash (/).

In the case of some BS2000 commands (e.g. `START-PROG`, `LOAD-PROG`, `CALL-PROCEDURE`, `DO`, `HELP-SDF`), control is not returned to the calling program after they are called. Programs that permit premature terminations should therefore flush all buffers (`fflush()`) and/or close files before the `system` call.

`system()` passes the *command* string as input to the BS2000 command processor MCLP without any changes (see also the manual "BS2000/OSD-BC Executive Macros"). No conversion to uppercase is performed.

**See also** `bs2system()`, `exec`, `fork()`, `pipe()`, `sysconf()`, `wait()`, `limits.h`, `signal.h`, `stdio.h`, and the command `sh` in the manual "POSIX Commands (BS2000/OSD)".

## tan - compute tangent

**Syntax**      `#include <math.h>`  
`double tan(double x);`

**Description** `tan()` computes the trigonometric function tangent of a floating-point number *x* (within the permissible range of floating-point numbers).

*x* is the floating-point number, specified in radians.

**Return value**

`tan(x)`            Tangent of *x* if successful.  
`+/-HUGE_VAL`      If an overflow occurs.  
                     `errno` is set to indicate the error.

**Errors**

`tan()` will fail if:  
`ERANGE`            The value of *x* causes an overflow.

**See also**      `atan()`, `cos()`, `sin()`, `tanh()`, `math.h`.

## tanh - compute hyperbolic tangent

**Syntax**      `#include <math.h>`  
`double tanh(double x);`

**Description** `tanh()` computes the hyperbolic tangent of a floating-point number *x* (within the permissible range of floating-point numbers).

**Return value**

`tanh(x)`            Hyperbolic tangent of *x* if successful.

**See also**      `atan()`, `cos()`, `cosh()`, `sin()`, `sinh()`, `tan()`, `math.h`.

## tcdrain - wait for transmission of output

**Syntax**      `#include <termios.h>`  
`int tcdrain (int fildes);`

**Description** `tcdrain()` waits until all output written to the object specified by *fildes* is transmitted. The *fildes* argument is an open file descriptor associated with a terminal.

Any attempts to use `tcdrain()` from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no signal is sent.

**Return value**

0                    if successful.  
-1                    If an error occurs. `errno` is set to indicate the error.

**Errors**          `tcdrain()` will fail if:

`EBADF`              *fildes* is not a valid file descriptor.  
`EINTR`              A signal was caught during the `tcdrain()` system call.

*Extension*

`EINVAL`              An attempt was made to access a BS2000 file. □  
`EIO`                  The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.  
`ENOTTY`              The file associated with *fildes* is not a terminal.

**Notes**            `tcdrain()` has no effect on block-mode terminals.

**See also**        `tcflush()`, `termios.h`, `unistd.h`, section "General terminal interface" on page 94.

## tcflow - suspend or restart data transmission

Syntax `#include <termios.h>`

```
int tcflow(int fildev, int action);
```

Description `tcflow()` suspends transmission or reception of data on the object referred to by *fildev*, depending on the value of *action*. The *fildev* argument is a file descriptor associated with a terminal.

If *action* is `TCOOFF`, output is suspended. If *action* is `TCOON`, suspended output is restarted. If *action* is `TCIOFF`, input is stopped by transmitting a `STOP` character, and if *action* is `TCION`, input is restarted by transmitting a `START` character.

The default on the opening of a terminal file is that neither its input nor its output are suspended.

Attempts to use `tcflow()` from a process which is a member of a background process group on a *fildev* associated with its controlling terminal, will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no `SIGTTOU` signal is sent.

### Extension

All values are supported for connections with a remote processor. □

Return value

0 if successful.  
-1 if an error occurs. `errno` is set to indicate the error.

Errors `tcflow()` will fail if:

`EBADF` *fildev* is not a valid file descriptor.

`EINVAL` *action* is not a supported value.

### Extension

`EINVAL` An attempt was made to access a `BS2000` file. □

`EIO` The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.

`ENOTTY` The file associated with *fildev* is not a terminal.

Notes `tcflow()` has no effect on block-mode terminals.

See also `tcsendbreak()`, `termios.h`, `unistd.h`, section "General terminal interface" on page 94.



## tcf flush - discard non-transmitted data

Syntax `#include <termios.h>`

```
int tcf flush(int fil des, int queue_selector);
```

Description *fil des* is a file descriptor associated with a terminal. Upon successful completion, `tcf flush()` discards data that was written to the object referred to by *fil des* but not transmitted, or data that was received but not read, depending on the value of *queue\_selector*:

If *queue\_selector* is `TCIFLUSH`, data that was received but not read is flushed; if *queue\_selector* is `TCOFLUSH`, data that was written but not transmitted is flushed, and if *queue\_selector* is `TCIOFLUSH`, both the data that was received but not read and the data that was written but not transmitted are flushed.

Attempts to use `tcf flush()` from a process which is a member of a background process group on a *fil des* associated with its controlling terminal, will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no `SIGTTOU` signal is sent.

### Extension

All values are supported for connections with a remote processor. □

Return value

0 if successful.  
-1 if an error occurs. `errno` is set to indicate the error.

Errors

`tcf flush()` will fail if:

`EBADF` *fil des* is not a valid file descriptor.  
`EINVAL` *queue\_selector* is not a supported value.

### Extension

`EINVAL` An attempt was made to access a BS2000 file. □  
`EIO` The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.  
`ENOTTY` The file associated with *fil des* is not a terminal.

Notes `tcf flush()` has no effect on block-mode terminals.

See also `tcdrain()`, `termios.h`, `unistd.h`, section "General terminal interface" on page 94.

## tcgetattr - get parameters associated with terminal

**Syntax**      `#include <termios.h>`

```
int tcgetattr(int fildes, struct termios *termios_p);
```

**Description** `tcgetattr()` reads the parameters of the terminal associated with *fildes* and writes them into the `termios` structure pointed to by *termios\_p*.

*fildes* is a file descriptor associated with a terminal.

*termios\_p* is a pointer to a `termios` structure.

`tcgetattr()` may be executed from any process.

`tcgetattr()` can be called from a background process, and the terminal attributes can then be modified from a foreground process.

### *Extension*

The output baud rate always corresponds to the input baud rate and is equal to 38400 (see `tcsetattr()` for details). □

If the terminal device does not support split baud rates, the input baud rate stored in the `termios` structure will be 0.

**Return value**

0                    if successful.

-1                   if an error occurs. `errno` is set to indicate the error.

**Errors**      `tcgetattr()` will fail if:

EBADF              *fildes* is not a valid file descriptor.

### *Extension*

EINVAL              An attempt was made to access a BS2000 file. □

ENOTTY              The file associated with *fildes* is not a terminal.

**See also**      `tcsetattr()`, `termios.h`, section "General terminal interface" on page 94.

## tcgetpgrp - get foreground process group ID

**Syntax**      `#include <unistd.h>`

*Optional*

`#include <sys/types.h> □`

`pid_t tcgetpgrp(int fildev);`

**Description** `tcgetpgrp()` returns the value of the foreground process group ID associated with the terminal.

If there is no foreground process group, `tcgetpgrp()` returns a value greater than 1 that does not match the process group ID of any existing process group.

`tcgetpgrp()` is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

**Return value**

Value of the foreground process group ID associated with the terminal  
if successful.

-1              if an error occurs. `errno` is set to indicate the error.

**Errors**      `tcgetpgrp()` will fail if:

EBADF          *fildev* is not a valid file descriptor.

*Extension*

EINVAL          An attempt was made to access a BS2000 file. □

ENOTTY          The calling process does not have a controlling terminal, or the file is not the controlling terminal.

**See also**      `setsid()`, `setpgid()`, `tcsetpgrp()`, `sys/types.h`, `unistd.h`.

## tcgetsid - get session ID of specified terminal

**Syntax**      `#include <termios.h>`

```
pid_t tcgetsid(int fildes);
```

**Description** `tcgetsid()` returns the process group ID of the session that is controlled by the terminal specified in *fildes*.

**Return value**

Process group ID of the session associated with the specified terminal if successful.

(pid\_t)-1      otherwise. `errno` is set to indicate the error.

**Errors**      `tcgetsid()` will fail if:

EACCES      No controlling terminal is assigned to the *fildes* argument.

EBADF      The *fildes* argument is not a valid file descriptor.

ENOTTY      The file *fildes* is not a terminal.

**See also**      `termios.h`.

## tcsendbreak - interrupt serial data transmission

Syntax      #include <termios.h>  
              int tcsendbreak(int *fildev*, int *duration*);

### Description

#### *Extension*

In non-conformance with XPG4, this function has no effect and returns without performing any action. □

### Return value

0                if successful.  
-1               if an error occurs. `errno` is set to indicate the error.

### Errors

`tcsendbreak()` will fail if:

EBADF           *fildev* is not a valid file descriptor.

#### *Extension*

EINVAL          An attempt was made to access a BS2000 file. □

EIO             The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.

ENOTTY          The file associated with *fildev* is not a terminal.

See also      *termios.h*, *unistd.h*, section "General terminal interface" on page 94.

## tcsetattr - set parameters associated with terminal

Syntax `#include <termios.h>`

```
int tcsetattr(int fildes, int optional_actions, const struct termios *termios_p);
```

Description The `tcsetattr()` function sets the parameters associated with the terminal referred to by the file descriptor *fildes* and stores them in the `termios` structure pointed to by *termios\_p* as follows:

If *optional\_actions* is `TCSANOW`, the change will occur immediately.

If *optional\_actions* is `TCSADRAIN`, the change will occur after all output written to *fildes* is transmitted. This function should be used when changing parameters that affect output.

If *optional\_actions* is `TCSAFLUSH`, the change will occur after all output written to *fildes* is transmitted, and all input so far received but not read will be discarded before the change is made.

If the output baud rate stored in the `termios` structure pointed to by *termios\_p* is 0, a call to `tcsetattr()` will disconnect the line.

If this value is non-zero, all related values in the `termios` structure will have no effect. If the other values in the `termios` structure are also without effect, -1 is returned, and `errno` is set to `EINVAL`.

If the input baud rate stored in the `termios` structure pointed to by *termios\_p* is 0, the input baud rate set in the hardware will be the same as the output baud rate stored in the `termios` structure.

The `tcsetattr()` function will return successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It will set all the attributes that implementation supports as requested and leave all the attributes not supported by the implementation unchanged. If none of the requested actions can be performed, it will return -1 and set `errno` to `EINVAL`. If the input and output baud rates differ and are a combination that is not supported by the hardware, neither baud rate is changed. A subsequent call to `tcgetattr()` will return the actual state of the terminal device (reflecting both the changes made and the values that could not be changed in the previous `tcsetattr()` call). The `tcsetattr()` function will not change the values in the `termios` structure, regardless of whether or not it actually accepts them.

No action other than a call to `tcsetattr()` or a close of the last file descriptor in the system associated with the terminal can cause any of the terminal attributes defined in this manual to change.

Attempts to use `tcsetattr()` from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no `SIGTTOU` signal is sent.

## Return value

- 0 if successful.
- 1 if an error occurs. `errno` is set to indicate the error.

Errors `tcsetattr()` will fail if:

- EBADF *fdes* is not a valid file descriptor.
- EINVAL *optional\_actions* is not a supported value.

*Extension*

- EINVAL An attempt was made to access a BS2000 file. □
- EIO The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.
- ENOTTY The file associated with *fdes* is not a terminal.

**Notes** When trying to change baud rates, applications should first call `tcsetattr()` and then call `tcgetattr()` in order to determine what baud rates were actually selected.

**See also** `cfgetispeed()`, `tcgetattr()`, `termios.h`, `unistd.h`, section "General terminal interface" on page 94.

## tcsetpgrp - set foreground process group ID

Syntax `#include <unistd.h>`

*Optional*

`#include <sys/types.h>`

`int tcsetpgrp(int fildev, pid_t pgid_id);`

**Description** If the process has a controlling terminal, `tcsetpgrp()` will set the foreground process group ID associated with the terminal to the value `pgid_id`. The file of the terminal specified by `fildev` must be the controlling terminal of the calling process, and the controlling terminal must be currently associated with the session of the calling process. The value of `pgid_id` must match a process group ID of a process in the same session as the calling process.

**Return value**

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

**Errors** `tcsetpgrp()` will fail if:

EBADF `fildev` is not a valid file descriptor.

EINVAL `pgid_id` is not a valid process group ID.

*Extension*

EINVAL An attempt was made to access a BS2000 file.

ENOTTY The calling process does not have a controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM The value of `pgid_id` does not match the process group ID of a process in the same session as the calling process.

**See also** `tcgetpgrp()`, `sys/types.h`, `unistd.h`.

## tdelete - delete node from binary search tree

Syntax `#include <search.h>`

`void *tdelete (const void *key, void **rootp, int (*compar) (const void *, const void *));`

**Description** See `tsearch()`.



**tell - get current value of file position indicator** (*BS2000*)

Syntax `#include <stdio.h>`

```
long tell(int fil-des);
```

Description `tell()` returns the current value of the file position indicator for the file associated with file descriptor *fil-des*. `tell()` may be used for binary files (PAM, INCORE) as well as text files (SAM, ISAM). SAM files are always processed as text files with elementary functions.

*fil-des* is the file descriptor of the file for which the current value of the file-position indicator is to be determined.

## Return value

For binary files, current value of the file position indicator  
i.e. the number of bytes that offsets the file position indicator from the beginning of the file, if successful.

For text files, absolute position  
of the file position indicator if successful.

-1 if an error occurs; `errno` is set to indicate the error (e.g. `tell()` not permitted; number of blocks or records too large).

## Notes

The calls `tell(fil-des)` and `lseek(fil-des, 0L, SEEK_CUR)` are equivalent. `tell()` cannot be applied on system files (SYSDTA, SYSLST, SYSOUT).

Since information on the file position is stored in a field that is 4 bytes long, the following restrictions apply to the size of SAM and ISAM files when processing them with `tell()/lseek()`:

**SAM file**

|                         |              |
|-------------------------|--------------|
| Record length           | ≤ 2048 bytes |
| Number of records/block | ≤ 256        |
| Number of blocks        | ≤ 2048       |

**ISAM file**

|                   |             |
|-------------------|-------------|
| Record length     | ≤ 32 Kbytes |
| Number of records | ≤ 32 K      |

See also `lseek()`, `fseek()`, `ftell()`, `stdio.h`.

## telldir - get current location of named directory stream

**Syntax**      `#include <dirent.h>`  
`long int telldir(DIR *dirp);`

**Description** `telldir()` returns the current location associated with the specified directory stream. If `seekdir()` was the last operation on the directory stream, then `telldir()` returns the position specified in the *loc* argument of the `seekdir()` call.

**Return value**

Current location  
if successful

*Extension*

-1 if an error occurs. `errno` is set to indicate the error. ❏

**Errors**      `telldir()` will fail if:

*Extension*

EBADF      The file descriptor associated with the directory is no longer valid. This error will occur if the directory was closed. ❏

**Notes**      `telldir()` is executed only for POSIX files

**See also**    `readdir()`, `seekdir()`, `dirent.h`.

## tempnam - create pathname for temporary file

Syntax `#include <stdio.h>`

```
char *tempnam(const char *dir, const char *pfx);
```

Description `tempnam()` generates a pathname that may be used for a temporary file.

`tempnam()` allows the user to control the choice of a directory.

*dir* points to the name of the directory in which the file is to be created. If the environment variable `TMPDIR` is set, the directory specified there is used; otherwise, the one named under *dir*. If *dir* is a null pointer and the directory `{P_tmpdir}` does not name an accessible directory, the file names are generated with the directory name `/tmp`. If this is not accessible either, 0 is returned.

`P_tmpdir` is defined in `stdio.h` as `"/var/tmp"` as the directory in which temporary files are created.

Many applications prefer their temporary files to have specific initial letter sequences in their names. The *pfx* argument should be used for this. This argument may be a null pointer or point to a string of up to five bytes to be used as the first bytes in the name of the temporary file.

The name component generated by `tempnam()` is made up of two parts: the first comprises three uppercase letters (AAA, BAA, ..., ZAA, ZBA, ..., ZZZ); the second consists of a letter and the last five characters of the process ID. If the process ID consists of less than five characters, it is padded to five characters with leading zeros. For example, a complete name produced would be: `/var/tmp/AAAa00123`.

`tempnam()` uses `malloc()` to obtain space for the generated file name and returns a pointer to that area. Thus, any pointer value returned by `tempnam()` can serve as an argument to `free()` (see `malloc()`). If `tempnam()` cannot return the expected result for some reason, e.g. because `malloc()` failed or no appropriate directory could be found, a null pointer is returned.

`tempnam()` will fail if not enough memory is available.

Return value

Pointer to a string containing the generated pathname if successful.

Null pointer if an error occurs; `errno` is set to indicate the error.

0 if `/tmp` is not accessible, or if the `PROGRAM-ENVIRONMENT` variable is not set to `SHELL`.

**Errors**      `tempnam()` will fail if:

`ENOMEM`      There is not enough memory available for the new pathname.

*Extension*

`EINVAL`      An attempt was made to access a BS2000 file. □

**Notes**      `tempnam()` is executed only for POSIX files.

`tempnam()` generates a different pathname at each call.

Files created using `tempnam()` and either `fopen()` or `creat()` are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to remove a file when it is no longer needed. If this function is called more than `{TMP_MAX}` (defined in `stdio.h`) times in a single process, the names created earlier will be reused.

Between the time a pathname is created and the file is opened, it is possible for some other process to create a file with the same name. However, this will not occur if the other process is using `tempnam()` or `mktemp()` and if the pathname is chosen so as to render duplication by other means unlikely.

**See also**      `fopen()`, `free()`, `open()`, `tmpfile()`, `tmpnam()`, `unlink()`, `stdio.h`.

## tfind - find node in binary search tree

**Syntax**      `#include <search.h>`  
                `void *tfind(const void *key, void *const *rootp, int (*compar) (const void *, const void *));`

**Description** See `tsearch()`.

## \_\_TIME\_\_ - macro for compilation time

**Syntax**      `__TIME__`

**Description** This macro generates the time of compilation of a source file as a string in the form:

`"hh:mm:ss\0"`

where:

`hh`      Hours

`mm`      Minutes

`ss`      Seconds

**Notes**        The format of the time information corresponds to the `asctime()` function.

This macro need not be defined in a header file. Its name is recognized and replaced by the compiler.

**See also**     `asctime()`, `__DATE__`.

## time - get time since the Epoch

Syntax        `#include <sys/types.h>`  
              `#include <time.h>`  
  
              `time_t time(time_t *tloc);`

Description `time()` returns the current time (local time) as the number of seconds that have elapsed since 00:00:00 UTC (Universal Time Coordinated, January 1, 1970).

If *tloc* is non-zero, the result is also stored in the location to which *tloc* points.

*BS2000*

`time()` returns the current time (local time) as the number of seconds that have elapsed since January 1, 1950, 00:00:00. □

### Return value

Time in seconds (see above) if successful.

`(time_t)-1` if an error occurs.  
`errno` is set to indicate the error.

Notes        `time()` fails and its actions are undefined if *tloc* points to an illegal address.

See also     `ctime()`, `time.h`.

## times - get process times

**Syntax**      `#include <sys/times.h>`  
`clock_t times(struct tms *buffer);`

**Description** `times()` fills the `tms` structure pointed to by `buffer` with information on execution times (see `sys/times.h`).

All time specifications are defined in terms of the number of clock ticks used.

The execution times of a terminated child process are included in the `tms_cutime` and `tms_cstime` components of the parent process when the `wait()` function returns the process ID of the terminated child process. If a child process does not wait for its children, their times are not included.

- The `tms_utime` structure member is the CPU time used for the execution of user instructions of the calling process.
- The `tms_stime` structure member is the CPU time used for the execution of system statements on behalf of the calling process.
- The `tms_cutime` structure member is the sum of the `tms_utime` and `tms_cutime` times of the child processes.
- The `tms_cstime` structure member is the sum of the `tms_stime` and `tms_cstime` times of the child processes.

### Return value

Elapsed real time, in clock ticks, since a particular point in time (e.g. since the system was activated). This point in time does not change from one `times()` function call within a process to another. The value returned may exceed the possible value range of type `clock_t` (overflow).

`(clock_t)-1` if an error occurs. `errno` is set to indicate the error.

**Notes**      Portable applications should use the function `sysconf(_SC_CLK_TCLK)` to determine the number of clock ticks per second, since this value may vary from system to system.

**See also**    `exec`, `fork()`, `sysconf()`, `time()`, `wait()`, `sys/times.h`.

## timezone - variable for difference between local time and UTC

**Syntax**      `#include <time.h>`  
`extern long int timezone;`

**Description** The external variable `timezone` contains the difference, in seconds, between Coordinated Universal Time (UTC) and the local standard time. The default for `timezone` is 0 (UTC).  
The environment-specific date and time information is contained in the file `/usr/lib/locale/language/LC_TIME`.

**Notes**        Setting the time during the interval of switching from `timezone` to `altzone` or vice versa can produce unpredictable results.  
The system administrator must change the start and end date for daylight savings time annually if the Julian calendar format is used.

**See also**     `altzone`, `asctime()`, `ctime()`, `daylight`, `environ`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `tzname`, `tzset()`.



## tmpfile - create temporary file

**Syntax**        `#include <stdio.h>`  
                 `FILE *tmpfile(void);`

**Description** `tmpfile()` creates a temporary file and opens an associated data stream.

*BS2000*

`tmpfile()` creates a binary SAM file with default attributes. □

The file is automatically deleted when all links to the file are closed. The file is opened as in `fopen()` for update (`w+`).

The directory in which the temporary file will be created, `{P_tmpdir}`, is defined in `stdio.h` as `/var/tmp`.

### Return value

Pointer to the stream for the created file  
if successful.

Null pointer    if an error occurs. `errno` is set to indicate the error.

### Errors

`tmpfile()` will fail if:

`EINTR`        A signal was caught when executing the `tmpfile()` function.

`EMFILE`        `{OPEN_MAX}` streams are currently open in the calling process.  
`{FOPEN_MAX}` streams are currently open in the calling process.

`ENFILE`        The maximum number of files allowed is currently open in the system.

`ENOSPC`        The directory or file system which would contain the new file cannot be expanded.

### Notes

The program environment determines whether `tmpfile()` is executed for a BS2000 or POSIX file.

Temporary files are not deleted when a program terminates abnormally with `abort()` or `_exit(-1)`.

**See also**        `fopen()`, `tmpnam()`, `unlink()`, `stdio.h`.

## tmpnam - create base name for temporary file

**Syntax**        `#include <stdio.h>`  
`char *tmpnam(char *s)`

**Description** `tmpnam()` generates a string that is a valid and unique file name.

`tmpnam()` generates a different string each time it is called from the same process, up to `{TMP_MAX}` times. If the function is called more than `{TMP_MAX}` times, previously created names are reused.

The implementation behaves as if no library functions call the `tmpnam()` function.

The directory in which temporary files are created, `P_tmpdir`, is defined in `stdio.h` as `/var/tmp`.

### Return value

Pointer to a string  
                  upon successful completion.

Null pointer     if `tmpnam()` was called more than `{TMP_MAX}` times.

If the argument `s` is a null pointer, `tmpnam()` places its result in an internal static area and returns a pointer to that area. Subsequent calls to `tmpnam()` may modify the same area.

If the argument `s` is not a null pointer, it is presumed to point to an array of type `char` with a minimum length of `{L_tmpnam}`; `tmpnam()` writes its result in that array and returns the argument as its return value.

### Notes

If the `tmpnam()` function is called more than `{TMP_MAX}` times a single process, the names created earlier will be reused.

It is the user's responsibility to delete the file pointed to by `*s` when it is no longer needed.

Between the time a pathname is created and the file is opened, it is possible for some other process to create a file with the same name. It may therefore be more practical to use the `tmpfile()` function.

Note that this cannot occur if the other process is using `tmpnam()` or `mktemp()` and if the pathname is chosen so as to render duplication by other means unlikely.

Files created using `tmpnam()` and either `fopen()` or `creat()` are temporary only in the sense that they reside in a directory intended for temporary use and have unique names.

The program environment determines `tmpnam()` is executed for a BS2000 or POSIX file.

**See also**        `fopen()`, `open()`, `tmpnam()`, `tmpfile()`, `unlink()`, `stdio.h`.

## toascii - convert integer to legal value

**Syntax**      `#include <ctype.h>`  
`int toascii(int i);`

**Description** `toascii()` uses the bitwise AND operator ( $i \& 0xFF$ ) to set the first 3 bytes of an integer variable  $i$  to 0 and returns the value of the least significant byte.

`toascii()` is a synonym for `toebcdic()`. On EBCDIC computers, `toascii()` returns a legal value from the EBCDIC character set. If portability to ASCII computers is essential, `toascii()` should be used.

$i$  is an integer variable whose least-significant byte is to be returned.

**Return value**  
Value of the least-significant byte of the variable  $i$  if successful.

**Notes**      `toascii()` does not convert values from other character sets (e.g. ASCII on EBCDIC computers).

**See also**    `isascii()`, `toebcdic()`, `ctype.h`.

## toebcdic - convert integer to legal value *(BS2000)*

**Syntax**      `#include <ctype.h>`  
                `int toebcdic(int i);`

**Description** `toebcdic()` uses the bitwise AND operator ( $i \& 0XFF$ ) to set the first 3 bytes of an integer variable  $i$  to 0 and returns the value of the least significant byte.  
 $i$  is an integer variable whose least-significant byte is to be returned.

**Return value**  
Least-significant byte of the variable  $i$  if successful.

**Notes**        `toebcdic()` is implemented both as a macro and as a function.  
`toebcdic()` does not convert values from other character sets (e.g. ASCII).  
`toebcdic()` is a synonym for `toascii()`. If portability to ASCII computers is essential, `toascii()` should be used instead of `toebcdic()`.

**See also**     `isascii()`, `toascii()`, `ctype.h`.

## \_tolower - convert uppercase letters to lowercase

**Syntax**      `#include <ctype.h>`  
                `int _tolower(int c);`

**Description** `_tolower()` converts the uppercase letter  $c$  to the corresponding lowercase letter.  
 $c$  must be an uppercase letter.

**Return value**  
Lowercase of  $c$ , if  $c$  is an uppercase letter.

**Notes**        `_tolower()` is implemented only as a macro.

**See also**     `tolower()`, `isupper()`, `ctype.h`.

## tolower - convert characters to lowercase

**Syntax**      `#include <ctype.h>`  
              `int tolower(int c);`

**Description** `tolower()` converts the uppercase letter *c* to the corresponding lowercase letter.

**Return value**  
Lowercase of *c* if *c* is an uppercase letter.

**See also**     `strlower()`, `strupper()`, `toupper()`, `setlocale()`, `ctype.h`.

## \_toupper - convert lowercase letters to uppercase

**Syntax**      `#include <ctype.h>`  
              `int _toupper(int c);`

**Description** `_toupper()` converts the lowercase letter *c* to the corresponding uppercase letter.  
*c* must be a lowercase letter.

**Return value**  
Uppercase of *c* if *c* is a lowercase letter.

**Notes**        `_toupper()` is implemented only as a macro .

**See also**     `toupper()`, `islower()`, `ctype.h`.

## toupper - convert characters to uppercase

**Syntax**      `#include <ctype.h>`  
              `int toupper(int c);`

**Description** `toupper()` converts the lowercase letter *c* to the corresponding uppercase letter.

**Return value**  
Uppercase of *c* if *c* is a lowercase letter.

**See also**     `strupper()`, `strlower()`, `tolower()`, `setlocale()`, `ctype.h`.

## towctrans - map wide characters

**Syntax**      `#include <wctype.h>`  
              `wint_t towctrans(wint_t wc, wctrans_t desc);`

**Description** `towctrans()` transforms the wide character *wc* according to the specification *desc*. The current value of the category `LC_CTYPE` must be the same as the one valid for the `towctrans()` call that returned the value *desc*.

The two following calls to `towctrans()` have the same affect as the calls for converting to small or capital letters shown in the corresponding comments:

```
towctrans(wc, wctrans("tolower"))                /* tolower(wc) */  
towctrans(wc, wctrans("toupper"))              /* toupper(wc) */
```

**Return value**  
transformed wide character  
if successful.

**Notes**        In this version of the C runtime system, only 1 byte characters are supported as wide characters.

**See also**     `tolower()`, `toupper()`, `tolower()`, `toupper()`, `wctrans()`

## tolower - convert wide characters to lowercase

**Syntax**      `#include <wchar.h>`

```
wint_t tolower(wint_t wc);
```

**Description** `tolower()` converts the wide character `wc` to the corresponding lowercase letter if `wc` is an uppercase wide-character code.

**Return value**  
Lowercase of `wc` if `wc` is an uppercase letter.

**Notes**      *Restriction*  
This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**    `toupper()`, `setlocale()`, `wchar.h`.

## toupper - convert wide characters to uppercase

**Syntax**      `#include <wchar.h>`

```
wint_t toupper(wint_t wc);
```

**Description** `toupper()` converts the wide character `wc` to the corresponding uppercase letter if `wc` is a lowercase wide-character code.

**Return value**  
Uppercase of `wc` if `wc` is a lowercase letter.

**Notes**      *Restriction*  
This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**    `tolower()`, `setlocale()`, `wchar.h`.

## truncate - set file to specified length

**Name**        **truncate, truncate64**

**Syntax**      `#include <unistd.h>`

```
int truncate (const char *path, off_t length);  
int truncate64 (const char *path, off64_t length);
```

**Description** See `ftruncate()`.

`truncate()` truncates the file specified in *path* to *length* bytes.



## tsearch, tfind, tdelete, twalk - process binary search trees

|             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | <pre>#include &lt;search.h&gt;  void *tsearch (const void *key, void **rootp, int (*compar) (const void *, const void *)); void *tfind (const void *key, void * const *rootp, int (*compar) (const void *, const void *)); void *tdelete (const void *key, void **rootp, int (*compar) (const void *, const void *)); void twalk (const void *root, void(*action) (const void *, VISIT, int));</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Description | <p>tsearch(), tfind(), tdelete() and twalk() manipulate binary search trees. Comparisons are made with a user-supplied <i>compar</i> function. This function is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to or greater than 0, depending on whether the first argument is less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.</p> <p>tsearch() is used to build and access the tree. The <i>key</i> argument is a pointer to an element to be accessed or stored. If there is an entry in the tree that is equal to <i>*key</i> (the value pointed to by the key), a pointer to this found entry is returned. Otherwise, <i>*key</i> is inserted, and a pointer to it is returned. Only pointers are copied, so the calling routine must store the data. The <i>rootp</i> argument points to a variable that points to the root of the tree. A null pointer value for the variable pointed to by <i>rootp</i> denotes an empty tree; in this case, the variable is set to point to the entry that appears at the root of the new tree.</p> <p>Like tsearch(), the tfind() function searches for an entry in the tree and returns a pointer to it if found. If the entry is not found, the tfind() function returns a null pointer. The arguments for tfind() are the same as for tsearch().</p> <p>tdelete() deletes a node from a binary search tree. The arguments are the same as for tsearch(). The variable to which <i>rootp</i> points is changed if the deleted node was the root of the tree. The tdelete() function returns a pointer to the parent of the deleted node, or a null pointer if the node is not found.</p> <p>twalk() traverses a binary search tree. The <i>root</i> argument is a pointer to the root of the tree to be traversed. Any node in a tree may be used as the root for a walk below that node. <i>action</i> is the name of a function to be invoked at each node. This function is called with three arguments. The first argument is the address of the node being visited. The structure pointed to by this argument is unspecified and must not be modified; however, the value of type "pointer-to-node" can be converted to the type "pointer-to-pointer-to-element" to access the element stored in the node.</p> |

The second argument is a value from the enumeration data type `typedef enum { preorder, postorder, endorder, leaf} VISIT`; (defined in the header `search.h`), depending on whether this is the first, second or third time that the node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level 0.

#### Return value

*\*key*            `tsearch()` and `tfind()` : if successful  
                  `tsearch()` : pointer to the inserted item.

Null pointer    `tsearch()` : if there is not enough space available to create a new node.  
                  `tsearch()`, `tfind()` and `tdelete()`: if *rootp* is a null pointer on entry.  
                  `tfind()`: if *\*key* was not found.

Pointer to the parent of the deleted node  
                  `tdelete()` if successful.

#### Notes

The *root* argument to `twalk()` is one level of indirection less than the *rootp* arguments to `tsearch()` and `tdelete()`.

There are two nomenclatures used to refer to the order in which tree nodes are visited. The `tsearch()` function uses `preorder`, `postorder` and `endorder` to refer, respectively, to visiting a node before any of its children, visiting it after its left child and before its right, and visiting a node after both its children. The alternative nomenclature uses `preorder`, `inorder` and `postorder` to refer to the same visits, which could result in some confusion over the meaning of `postorder`.

If the calling function alters the pointer to the root, the results are unpredictable.

#### See also

`bsearch()`, `hsearch()`, `lsearch()`, `search.h`.

## ttyname - find pathname of terminal

**Syntax**      `#include <unistd.h>`

```
char *ttyname(int fildes);
```

**Description** `ttyname()` returns a pointer to a string containing a null-terminated pathname of the terminal associated with file descriptor *fildes*. The return value may point to a static area that is overwritten at each call.

The controlling terminal may have the following names:

`/dev/term/0000, ..., /dev/term/4096` (for block-mode terminals)

`/dev/pts/0, ..., /dev/pts/4096` (for rlogin access)

**Return value**

Pointer to a string  
if successful.

Null pointer    if an error occurs. `errno` is set to indicate the error.

**Errors**      `ttyname()` will fail if:

`EBADF`        *fildes* is not a valid file descriptor.

`ENOTTY`       *fildes* does not point to a terminal.

**Notes**        `ttyname()` is executed only for POSIX files

`ttyname()` is not thread-safe. Use the reentrant function `ttyname_r()` when needed.

**See also**     `isatty()`, `ttyname_r()`, `unistd.h`.

## ttyname\_r - find pathname of terminal (thread-safe)

**Syntax**      `#include <unistd.h>`

```
int ttyname_r(int fildev, char * name, size_t namesize);
```

**Description** The function `ttyname_r()` stores the null-terminated pathname of the terminal associated with file descriptor *fildev* in the data area pointed to by *name*. The data area is *namesize* characters long and should provide enough storage space for the name and the terminating null. The maximum length of the terminal name is `{TTY_NAME_MAX}`.

**Return value**

0                      if successful.

Otherwise the error number.

**Errors**      `ttyname_r()` fails if:

EBADF              *fildev* is not a valid file descriptor.

ENOTTY             *fildev* does not point to a terminal.

ERANGE             the value of *namesize* is smaller than the length of the string returned including the terminating null byte.

**See also**      `ttyname()`, `isatty()`, `unistd.h`.

## ttyslot - find entry of current user in utmp file

**Syntax**      `#include <stdlib.h>`  
`int ttyslot (void);`

**Description** `ttyslot()` returns the index of the current user's entry in the `/var/adm/utmp` file. The entry for the current user is an entry for which the `utline` structure element matches the name of a terminal in `/dev` that is linked to the standard input, standard output or error output (0, 1 or 2).

The returned index is an integer which represents the record number of the entry in the `/var/adm/utmp` file. The index 0 is returned for the first record.

`ttyslot()` is not thread-safe.

### Return value

Index of the entry

if successful.

-1              if an error occurred during the search for the terminal name, or if none of the file descriptors 0, 1 or 2 was assigned to a terminal.

**Notes**        `ttyslot()` will not be supported in the next version of the X/Open standard.

**See also**     `endutxent()`, `ttyname()`, `stdlib.h`.

## twalk - traverse binary search tree

**Syntax**     `#include <search.h>`  
              `void twalk(const void *root, void (*action) (const void *, VISIT, int *));`

**Description** See `tsearch()`.

## tzname - array variable for timezone strings

**Syntax**     `#include <time.h>`  
              `extern char *tzname[2];`

**Description** The external variable `tzname` contains the names of time zones. `tzname` is set by default as follows:

```
char *tzname[2] = { "GMT", "" };
```

**See also**    `altzone`, `asctime()`, `ctime()`, `daylight`, `gmtime()`, `localtime()`, `timezone`, `tzset()`.

## tzset - set timezone conversion information

**Syntax**      `#include <time.h>`

`void tzset(void);`

**Description** `tzset()` uses the contents of the environment variable `TZ` to override the value of the different external variables. The `tzset()` function is called by `asctime()` and may also be called by the user.

`tzset()` scans the contents of the environment variable and assigns the different fields to the respective variable. For example, the complete setting for New Jersey in 1986 would be:

```
EST5EDT4,116/2:00:00,298/2:00:00 or simply: EST5EDT
```

A typical example of a southern hemisphere setting such as the Cook Islands would be:

```
KDT9:30KST10:00,63/5:00,302/20:00
```

In the longer version of the New Jersey example of `TZ`, `tzname[0]` is `EST`; `timezone` will be set to `5 *60 *60`; `tzname[1]` is `EDT`; `altzone` will be set to `4 *60 *60`; the starting date for daylight savings time is the 117th day at 2 a.m.; the ending date is the 299th day at 2 a.m. (using the Julian calendar), and `daylight` will be set to a positive value. The starting and ending times are relative to the daylight savings time. If the starting and ending dates for daylight savings time are not provided, the days applicable to the United States for that year will be used, and the time will be 2 a.m. If only the starting and ending times are not available, the time will be set to 2 a.m.

`tzset()` thus effectively changes the values of the external variables `timezone`, `altzone`, `daylight`, and `tzname`. The `ctime()`, `localtime()`, `mktime()`, and `strftime()` functions will also update these external variables as if they had called `tzset()` at the time specified by the `time_t` or `struct-tm` value that they are converting.

The environment-specific date and time information is contained in the file `/usr/lib/locale/language/LC_TIME`.

`tzset()` sets the external variable `daylight` to 0 if no daylight saving conversion is to be processed for the specified time zone. Otherwise `daylight` is set to a value  $\neq 0$ . The external variable `timezone` is set to the difference, in seconds, between Coordinated Universal Time (UTC) and the local standard time.

**Notes**      If the `TZ` variable is absent from the environment, the applicable values for CET (Central European Time) are used.

**See also**    `altzone`, `asctime()`, `ctime()`, `daylight`, `environ`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `timezone`, `tzname()`.

## ualarm - set interval timer

**Syntax**      `#include <unistd.h>`  
`useconds_t ualarm(useconds_t useconds, useconds_t interval)`

**Description** `ualarm()` sends the SIGALRM signal to the calling process after *useconds* microseconds. Unless it is ignored or caught, the signal terminates the process.

If the *interval* argument is not zero, the SIGALRM signal will be sent to the process every *interval* microseconds after expiry of the timer (e.g. after *useconds* microseconds have elapsed).

Because of delays in the scheduling, the resumption of execution after the signal is caught can be delayed. The longest delay time that can be specified is 2.147.483.647 microseconds.

**Return value**  
The return value is the time remaining until the alarm signal is output.

**Notes**      `ualarm()` is a simplified interface for `setitimer()`.

**See also**    `alarm()`, `setitimer()`, `sleep()`, `unistd.h`.



## ulimit - get and set process limits

**Syntax**      `#include <ulimit.h>`

`long int ulimit (int cmd, ...);`

**Description** `ulimit()` provides for control over process limits. The possible values for *cmd*, which are defined in `ulimit.h`, include:

`UL_GETFSIZE` Returns the file size limit for the process. The limit is specified in 512-byte blocks and is inherited by child processes. Files of any size can be read.

`UL_SETFSIZE` Sets the file size limit for output operations of the process to the value of the second argument, which is interpreted as a `long int`. Any process may lower its own limit, but only a process with appropriate privileges may increase the limit. The return value is the new file size limit.

**Return value**

Value of the requested limit  
if successful.

-1              if an error occurs. `errno` is set to indicate the error.

**Errors**        `ulimit()` will fail and the limit will not be changed if:

`EINVAL`        The argument *cmd* is invalid.

`EPERM`        A process without appropriate privileges is attempting to increase the file size limit.

**Notes**        Since any return value is permitted if the function is successful, an application wishing to check for error conditions should set `errno` to 0 before calling `ulimit()`. If the return value after the function returns is -1 and `errno` is set, an error has occurred.

**See also**     `write()`, `ulimit.h`.

## umask - get and set file mode creation mask

Syntax `#include <sys/stat.h>`

*Optional*

`#include <sys/types.h> □`

`mode_t umask (mode_t cmask);`

Description `umask()` sets the file mode creation mask of the process to *cmask* and returns the previous value of the mask. Only the file permission bits of *cmask* (see `sys/stat.h`) are used; the other bits are ignored.

The file mode creation mask of the process is used by the functions `open()`, `creat()`, `mkdir()` and `mkfifo()` to remove access permissions in *mode*. Bit positions that are set in *cmask* are cleared in the access permissions of the created file.

The state of the mask before the first call to `umask()`, including all other bits, can be restored by a subsequent call to `umask()` with the return value of the first call as the argument.

Return value

If the user ID is 0, the default value is 022 (octal); otherwise, 066. Previous value of the file mode creation mask if successful. The other bits are ignored. A subsequent call to `umask()` with the return value of the preceding call as *cmask* will reset the mask to the same state as before the first call.

Notes `umask()` is executed only for POSIX files

See also `creat()`, `mkdir()`, `mkfifo()`, `open()`, `sys/stat.h`, `sys/types.h`.

## umount - unmount file system *(extension)*

**Syntax**      `#include <sys/mount.h>`  
`int umount(const char *path);`

**Description** `umount()` can be used to unmount a file system that was mounted earlier with `mount()` under the directory pointed to by *path* (mount point). The *path* argument may point to a block-special file or a directory. After unmounting the file system, the directory in which the file system was mounted reverts to its ordinary interpretation.

**Return value**  
0                upon successful completion.  
-1                if an error occurs. `errno` is set to indicate the error.

**Errors**        `umount()` will fail if:

- `EBUSY`         A file in *path* is being used.
- `EFAULT`        *path* points to an invalid address.
- `EINVAL`        *path* does not exist, or  
                  *path* has not been mounted.
- `ELOOP`         Too many symbolic links were encountered when resolving the path pointed to by *path*.
- `ENAMETOOLONG`  
                  *path* is longer than `{PATH_MAX}`, or the length of a *path* component exceeds `{NAME_MAX}`.
- `ENOTBLK`       *path* is not a block-special file.
- `EPERM`         The effective user ID is not that of a process with appropriate privileges.
- `EREMOTE`        *path* points to a remote pathname.

**Notes**         `umount()` may only be called under the effective user ID of a process with appropriate privileges.  
`umount()` is executed only for POSIX files.

**See also**      `mount()`, `sys/mount.h`.

## uname - get basic data on current operating system

**Syntax**      `#include <sys/utsname.h>`  
`int uname(struct utsname *name);`

**Description** `uname()` obtains basic information on the current operating system and stores it in the structure pointed to by *name*.

`uname()` uses the `utsname` structure defined in `sys/utsname.h`. The members of the structure are the `char` arrays `sysname`, `nodename`, `release`, `version` and `machine`. The name of the current operating system is entered in the array `sysname`. Similarly, `nodename` contains the name that the system is known by on a communications network. The arrays `release` and `version` contain the release number and release date of the operating system, and the array `machine` contains a name that identifies the hardware on which the system is running.

**Return value**

Non-negative value  
                  if successful.

-1  
                  if an error occurs. `errno` is set to indicate the error.

**Errors**      `uname()` will fail if:

*Extension*  
`EFAULT`      *name* is an invalid address. □

**Notes**      The inclusion of the `nodename` member in this structure does not mean that this information is sufficient for addressing communications networks.

**See also**    `sys/utsname.h`.

## ungetc - push byte back onto input stream

Syntax `#include <stdio.h>`  
`int ungetc(int c, FILE *stream);`

Description `ungetc()` converts the previously read byte *c* to type `unsigned char` and pushes it back onto the input stream pointed to by *stream*. The pushed-back bytes will be returned by subsequent reads on that stream in reverse order. A successful intervening call to a file-positioning function (`fseek()`, `fsetpos()` or `rewind()`) for the same data stream will delete any pushed-back bytes for the stream. The external storage associated with the stream remains unchanged.

### *BS2000*

A call to one of the following functions cancels the effects of the `ungetc` call (e.g. backward positioning): `fseek()`, `fsetpos()`, `lseek()`, `rewind()`, `fflush()`. □

One byte of pushback is guaranteed. If `ungetc()` is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the pushback operation may fail. A maximum of `{BUFSIZE}` bytes can be pushed back in the C runtime system (see `stdio.h`).

If the value of *c* is equal to the macro `EOF`, the operation will fail and the input stream will remain unchanged.

A successful call to `ungetc()` clears the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back bytes will be the same as it was before the bytes were pushed back. The file-position indicator is decremented by each successful call to `ungetc()`; if its value was 0 before a call, its value will be indeterminate after the call.

### Return value

Byte pushed back  
upon successful completion.

`EOF` if *c* is equal to `EOF` or if an error occurs.

- Notes      At least one byte must always have been read from the file before the first `ungetc()` call. The program environment determines `ungetc()` is executed for a BS2000 or POSIX file.
- BS2000*  
If a byte other than the one just read is pushed back onto the buffer when accessing BS2000 files, the behavior will depend on whether KR or ANSI functionality is set:
- KR functionality (only available with C/C++ versions lower than V3): when the buffer contents are written to the external file, the original data is not changed.
  - ANSI functionality: when the buffer contents are written to the external file, the original data is not changed, i.e. the original data prior to the `ungetc()` call is always written to the external file. □

See also    `fseek()`, `getc()`, `fsetpos()`, `read()`, `rewind()`, `setbuf()`, `stdio.h`.

## ungetwc - push wide character back onto input stream

Syntax `#include <wchar.h>`

*Optional*

`#include <stdio.h>` □

`wint_t ungetwc(wint_t wc, FILE *stream);`

Description `ungetwc()` pushes the character corresponding to the wide character code *wc* back onto the input stream pointed to by *stream*. The pushed-back characters will be returned by subsequent reads on that stream in reverse order. A successful intervening call to a file-positioning function (`fseek()`, `fsetpos()` or `rewind()`) for the same data stream deletes the pushed-back characters for the stream. The external storage associated with the data stream remains unchanged.

One byte of pushback is guaranteed. If `ungetwc()` is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the pushback operation may fail.

If the value of *wc* is equal to the macro `WEOF`, the operation will fail and the input stream will remain unchanged.

A successful call to `ungetwc()` clears the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back bytes will be the same as it was before the bytes were pushed back. The file-position indicator is decremented by each successful call to `ungetwc()`; if its value was 0 before a call, its value will be indeterminate after the call.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

Return value

Pushed back wide character  
upon successful completion.

`WEOF` if the wide character could not be pushed back. `errno` is set to indicate the error.

Errors `ungetwc()` will fail if:

*Extension*

`EINVAL` An attempt was made to access a BS2000 file.

See also `fseek()`, `fsetpos()`, `read()`, `rewind()`, `setbuf()`, `stdio.h`, `wchar.h`.

## unlink - remove link

Syntax `#include <unistd.h>`

```
int unlink(const char *path);
```

Description `unlink()` removes the directory entry specified by the pathname pointed to by *path*, and decrements the link count of the file referenced by the directory entry. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed, and the file is no longer be accessible. If one or more processes have the file open when the last link is removed, the space occupied by the file is not released until all references to the file have been closed. If *path* is a symbolic link, the symbolic link is removed.

*path* should not name a directory unless the process has appropriate privileges. Applications should use `rmdir()` to remove directories.

Upon successful completion, `unlink()` marks the `st_ctime` and `st_mtime` structure components of the parent directory for update. If the file's link count is not 0, the `st_ctime` structure component of the file is also marked for update.

### BS2000

`unlink()` continues to be supported for compatibility reasons; it has the same effect as `remove()`, i.e. deletes the file (see `remove()`). □

### Return value

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error. The file named by *path* is not changed.

### Errors

`unlink()` will fail if:

EACCES Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the directory entry to be removed.

EBUSY The entry to be removed is the mount point for a mounted file system.

### Extension

EFAULT *path* points outside the allocated address space of the process.

EINTR A signal was caught during the `unlink()` system call.

ELOOP Too many symbolic links were encountered in resolving *path*. □

ENAMETOOLONG

The length of the *path* argument exceeds `{PATH_MAX}` or a component of *path* is longer than `{NAME_MAX}`.



|         |                                                                                                             |
|---------|-------------------------------------------------------------------------------------------------------------|
| ENOENT  | The named file does not exist or is an empty string. The user is not a system administrator.                |
| ENOTDIR | A component of <i>path</i> is not a directory.                                                              |
| EPERM   | The file named by <i>path</i> is a directory, and the calling process does not have appropriate privileges. |
| EROFS   | The directory entry to be unlinked is part of a read-only file system.                                      |

**Notes** `rmdir()` is used to delete a directory.

The program environment determines whether `unlink()` is executed for a BS2000 or POSIX file.

**See also** `close()`, `link()`, `remove()`, `rmdir()`, `unistd.h`.

## unlockpt - remove lock from master/slave pseudoterminal pair

**Syntax**      `#include <stdlib.h>`  
`int unlockpt (int fildes);`

**Description** The `unlockpt()` function unlocks the slave pseudoterminal associated with the master pseudoterminal specified in *fildes*.

Portable applications must call `unlockpt()` before they open the slave side of a pseudoterminal device.

**Return value**

|    |                                                             |
|----|-------------------------------------------------------------|
| 0  | if successful.                                              |
| -1 | otherwise. <code>errno</code> is set to indicate the error. |

**Errors**      `unlockpt()` will fail if:

|        |                                                                        |
|--------|------------------------------------------------------------------------|
| EBADF  | The <i>fildes</i> argument is not a file descriptor open for writing.  |
| EINVAL | The <i>fildes</i> argument is not assigned to a master pseudoterminal. |

**See also**      `grantpt()`, `open()`, `ptsname()`, `stdlib.h`.

## usleep - suspend process for defined interval

**Syntax**        `#include <unistd.h>`  
`int usleep(useconds_t useconds);`

**Description** Suspend the current process for *useconds* microseconds. The actual length of time for which the process is suspended can be longer than *useconds* microseconds due to other activities in the system or because of the time required for processing the call.

*useconds* must be < 1 000 000. If *useconds* = 0, then `usleep()` has no effect.

The routine is implemented by setting the interval timer of the process and then waiting until it expires. The previous status of the timer is saved and restored. If the wait time or 'sleep time' exceeds the period until expiry of the previous timer, the process is only suspended until the signal would have occurred, and the signal is sent shortly before this sleep time expires.

If threads are used, then the function affects the process or a thread in the following manner: `usleep()` causes the current thread to be suspended until a specified time expires or a signal is sent to the thread.

**Return value**  
0                if successful.  
-1               otherwise.

**Notes**        `usleep()` is supported for historical reasons. `setitimer()` should be used instead.

**See also**     `alarm()`, `getitimer()`, `sigaction()`, `sleep()`, `unistd.h`.

## utime - set file access and modification times

Syntax `#include <utime.h>`

*Optional*

`#include <sys/types.h>` □

`int utime(const char *path, const struct utimbuf *times);`

Description `utime()` sets the access and modification times of the file specified by the *path* argument.

If *times* 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 match the owner of the file, or the process must have write permission to the file or have appropriate privileges to use `utime()` in this manner.

If *times* is not a null pointer, then *times* is interpreted as a pointer to a `utimbuf` structure, and the access and modification times are set to the values contained in this structure. Only a process with an effective user ID that matches the file's owner or a process with appropriate privileges may use `utime()` this way.

The times in the structure `utimbuf` are measured in seconds since 00:00:00 GMT, January 1, 1970 (see `utime.h`).

Upon successful completion, `utime()` marks the time of the last change to the file, `st_ctime`, for update (see `sys/stat.h`).

Return value

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

Errors `utime()` will fail if:

`EACCES` Search permission is denied for a component of the path; or the effective user ID does not match that of a system administrator or the owner of the file, *times* is a null pointer, and write access is denied.

*Extension*

`EFAULT` *times* is not null and points outside the allocated space of the process, or *path* points outside the allocated space of the process.

`EINTR` A signal was caught during the system call `utime()`.

`EINVAL` An attempt was made to access a BS2000 file.

`ELOOP` Too many symbolic links were encountered in resolving *path*. □

ENAMETOOLONG

The length of *path* exceeds `{PATH_MAX}` or the length of a component of *path* exceeds `{NAME_MAX}`.

ENOENT

The named file does not exist.

ENOTDIR

A component of the path is not a directory.

EPERM

The effective user ID does not match that of a system administrator or the owner of the file and *times* is a null pointer.

EROFS

The file system containing the file is mounted as a read-only file system.

**Notes** `utime()` is executed only for POSIX files

**See also** `stat()`, `sys/types.h`, `utime.h`.

## utimes - set file access time and file modification time

Syntax `#include <sys/time.h>`

```
int utimes(const char *path, const struct timeval times[2]);
```

Description `utimes()` sets the access and modification times of the file pointed to by *path* to the values specified in *times*.

The function allows time specifications accurate to the microsecond.

The *times* argument is an array consisting of two structures of type `timeval`. The access time is set to the value of the first element and the modification time to the value of the second element. The times in the `timeval` structure are measured in seconds and microseconds since 00:00:00 GMT, January 1, 1970 (see `utime.h`).

If *times* is the null pointer, the access and modification times are set to the current time. If `utimes()` is to be used in this way, the process must be the owner of the file, must have write permission for the file or must be a process with special permissions.

On successful completion, `utimes()` marks the `st_ctime` field for update (see `sys/stat.h`).

Return value

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

Errors

`utimes()` will fail if:

`EACCES` Search permission is denied for a component of the path, or *times* is a null pointer and the effective user ID is not that of the system administrator or the owner of the file, and write access is refused.

*Extension*

`EFAULT` *times* is non-zero and points outside the allocated address space, or *path* points outside the allocated address space of the process.

`EINTR` A signal was caught during the `utime()` system call.

`EINVAL` An attempt was made to access a BS2000 file.

`ELOOP` Too many symbolic links were encountered in resolving path. □

`ENAMETOOLONG`

The length of *path* exceeds `{PATH_MAX}` or the length of a component of *path* exceeds `{NAME_MAX}`.

|         |                                                                                                                      |
|---------|----------------------------------------------------------------------------------------------------------------------|
| ENOENT  | The named file does not exist.                                                                                       |
| ENOTDIR | A component of the path is not a directory.                                                                          |
| EPERM   | The effective user ID is not that of the system administrator or the user of the file, and <i>times</i> is not zero. |
| EROFS   | The file system containing the file is mounted as a read-only file system.                                           |

**See also** `sys/time.h`.

## va\_arg - process variable argument list

Syntax `#include <stdarg.h>`

*Optional*

`#include <varargs.h> □`

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

Description The `va_arg`, `va_start` and `va_end` macros allow portable procedures that accept variable argument lists, as defined in `stdarg.h`, to be written. They are used to process a list of arguments which may vary in number and type at each function call.

`va_arg` returns the data type and value of the next argument in a variable argument list *ap*, starting with the first argument. Technically speaking, the macro expands into an expression of the data type and value of the argument.

The variable argument list to which *ap* points must be initialized with `va_start` before the first call to `va_arg`. Each invocation of `va_arg` modifies *ap* so that the value of the next argument in turn is returned.

*ap* is a pointer to the argument list initialized with `va_start` before `va_arg` is called for the first time.

*type* is a type name matching the type of the current argument. Any C data type, for which a pointer to an object of the specified *type* is defined by simply appending an `*` to *type*, is allowed. Array and function types, for example, are invalid.

If there is no next argument or if *type* does not match the current argument, the behavior is undefined.

Return value

Value of the first argument

when `va_arg()` is called for the first time after `va_start`. This argument comes after the last "named" argument *parmN* in the formal parameter list (see also `va_start()`). Subsequent calls return the remaining argument values in succession.

Notes

Compatibility of argument types is supported by the C runtime system to the extent that similar types are stored in the same way in the parameter list, i.e.: all unsigned types (including `char`) are represented as `unsigned int` (right-justified in a word), and all other integer types are represented as `int` (right-justified in a word). `float` is represented as `double` (right-justified in a doubleword).

The `va_end` macro must be called before returning from a function whose argument list was processed with `va_arg`.

See also

`va_start()`, `va_end()`, `stdarg.h`, `varargs.h`.



## va\_end - end variable argument list

Syntax `#include <stdarg.h>`

*Optional*

`#include <varargs.h>`

`void va_end(va_list ap);`

Description The `va_end`, `va_start` and `va_arg` macros allow portable procedures that accept variable argument lists, as defined in `stdarg.h`, to be written. They are used to process a list of arguments which may vary in number and type at each function call.

`va_end` performs cleanup activities on the variable argument list `ap`. This macro must be called before returning from a function whose argument list has been processed with `va_start` and `va_arg`.

`ap` is the argument list that was processed. If it is to be used again, the argument list must be re-initialized with `va_start`, as `va_end` changes the argument list `ap`.

See also `va_arg()`, `va_start()`, `stdarg.h`, `varargs.h`.

## va\_start - initialize variable argument list

Syntax `#include <stdarg.h>`

*Optional*

`#include <varargs.h> □`

`void va_start(va_list ap, parmN);`

Description The `va_start`, `va_arg` and `va_end` macros allow portable procedures that accept variable argument lists, as defined in `stdarg.h`, to be written. They are used to process a list of arguments which may vary in number and type at each function call.

`va_start` initializes the variable argument list `ap` for subsequent calls to `va_arg` and `va_end`.

`ap` is a pointer to the argument list.

`parmN` is the name of the last argument of the variable argument list. Functions which process variable argument lists must define at least one argument.

Return value

Number of output characters  
if successful.

0 if an error occurs.

Notes If `parmN` has an invalid data type or its type does not match the current argument, the behavior is undefined.

Compatibility of argument types is supported by the C runtime system to the extent that similar types are stored in the same way in the parameter list, i.e.: all unsigned types (including `char`) are represented as `unsigned int` (right-justified in a word), and all other integer types are represented as `int` (right-justified in a word). `float` is represented as `double` (right-justified in a doubleword).

See also `va_arg()`, `va_end()`, `stdarg.h`, `varargs.h`.

## valloc - request memory aligned with page boundary

**Syntax**      `#include <stdlib.h>`

```
void *valloc (size_t size);
```

**Description** `valloc()` has the same effect as `malloc()`, except that the allocated memory area is aligned with the page border, i.e. an integer multiple of the return value of `sysconf(_SC_PAGESIZE)`.

If `size = 0`, `valloc()` returns a null pointer. `errno` is not set in this case.

**Return value**

Pointer to the allocated memory area  
if successful.

Null pointer      otherwise. `errno` is set to indicate the error.

**Errors**      `valloc()` will fail if

`ENOMEM`      There is not enough memory available.

**Notes**      Instead of `valloc()`, applications should use `malloc()` or `mmap()`. In systems with a large page size, it may not be possible to call `valloc()` successfully.

`valloc()` will no longer be supported in the next version of the X/Open standard.

**See also**      `malloc()`, `sysconf()`, `stdlib.h`.

## vfork - generate new process in virtual memory

**Syntax**      `#include <unistd.h>`  
                `pid_t vfork (void);`

**Description** `vfork()` is mapped to `fork()`. See the relevant section for a description.

**Return value**

|          |                                                                                                                               |
|----------|-------------------------------------------------------------------------------------------------------------------------------|
| 0 or PID | if successful. 0 is returned to the child process, and the process ID of the child process is returned to the parent process. |
| -1       | to the parent process if an error occurs. No child process is generated. <code>errno</code> is set to indicate the error.     |

**Errors**      `vfork()` will fail if.

|        |                                                                                                                                                                               |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EAGAIN | The system-dependent limit to the maximum number of processes possible throughout the system or per user was exceeded. These limits are defined when the system is generated. |
| ENOMEM | The swap area is not large enough for the new process.                                                                                                                        |

**See also**     `exec()`, `exit()`, `fork()`, `wait()`, `unistd.h`.

## vfprintf, vprintf, vsprintf - formatted output of variable argument list

**Syntax**      `#include <stdarg.h>`  
                 `#include <stdio.h>`

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

**Description** `vfprintf()`, `vprintf()` and `vsprintf()` correspond to the functions `fprintf()`, `printf()` and `sprintf()`, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by `stdarg.h`. The number of arguments in the argument list and their types are not known at the time of compilation.

Since the `vprint` functions invoke the `va_arg` macro, but not the `va_end` macro, the value of `ap` after the return is indeterminate.

**Return value**  
See `fprintf()`.

**Errors**      See `fprintf()`.

**Notes**      The macro `va_end(ap)` should be called after using these functions in order to reset the pointer `ap` to a defined value so that any subsequent calls to these functions will have the correct initial values.

`vfprint()` always starts with the first argument in the variable argument list. It is possible to start output from any particular argument by issuing the appropriate number of `va_arg` calls before calling the `vfprintf()` function. Each `va_arg` call advances the position in the argument list by one argument.

The program environment determines whether `vprintf()` is executed for a BS2000 or POSIX file.

### *BS2000*

The ANSI syntax of the format string applies both in KR mode (only available with C/C++ versions lower than V3) and in ANSI modes (as defined by the `LANGUAGE-STANDARD` operands of the `SOURCE-PROPERTIES` option). □

**See also**      `fprintf()`, `stdarg.h`, `stdio.h`, `varargs.h`.

## vfwprintf - formatted output of wide characters

**Syntax**

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

int vfwprintf(FILE *dz, const wchar_t *format, va_list arg);
```

**Description** A detailed description can be found under `fwprintf()`.

## vprintf - formatted output to standard out

**Syntax**

```
#include <stdio.h>

int vprintf(const char *format, va_list arg);
```

**Description** `vprintf()` is the same as the `printf()` function. In contrast to `printf()`, `vprintf()` allows for the output of arguments whose number and data type are not known at the time of compilation.

`vprintf()` is used in functions that can be passed a different format string as well as different arguments for output from the caller. The format string *format* stands for the formal parameter list of the function definition and a variable argument list `"..."`.

*format* is a format string just like for `printf()` with ANSI functionality (see `printf()`).

`vprintf()` processes an argument list *arg* with successive internal `va_arg` calls and writes the arguments to the standard output `stdout` according to the format string *format*. The variable argument list *arg* must be initialized before calling `vprintf()` using the `va_start` macro.

**Return value**

Number of characters to be output  
if successful.

Integer < 0 if an error occurs

**Notes** `vprintf()` always starts with the first argument in the variable argument list. It is possible to start output from any particular argument by issuing the appropriate number of `va_arg` calls before calling the `vprintf()` function. Each `va_arg` call advances the position in the argument list by one argument.

`vprintf()` does not call the `va_end` macro. Since `vprintf()` uses the `va_arg` macro, the value of *arg* is undefined upon returning.

**See also** `fwprintf()`, `vsprintf()`

## vsprintf - formatted output to a string

**Syntax**      `#include <stdio.h>`  
`int vsprintf(char *s, const char *format, va_list arg);`

**Description** `vsprintf()` is the same as the `sprintf()` function. In contrast to `sprintf()`, `vsprintf()` allows for the output of arguments whose number and data type are not known at the time of compilation.

`vsprintf()` is used in functions that can be passed a different format string as well as different arguments for output from the caller. The format string *format* stands for the formal parameter list of the function definition and a variable argument list ", ...".

`vsprintf()` processes an argument list *arg* with successive internal `va_arg` calls and writes the arguments to the string *s* according to the format string *format*. The variable argument list *arg* must be initialized before calling `vsprintf()` using the `va_start` macro. The function has the following parameters:

`char *s`  
    Pointer to the resulting string. `vsprintf()` terminates the string with the null byte (`\0`).

`const char *format`  
    Format string like for `printf()` with ANSI functionality (see `printf()` for a description).

The following difference exists with respect to white space characters (`\n`, `\t`, etc.):

`vsprintf()` enters the EBCDIC value of the control character in the resulting string. Only when it is output to a text file will the control characters be converted to their appropriate effect in accordance with the type of text file (see page 84).

`va_list arg`  
    Pointer to the variable argument list that was initialized with `va_start`.

**Return value**  
    Number of characters stored in *s*. The terminating null byte (`\0`) generated by `vsprintf()` is not counted.

**Notes** `vsprintf()` always starts with the first argument in the variable argument list. It is possible to start output from any particular argument by issuing the appropriate number of `va_arg` calls before calling the `vsprintf()` function. Each `va_arg` call advances the position in the argument list by one argument.

`vsprintf()` does not call the `va_end` macro. Since `vsprintf()` uses the `va_arg` macro, the value of *arg* is undefined upon returning.

The behavior is undefined for overlapping memory areas.

**See also**      `vfprintf()`, `vprintf()`

## vswprintf - formatted output of wide characters

Syntax      `#include <stdarg.h>`  
             `#include <stdio.h>`  
             `#include <wchar.h>`  
  
             `int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);`

Description A detailed description can be found under `fwprintf()`.

## vwprintf - formatted output of wide characters

Syntax      `#include <stdarg.h>`  
             `#include <wchar.h>`  
  
             `int vwprintf(const wchar_t *format, va_list arg);`

Description A detailed description can be found under `fwprintf()`.



## wait, waitpid - wait for child process to stop or terminate

Syntax `#include <sys/wait.h>`

*Optional*

`#include <sys/types.h> □`

`pid_t wait (int *stat_loc);`

`pid_t waitpid (pid_t pid, int *stat_loc, int options);`

**Description** `wait()` and `waitpid()` allow the calling process to obtain status information on one of its child processes. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

`wait()` suspends execution of the calling process until the exit status for one of its child processes is available, or until delivery of a signal whose action is either to execute a signal-handling function or `SIG_DFL`. If the status information is available before the call to `wait()`, the function will return immediately.

`waitpid()` behaves identically to the `wait()` function if the value of `pid` is `(pid_t)-1` and the value of `options` is 0. Otherwise, its behavior is modified by the values of the `pid` and `options` arguments.

`pid` specifies a set of child processes for which status is requested. `waitpid()` will only return the status of a child process from this set:

- If `pid` is equal to `(pid_t)-1`, the status is requested for any child process. In this respect, `waitpid()` is then equivalent to `wait()`.
- If `pid` is greater than 0, it specifies the process ID of a single child process for which the status is requested.
- If `pid` is 0, the status is requested for any child process whose process group ID is equal to that of the calling process.
- If `pid` is less than `(pid_t)-1`, the status is requested for any child process whose process group ID is equal to the absolute value of `pid`.

`options` is constructed from the bitwise-inclusive OR of zero or more of the following flags, which are defined in the header `sys/wait.h`.

**WCONTINUED** `waitpid()` determines the status of a child process specified by `pid` which is continued and whose status has not been queried since being resumed after a job control stop.

**WNOHANG** `waitpid()` will not suspend execution of the calling process if the status is not immediately available for one of the child processes specified by `pid`.

**WUNTRACED** The status of any child processes specified by *pid* that are stopped, and whose status has not yet been returned since they stopped, will also be reported to the calling process.

If `wait()` or `waitpid()` returns because the status of a child process is available, the return value of these functions will be the process ID of the child process. In this case, if the value of *stat\_loc* is not a null pointer, the status information will be stored in the location pointed to by *stat\_loc*.

If the status returned is from a terminated child process that returned the value 0 from `main()` or passed 0 as the status argument to `_exit()` or `exit()`, the value stored at the address pointed to by *stat\_loc* will be 0. Regardless of its value, this information may be interpreted using the following macros, which are defined in `sys/wait.h` and evaluate to integral expressions; the *stat\_val* argument is the integer value pointed to by *stat\_loc*.

**WIFEXITED(*stat\_val*)**

Evaluates to a non-zero value (true in C) if the status was returned for a child process that terminated normally.

**WEXITSTATUS(*stat\_val*)**

If the value of `WIFEXITED(stat_val)` is non-zero, this macro evaluates to the low-order 8 bits of the exit status that the child process passed to `_exit()` or `exit()`, or the value the child process returned from `main()`.

**WIFSIGNALED(*stat\_val*)**

Evaluates to non-zero value if the status was returned for a child process that terminated due to the receipt of a signal that was not caught (see also `signal.h`).

**WTERMSIG(*stat\_val*)**

If the value of `WIFSIGNALED(stat_val)` is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

**WIFSTOPPED(*stat\_val*)**

Evaluates to a non-zero value if the status was returned for a child process that is currently stopped.

**WSTOPSIG(*stat\_val*)**

If the value of `WIFSTOPPED(stat_val)` is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

**WIFCONTINUED(*stat\_val*)**

Calculates a non-zero value if the status for a child process that was resumed after a job control stop is returned.

If the status stored at the location *stat\_loc* was stored there by a `waitpid()` call which:

- specified the flag `WUNTRACED` but not the flag `WCONTINUED`:  
then precisely one of the macros `WIFEXITED(*stat_loc)`, `WIFSIGNALED(*stat_loc)` or `WIFSTOPPED(*stat_loc)` returns a non-zero value.
- specified the flags `WUNTRACED` and `WCONTINUED`:  
then precisely one of the macros `WIFEXITED(*stat_loc)`, `WIFSIGNALED(*stat_loc)` and `WIFSTOPPED(*stat_loc)` or `WIFCONTINUED(*stat_loc)` returns a non-zero value.
- specified neither the flag `WUNTRACED` nor the flag `WCONTINUED`,  
or was stored by a call of the `wait()` function:  
then precisely one of the macros `WIFEXITED(*stat_loc)` or `WIFSIGNALED(*stat_loc)` returns a non-zero value.
- specified the flag `WCONTINUED` but not the flag `WUNTRACED`,  
or was stored by a call of the `wait()` function:  
precisely one of the macros `WIFEXITED(*stat_loc)`, `WIFSIGNALED(*stat_loc)` or `WIFCONTINUED(*stat_loc)` returns a non-zero value.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes will be assigned a new parent process ID, namely that of the system process `init`.

If threads are used, the `wait()` and `waitpid()` functions affect the process or a thread in the following manner: The calling thread is suspended until the status information is available.

## Return value

Process ID of the child process

- if `wait()` or `waitpid()` returns because the status of a child process is available.
- 1 if the `wait()` or `waitpid()` returns because a signal is delivered. `errno` is set to `EINTR`.
- 0 if `waitpid()` was invoked with the flag `WNOHANG` set in the *options* argument and the function has at least one child process specified by *pid*.
- (`pid_t`)-1 if an error occurs. `errno` is set to indicate the error.

Errors      `wait()` will fail if:

ECHILD      The calling process has no existing unwaited-for child processes.

EINTR      The function was interrupted by a signal. The value of the object pointed to by *stat\_loc* is undefined in this case.

`waitpid()` will fail if:

ECHILD      The process specified with *pid* or the process group does not exist, or is not a child process of the calling process.

EINTR      The function was interrupted by a signal. The value of the object pointed to by *stat\_loc* is undefined in this case.

EINVAL      *options* is not valid.

See also    `exec`, `exit()`, `fork()`, `sys/types.h`, `sys/wait.h`.

## wait3 - wait for status change of child processes

**Syntax**      `#include <sys/wait.h>`

```
pid_t wait3(int *stat_loc, int options, struct rusage *resource_usage);
```

**Description** `wait3()` returns status information on the specified child process to the calling process.

**The call**

```
wait3(stat_loc, options, resource_usage);
```

is equivalent to the call

```
waitpid( (pid_t)-1, stat_loc, options);
```

except that on successful execution in the specified `rusage` structure `resource_usage`, the status information for the child process identified by the return value is entered.

`wait3()` is not thread-safe.

If threads are used, the `wait()` and `waitpid()` functions affect the process or a thread in the following manner: `wait3()` returns status information on the specified child process to the calling thread.

**Return value**

see `waitpid()`.

In addition to the errors specified for `waitpid()`, `wait3()` will fail if:

**ECHILD**      For the calling process there are no child processes which are not waited for, or  
the group of processes specified by the `pid` argument can never acquire the status specified by `options`.

**Notes**      If a parent process is terminated without waiting for its child processes, the initialization process (process ID = 1) takes over the child processes.

**See also**    `exec`, `exit()`, `fork()`, `pause()`, `sys/wait.h`.

## waitid - wait for status change of child processes

Syntax `#include <wait.h>`

```
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Description The calling process is suspended by `waitid()` until one of the child processes changes its status. The current status of the relevant child process is entered in the structure pointed to by `infop`. If a child process has changed its status before the `waitid()` call, `waitid()` returns immediately.

The `idtype` and `id` arguments indicate which child processes `waitid()` is to wait for.

- If `idtype` is `P_PID`, then `waitid()` waits for the child process with the process ID (`pid_t`) `id`.
- If `idtype` is `P_PGID`, then `waitid()` waits for one of the child processes with the process group ID (`pid_t`) `id`.
- If `idtype` is `P_ALL`, then `waitid()` waits for any child process and `id` is ignored.

The `options` argument is used to specify which status changes `waitid()` is to wait for. The status changes are specified via bitwise ORing of the following flags:

|                         |                                                                                                                                                                                                 |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>WEXITED</code>    | waits for processes to exit.                                                                                                                                                                    |
| <code>WTRAPPED</code>   | waits for traced processes to be interrupted or reach a breakpoint (see <code>ptrace()</code> ).                                                                                                |
| <code>WSTOPPED</code>   | waits and returns the process status of a child process which stopped after a signal was received.                                                                                              |
| <code>WCONTINUED</code> | returns the status of a child process that was suspended and then resumed.                                                                                                                      |
| <code>WNOHANG</code>    | returns immediately if there are no child processes to be waited for.                                                                                                                           |
| <code>WNOWAIT</code>    | keeps the process whose status was returned in <code>infop</code> in a wait state. The status of this process is not affected. This process can be waited for again when the call is completed. |

`infop` must point to a `siginfo_t` structure, as it is defined in `siginfo()`. If `waitid()` returns because it has found a child process which fulfils the conditions specified in `idtype` and `options`, the system enters the status of this process in `siginfo_t`. The structure element `si_signo` always has the value `SIGCHILD`.

If threads are used, the `wait()` and `waitpid()` functions affect the process or a thread in the following manner: The calling thread is suspended until the status of one of the child processes changes.

## Return value

- 0 if `waitid()` returns because of a status change of a child process
- 1 otherwise. `errno` is set to indicate the error.

Errors `waitid()` will fail if at least one of the following occurs:

- ECHILD For the calling process there are no child processes which are not being waited for.
- EINTR `waitid()` was interrupted because the calling process has received a signal.
- EINVAL An invalid value was passed for *options*, or *idtype* and *id* indicate an invalid number of process set.
- EFAULT *infp* points to an invalid address.

See also `exec`, `exit()`, `wait()`, `sys/wait.h`

## wrtomb - convert wide characters to multi-byte characters

**Syntax**      `#include <wchar.h>`

```
size_t wrtomb(char *s, wchar_t wc, mbstate_t *ps);
```

**Description** If *s* is a null pointer, `wrtomb()` corresponds to the call `wrtomb(buf, L'\0', ps)` where *buf* designates an internal buffer.

If *s* is not a null pointer, `wrtomb()` determines how many bytes are required to represent the multi-byte character corresponding to *wc*. Any Shift sequences are also taken into account. The resulting bytes are written to the array whose first element is pointed to by *s*. A maximum of `{MB_CUR_MAX}` bytes are written.

If *wc* is the null character, a null byte is written that can precede a Shift sequence that restores the initial conversion state.

The final state corresponds to the “initial conversion” state.

**Return value**

`(size_t)-1` if *wc* does not represent a valid wide character. The value of the `EILSEQ` macro is written to `errno`. The conversion status is undefined.

Otherwise the number of bytes written to the array *\*s*.

**Notes** This version of the C runtime system only supports 1-byte characters as wide character codes.

**See also** `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`



## wscat - concatenate two wide character strings

**Syntax**      `#include <wchar.h>`

```
wchar_t *wscat(wchar_t *ws1, const wchar_t *ws2);
```

**Description** `wscat()` appends a copy of the wide character string `ws2` to the end of the wide character string `ws1` and returns a pointer to `ws1`.

The null wide character (`\0`) at the end of the wide character string `ws1` is overwritten by the first character of the wide character string `ws2`.

`wscat()` terminates the wide character string with a null byte (`\0`).

**Return value**

Pointer            to the resulting wide character string `ws1`.

**Notes**            Wide character strings terminated with the null wide character (`\0`) are expected as arguments.

`wscat()` does not verify whether `ws1` has enough space to accommodate the result!

The behavior is undefined if memory areas overlap.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**        `wcsncat()`, `wchar.h`.

## wchr - scan wide character string for wide characters

**Syntax**      `#include <wchr.h>`

```
wchr_t *wchr(const wchr_t *ws, wint_t wc);
```

**Description** `wchr()` searches for the first occurrence of the wide character `wc` in the wide character string `ws` and returns a pointer to the located position in `ws` if successful. The value of `wc` must be a character representable as a type `wchr_t` and must be a wide-character code corresponding to a valid character in the current locale.

The terminating null wide-character code (`\0`) is considered part of the wide character string.

**Return value**

Pointer            to the position of `wc` in the wide character string `ws` if successful.

Null pointer      if `wc` is not contained in the wide character string `ws`.

**Notes**

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchr_t` (see `stddef.h`). □

**See also**      `wchr()`, `wchr.h`.

## wcscmp - compare two wide character strings

**Syntax**      `#include <wchar.h>`

```
int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
```

**Description** `wcscmp()` compares wide character strings `ws1` and `ws2` lexically, e.g.:

"circle" is lexically less than "circular";

"bustle" is lexically greater than "bus".

**Return value**

Integer value, i.e.:

< 0            `ws1` is lexically less than `ws2`.

= 0            `ws1` and `ws2` are lexically equal.

> 0            `ws1` is lexically greater than `ws2`.

**Notes**        Wide character strings terminated with the null wide character code (`\0`) are expected as arguments.

The collating sequence is based on the EBCDIC character set.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**     `wcsncmp()`, `wchar.h`.

## wscoll - compare two wide character strings according to LC\_COLLATE

**Syntax**      `#include <wchar.h>`

```
int wscoll(const wchar_t *ws1, const wchar_t *ws2);
```

**Description** `wscoll()` lexically compares two wide character strings `ws1` and `ws2`, in accordance with the collation sequence defined for the locale in `LC_COLLATE`.

**Return value**

Integer value, where the following applies:

< 0            `ws1` is less than `ws2` with regard to the defined collation sequence.  
= 0            `ws1` and `ws2` are equal with regard to the defined collation sequence.  
> 0            `ws1` is greater than `ws2` with regard to the defined collation sequence.

**Errors**      `wscoll()` will fail if:

`EINVAL`      One of the two wide character strings cannot be converted into a multi-byte string.

**Notes**        Because there is no default value defined for if an error occurs, it is advisable to set `errno` to 0, then call `wscoll()` and after the call check `errno`. If `errno` is not 0, assume that an error has occurred.

For sorting long lists, the `wcsxfrm()` and `wscmp()` functions should be used.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**      `wcncmp()`, `wcsxfrm()`, `wchar.h`.

## wcsncpy - copy wide character string

**Syntax**        `#include <wchar.h>`

```
wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2);
```

**Description** `wcsncpy()` copies the wide character string `ws2`, including the terminating null wide character code (`\0`), into the memory area pointed to by `ws1`. The space pointed to by `ws1` must be large enough to accommodate the wide character string `ws2` as well as the terminating null wide character code (`\0`).

**Return value**

Pointer            to the resulting wide character string `ws1`.

**Notes**            Wide character strings terminated with the null wide character code (`\0`) are expected as arguments.

`wcsncpy()` does not verify whether `ws1` is large enough to accommodate the result. The behavior is undefined if memory areas overlap.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**        `wcsncpy()`, `wchar.h`.

## wcscspn - get length of complementary wide character substring

**Syntax**      `#include <wchar.h>`

```
size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

**Description** Starting at the beginning of the wide character string *ws1*, `wcscspn()` calculates the length of the segment that does not contain a single character from the wide character string *ws2*. The terminating null byte (`\0`) is not treated as part of the wide character string *ws2*.

The function is terminated and the segment length is returned on encountering a character in *ws1* that matches a character in *ws2*.

If the first character in *ws1* already matches a character in *ws2*, the segment length is equal to 0.

**Return value**

Integer            that indicates the segment length (number of non-matching characters), starting at the beginning of wide character string *ws1*.

**Notes**

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**

`wcsspn()`, `wchar.h`.

## wcsftime - convert date and time to wide character string

**Syntax**      `#include <wchar.h>`

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const wchar_t *format,
                const struct tm *timptr);
```

**Description** `wcsftime()` writes wide character codes to the field pointed to by `wcs` in accordance with the string specified in `format`.

The function behaves as if a string generated by `strftime()` had been passed to `mbtowcs()` as an argument and `mbtowcs()` in turn passes the result to `wcsftime()` as a wide character string with maximum `maxsize` wide character codes.

If copying is between overlapping objects, the result is undefined.

**Return value**

Integer      which indicates the number of wide character codes written to the field (without a terminating null) if the number of wide character codes including the terminating null is less than or equal to `maxsize`.

0            otherwise. In this case the field content is undefined.

**Errors**      `wcsftime()` will fail if:

ENOMEM      There is not enough memory available for the internal management data.

**See also**    `strftime()`, `mbtowcs()`, `wchar.h`.

## wcslen - get length of wide character string

**Syntax**      `#include <wchar.h>`

```
size_t wcslen(const wchar_t *ws);
```

**Description** `wcslen()` determines the length of the wide character string `ws`, excluding the terminating null wide character code (`\0`).

**Return value**

Length of the wide character string `ws`. The terminating null wide character code (`\0`) is not included in the count.

**Notes**      A wide character string terminated with the null wide character code (`\0`) is expected as the argument.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**    `wchar.h`.



## wcsncat - concatenate two wide character strings

**Syntax**        `#include <wchar.h>`

```
wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

**Description** `wcsncat()` appends a maximum of *n* characters of the wide character string *ws2* to the end of the wide character string *ws1* and returns a pointer to *ws1*.

The terminating null wide character code (`\0`) at the end of the wide character string *ws1* is overwritten by the first byte of the wide character string *ws2*.

If the wide character string *ws2* contains less than *n* characters, only the characters in *ws2* will be appended to *ws1*, and if *ws2* contains more than *n* characters, then only the leading *n* characters of *ws2* will be appended to *ws1*.

`wcsncat()` terminates the wide character string with a null wide character code (`\0`).

**Return value**

Pointer            to the resulting wide character string *ws1*.

**Notes**            Wide character strings terminated with a null wide character code (`\0`) are expected as arguments.

`wcsncat()` does not verify whether *ws1* has enough space to accommodate the result. The behavior is undefined if memory areas overlap.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**        `wscat()`, `wchar.h`.

## wcsncmp - compare two wide character substrings

**Syntax**      `#include <wchar.h>`

```
int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

**Description** `wcsncmp()` compares the wide character strings `ws1` and `ws2` lexically up to a maximum length of `n`. For example:

```
wcsncmp("Sie", "Siemens", 3)
```

returns 0 (equal), because the first three characters of both arguments match one another.

**Return value**

Integer value:

< 0            In the first `n` characters, `ws1` is lexically less than `ws2`.

0             In the first `n` characters, `ws1` and `ws2` are lexically equal.

> 0            In the first `n` characters, `ws1` is lexically greater than `ws2`.

**Notes**        Wide character strings terminated with a null wide character code (`\0`) are expected as arguments.

The collating sequence is based on the EBCDIC character set.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**     `wscmp()`, `wchar.h`.

## wcsncpy - copy wide character substring

**Syntax**      `#include <wchar.h>`

```
wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

**Description** `wcsncpy()` copies a maximum of  $n$  characters from the wide character string `ws2` to the memory area pointed to by `ws1`.

If the wide character string `ws2` contains less than  $n$  characters, only the length of `ws2` (`wcslen + 1`) is copied, and `ws1` is then padded to the length of  $n$  with null wide character codes.

If the wide character string `ws2` contains  $n$  or more characters (excluding the null wide character code), the wide character string `ws1` is not automatically terminated with a null wide character code.

If the wide character string `ws1` contains more than  $n$  characters and the last character copied from `ws2` is not a null wide character code, any data which may still remain in `ws1` will be retained.

`wcsncpy()` does not automatically terminate `ws1` with a null wide character code.

**Return value**

Pointer            to the resulting wide character string `ws1`.

**Notes**            `wcsncpy()` does not verify whether `ws1` has enough space to accommodate the result!

Since `wcsncpy()` does not automatically terminate the resulting wide character string with a null wide character code, it may often be necessary to explicitly terminate `ws1` with a null wide character code. This is typically the case when only a part of `ws2` is being copied, and `ws2` does not contain a null wide character code either.

The behavior is undefined if memory areas overlap.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**        `wscpy()`, `wchar.h`.

## wcpbrk - get first occurrence of wide character in wide character string

**Syntax**      `#include <wchar.h>`

```
wchar_t *wcpbrk(const wchar_t *ws1, const wchar_t *ws2);
```

**Description** `wcpbrk()` searches the wide character string `ws1` for the first character that matches any character in the wide character string `ws2`. The terminating null wide character code (`\0`) is not considered part of the wide character string `ws2`.

**Return value**

Pointer            to the first matching character found in `ws1`.

Null pointer      if not a single match is present.

**Notes**          Wide character strings terminated with a null wide character code (`\0`) are expected as arguments.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**        `wcschr()`, `wcsrchr()`, `wchar.h`.

## wcsrchr - get last occurrence of wide character in wide character string

**Syntax**      `#include <wchar.h>`

```
wchar_t *wcsrchr(const wchar_t *ws, wint_t wc);
```

**Description** `wcsrchr()` searches for the last occurrence of character `wc` in the wide character string `ws` and returns a pointer to the located position in `ws` if successful.

The terminating null wide character code (`\0`) is considered to be part of the wide character string.

**Return value**

Pointer            to the position of `wc` in the wide character string `ws`.

Null pointer      if `wc` is not contained in the wide character string `ws`.

**Notes**          *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**      `wcschr()`, `wchar.h`.

## wcsrtombs - convert wide character string to multi-byte string

**Syntax**      `#include <wchar.h>`

```
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);
```

**Description** `wcsrtombs()` converts a sequence of wide characters in the array indirectly pointed to by `src` to multi-byte characters. `wcsrtombs()` starts the conversion with the conversion state described in `*ps`. The converted characters are written to the array pointed to by `dst` as long as `dst` is not a null pointer. Every character is converted as if `wctomb()` was called.

The conversion terminates when a terminating null character is encountered. The null character is also converted and written into the array.

The conversion is terminated abnormally if

- a sequence of bytes is found that does not represent a valid multi-byte character or
- `dst` is not a null pointer and the next multi-byte character would exceed the entire length `len` of characters to be written into the array.

If `dst` is not a null pointer, the pointer object pointed to by `src` is assigned one of the following two values:

- a null pointer if the conversion terminated when it reached a null character
- the address directly after the last multi-byte character converted

If `dst` is not a null pointer and the conversion terminated when it reached a null character, then the final state is the same as the “initial conversion” state.

**Return value**

`(size_t)-1` if a conversion error occurred, i.e. a sequence of bytes that does not represent a valid multi-byte character was found. The value of the `EILSEQ` macro is written in `errno`. The conversion status is undefined.

Otherwise the number of successfully converted multi-byte characters.  
The terminating null character (if present) is not counted.

**See also**      `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

## wcssp - get length of wide character substring

**Syntax**      `#include <wchar.h>`

```
size_t wcssp(const wchar_t *ws1, const wchar_t *ws2);
```

**Description** Starting at the beginning of the wide character string *ws1*, `wcssp()` computes the length of the segment that contains only characters from the wide character string *ws2*.

The function is terminated, and the segment length is returned on encountering the first character in *ws1* that does not match any character in *ws2*.

If the first character in *ws1* matches none of the characters in *ws2*, the segment length is equal to 0.

**Return value**

Integer value

that indicates the segment length (number of matching characters), starting at the beginning of string *ws1*.

**Notes** Wide character strings terminated with a null wide character code (`\0`) are expected as arguments.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also** `wcscsp()`, `wchar.h`.

## wcsstr - search for first occurrence of a wide character string

**Syntax**     `#include <wchar.h>`

```
wchar_t *wcsstr( const wchar_t *ws1, const wchar_t *ws2);
```

**Description** `wcsstr()` searches for the first occurrence of the wide character string `ws2` (not including the terminating null) in the wide character string `ws1`.

**Return value**

Pointer to the start of the string found  
if `ws2` is found in `ws1`.

Null pointer    if `ws2` is not found in `ws1`.

`ws1`            if `ws2` is a null pointer.

**Notes**        The following two function prototypes of the function `wcsstr()` are valid for C++:

```
const wchar_t* wcsstr(const wchar_t *ws1, const wchar_t *ws2);  
wchar_t* wcsstr(        wchar_t *ws1, const wchar_t *ws2);
```

**See also**     `strstr()`, `wmemcmp()`, `wmemcpy()`, `wmemchr()`



## wcstod - convert wide character string to double-precision number

Syntax `#include <wchar.h>`

```
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

Description `wcstod()` converts the initial portion of the wide character string pointed to by `nptr` to a double-precision representation. The input wide character string is first decomposed into three parts:

- an initial, possibly empty, sequence of white-space wide character codes (as specified by `iswspace()`),
- a subject sequence interpreted as a floating-point constant,
- and a final wide character string of one or more unrecognized wide character codes, including the terminating null wide character code of the input wide character string.

`wcstod()` then attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional + or - sign, then a non-empty sequence of digits optionally containing a radix, then an optional exponent part. An exponent part consists of the character `e` or `E`, followed by an optional sign, followed by one or more decimal digits. The subject sequence is defined as the longest initial subsequence of the input wide character string, starting with the first non-white-space wide character code, that is of the expected form. The subject sequence contains no wide character codes if the input wide character string is empty or consists entirely of white-space wide character codes, or if the first wide character code that is not white space is other than a sign, a digit or a radix.

If the subject sequence has the expected form, the sequence of wide character codes starting with the first digit or the radix (whichever occurs first) is interpreted as a floating constant as defined in the C language, except that the radix is used in place of a period, and that if neither an exponent part nor a radix appears, a radix is assumed to follow the last digit in the wide character string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The radix is defined in the program's locale (category `LC_NUMERIC`). In the `POSIX` locale, or in a locale where the radix is not defined, the radix defaults to a period (`.`).

In a locale other than the `POSIX` locale, other implementation-dependent subject sequence forms may be accepted. If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

**Return value**

Converted value if successful.

0 if no conversion could be performed.

HUGE\_VAL If the correct value is outside the range of representable values, (according to the sign of the value).  
errno is set to indicate the error.

**Errors**

wcstod() will fail if:

ERANGE the value to be returned would cause overflow or underflow.

**Notes**

Since 0 is returned on error and is also a valid return value on success, an application wishing to check for error situations should perform the following actions: set errno to 0, call wcstod(), then check errno, and if it is non-zero, assume that an error has occurred.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type wchar\_t (see stddef.h). □

**See also**

isspace(), localeconv(), scanf(), setlocale(), wcstol(), wchar.h.

## wcstok - split wide character string into tokens

**Syntax**      `#include <wchar.h>`

```
wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2);
```

**Description** `wcstok()` can be used to split a wide character string `ws1` into wide character substrings called "tokens", e.g. a sentence into individual words, or a source program statement into its smallest syntactical units. The pointer to `ws1` may only be passed in the first call to `wcstok()`; subsequent calls must be specified with a null pointer.

The start and end criterion for each token are separator characters (delimiters), which must be specified in a second wide character string `ws2`. Tokens may be delimited by one or more such separators or by the beginning and end of the entire wide character string `ws1`. Blanks, colons, commas, etc., are typical separators between the words of a sentence.

`wcstok()` processes exactly one token per call. The first call returns a pointer to the beginning of the first wide character token found, and each subsequent call returns a pointer to the beginning of the next such token. `wcstok()` terminates each wide character token with a null wide character code (`\0`).

A different delimiter string `ws2` may be specified in each call.

**Return value**

Pointer to the start of a wide character token.

A pointer to the first wide character token is returned at the first call; a pointer to the next wide character token at the next call, and so on. `wcstok()` terminates each wide character token in `ws1` with a null wide character code (`\0`) by overwriting the first found delimiter in each case with the null wide character code (`\0`).

Null pointer, if no wide character token, or no further wide character token was found.

**Notes**      *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**      `wchar.h`.

## wcstol - convert wide character string to long integer

Syntax `#include <wchar.h>`

```
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

Description `wcstol()` converts the initial portion of the wide character string pointed to by *nptr* to long int representation. The input wide character string is first decomposed into three parts:

- an initial, possibly empty, sequence of white-space wide-character codes (as specified by `iswspace()`),
- a subject sequence interpreted as an integer represented in some radix determined by the value of *base*,
- and a final wide character string of one or more unrecognized wide character codes, including the terminating null wide character code of the input wide character string.

`wcstol()` then attempts to convert the subject sequence to an integer, and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a + or - sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0, optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X, followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 through 15, respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a + or - sign, but not including an integer suffix. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the wide character code representations of 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide character string, starting with the first non-white-space wide character code that is of the expected form. The subject sequence contains no wide character codes if the input wide character string is empty or consists entirely of white-space wide character codes, or if the first non-white-space wide character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of wide character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

#### Return value

Converted value

if successful.

0

if no conversion could be performed.

LONG\_MAX, LONG\_MIN

if the correct value is outside the range of representable values (according to the sign of the value).

*errno* is set to indicate the error.

#### Errors

`wcstol()` will fail if:

EINVAL

The value of *base* is not supported.

ERANGE

The value to be returned is not representable.

#### Notes

Since 0, LONG\_MIN and LONG\_MAX are returned on error and are also valid return values on success, an application wishing to check for error situations should perform the following actions: set *errno* to 0, call `wcstol()`, then check *errno*, and if it is 0, assume that an error has occurred.

#### Restriction

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

#### See also

`iswalpha()`, `scanf()`, `wcstod()`, `wchar.h`.

## wcstoll - convert wide character string to long long integer

Syntax `#include <wchar.h>`

```
long long int wcstoll(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
```

Description `wcstoll()` converts the initial portion of the wide character string pointed to by *nptr* to long long int representation. The input wide character string is first decomposed into three parts:

- an initial, possibly empty, sequence of white-space wide-character codes (as specified by `iswspace()`),
- a subject sequence interpreted as an integer represented in some radix determined by the value of *base*,
- and a final wide character string of one or more unrecognized wide character codes, including the terminating null wide character code of the input wide character string.

`wcstoll()` then attempts to convert the subject sequence to an integer, and returns the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a + or - sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0, optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X, followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 through 15, respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a + or - sign, but not including an integer suffix. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the wide character code representations of 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide character string, starting with the first non-white-space wide character code that is of the expected form. The subject sequence contains no wide character codes if the input wide character string is empty or consists entirely of white-space wide character codes, or if the first non-white-space wide character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of wide character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

#### Return value

Converted value

if successful.

0

if no conversion could be performed.

`errno` is set to `EINVAL` if the value of *base* is not supported.

`LLONG_MAX`, `LLONG_MIN`

depending on the sign of the value.

`ULLONG_MAX`

if the correct value is outside the range of representable values.

`errno` is set to `ERANGE` to indicate an error

#### Errors

Since 0 is returned on error as well as when a valid return can be successfully represented, an application wishing to check for error situations should perform the following actions: set `errno` to 0, call `wcstoll()`, then check `errno`, and if it is not equal to 0, assume that an error has occurred.

#### Notes

This version of the C runtime system only supports 1-byte characters as wide character codes.

#### See also

`iswalph()`, `iswspace()`, `scanf()`, `strtol()`, `strtoll()`, `strtoul()`, `strtoull()`, `wcstod()`, `wcstol()`, `wcstoul()`

## wcstombs - convert wide character string to character string

**Syntax**      `#include <stdlib.h>`

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

**Description** `wcstombs()` converts a sequence of `wchar_t` values located in `pwcs` to the appropriate multi-byte characters and stores them in string `s`.  
`n` specifies the maximum number of bytes to be stored in `s`.

No characters consisting of multiple bytes are implemented in this version. Multi-byte characters always have a length of 1 byte, and `wchar_t` values are always of type `long`. `wcstombs()` assigns each `wchar_t` value (of type `long`) in `pwcs` to an area of 1-byte length in string `s`.

The assignment terminates:

- on encountering the `wchar_t` value 0 in `pwcs`,
- when `n` bytes have been assigned or
- on encountering a `wchar_t` value that cannot be represented in 1 byte.

**Return value**

Number of assigned bytes  
upon successful conversion.

`(size_t)-1`  
if a `wchar_t` value cannot be converted to a multi-byte character.

**Notes** If a `wchar_t` value in `pwcs` cannot be converted to a multi-byte character, the `wchar_t` values already converted will be stored in `s`.

The behavior is undefined if memory areas overlap.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also** `mblen()`, `mbtowc()`, `mbstowcs()`, `wctomb()`, `stdlib.h`.



## wcstoul - convert wide character string to unsigned long

**Syntax**      `#include <wchar.h>`  
`unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);`

**Description** `wcstoul()` converts the initial portion of the wide character string pointed to by `nptr` to `unsigned long int` representation. The input wide character string is first decomposed into three parts:

- an initial, possibly empty, sequence of white-space wide character codes (as specified by `iswspace()`),
- a subject sequence interpreted as an integer represented in some radix determined by the value of `base`,
- and a final wide-character string of one or more unrecognized wide character codes, including the terminating null wide-character code of the input wide character string.

`wcstoul()` then attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of `base` is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a + or - sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 through 15, respectively.

If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a + or - sign, but not including an integer suffix. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the wide character codes 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide character string, starting with the first wide character code that is not white space and is of the expected form. The subject sequence contains no wide character codes if the input wide character string is empty or consists entirely of white-space wide character codes, or if the first wide character code that is not white space is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of wide character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

#### Return value

Converted value if successful.

0 if no conversion could be performed.

ULONG\_MAX if the correct value is outside the range of representable values (according to the sign of the value). *errno* is set to indicate the error.

#### Errors

`wcstoul()` will fail if:

EINVAL The value of *base* is not supported.

ERANGE The value to be returned is not representable.

#### Notes

Since 0 and ULONG\_MAX are returned on error and 0 is also a valid return value on success, an application wishing to check for error situations should perform the following actions: set *errno* to 0, call `wcstoul()`, then check *errno*, and if it is non-zero, assume that an error has occurred. Unlike `wcstod()` and `wcstol()`, `wcstoul()` must always return a non-negative number, so using the return value of `wcstoul()` for out-of-range numbers with `wcstoul()` could cause more severe problems than just loss of precision if those numbers can ever be negative.

#### Restriction

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

#### See also

`iswalph()`, `scanf()`, `wcstod()`, `wcstol()`, `wchar.h`.



If the subject sequence has the expected form and the value of *base* is 0, the sequence of wide character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

#### Return value

Converted value

if successful.

0

if no conversion could be performed.

`errno` is set to `EINVAL` if the value of *base* is not supported.

`LLONG_MAX`, `LLONG_MIN`

depending on the sign of the value.

`ULLONG_MAX`

if the correct value is outside the range of representable values.

`errno` is set to `ERANGE` to indicate an error

#### Errors

Since 0 is returned on error as well as when a valid return can be successfully represented, an application wishing to check for error situations should perform the following actions: set `errno` to 0, call `wcstoull()`, then check `errno`, and if it is not equal to 0, assume that an error has occurred.

#### Notes

This version of the C runtime system only supports 1-byte characters as wide character codes.

#### See also

`iswalpha()`, `iswspace()`, `scanf()`, `strtoul()`, `wcstod()`, `wcstol()`

## wcs wcs - find wide character substring in wide character string

**Syntax**      `#include <wchar.h>`

```
wchar_t *wcs wcs(const wchar_t *ws1, const wchar_t *ws2);
```

**Description** `wcs wcs()` locates the first occurrence of the wide character string `ws2` (excluding the terminating null wide character code) in the wide character string `ws1`.

**Return value**

Pointer            to the start of the wide character string found in `ws1`.

Null pointer      if `ws2` is not contained in `ws1`.

Pointer            to the start of `ws1` if `ws2` has a length of 0.

**Notes**            Wide character strings terminated with a null wide character code (`\0`) are expected as arguments.

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**        `wcschr()`, `wchar.h`.

## wcswidth - get number of column positions of wide character string

**Syntax**      `#include <wchar.h>`  
`int wcswidth(const wchar_t *pwcs, size_t n);`

**Description** `wcswidth()` determines the number of column positions required for  $n$  characters in the string pointed to by `pwcs`. If a null wide character code is encountered before  $n$  characters are exhausted, fewer than  $n$  characters are processed.

**Return value**  
Number of column positions for the wide character string `pwcs`.

- 0                    if `pwcs` points to a null wide character code.
- 1                   if `pwcs` contains a non-printing wide character code.

**Notes**          *Restriction*  
This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**      `wchar.h`.

## wcsxfrm - transform wide character string

**Syntax**      `#include <wchar.h>`

```
size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

**Description** `wcsxfrm()` transforms the wide character string pointed to by `ws2`, and writes the result of the transformation to the field pointed to by `ws1`. The transformation is performed such that the `wscmp()` function returns the same return value (greater than, equal to or less than zero) for two transformed wide character strings as the `wscoll()` function does for the two original non-transformed wide character strings.

A maximum of `n` wide character codes are written to the field (including the terminating null character).

If `n` is 0, `ws1` can be a null pointer.

If copying is between overlapping objects, the result is undefined.

**Return value**

Integer value  $< n$

indicating the number of wide character codes written to the field (without terminating null).

Integer value  $\geq n$

in this case the content of the `ws1` field is undefined.

$(\text{size\_t}) - 1$

if an error occurs. `errno` is set to indicate the error.

**Errors**

`wcsxfrm()` will fail if:

`EINVAL`

The wide character string pointed to by `ws2` contains wide character codes from outside the value range of the selected collation sequence.

`ENOMEM`

There is not enough memory available for the internal management data.

**Notes**

Transformation is such that two transformed wide character strings are arranged by `wscmp()` in accordance with the collation sequence defined in `LC_COLLATE`.

The fact that `ws1` can be a null pointer if `n` is 0, is useful if the size of the field is to be determined before the transformation. Because there is no default value defined for if an error occurs, it is advisable to set `errno` to 0, then call `wscoll()` and after the call check `errno`. If `errno` is not 0, assume that an error has occurred.

### *Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**      `wscmp()`, `wscoll()`, `wchar.h`.

## wctob - convert wide character to 1-byte multi-byte character

Syntax      `#include <stdio.h>`  
             `#include <wchar.h>`  
  
             `int wctob(wint_t c);`

Description `wctob()` tests if the character `c` corresponds to an element of the extended character set whose multi-byte representation consists of one byte in the “initial shift” state.

Return value  
EOF            if no corresponding multi-byte character of length one exists in the “initial shift” state for `c`.  
  
Otherwise the multi-byte character of length one that corresponds to `c`.

See also      `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

## wctomb - convert wide character code to character

Syntax      `#include <stdlib.h>`  
  
             `int wctomb(char *s, wchar_t wchar);`

Description `wctomb()` converts the `wchar_t` value `wchar` to the appropriate multi-byte character and stores it in string `s`.  
No characters consisting of multiple bytes are implemented in this version. Multi-byte characters always have a length of 1 byte, and `wchar_t` values are always of type `long`. `wctomb()` assigns the `wchar_t` value (of type `long`) to the area `s`, of 1-byte length. No assignment occurs if `s` is a null pointer or if the `wchar_t` value cannot be represented in 1 byte.

Return value  
0              if `s` is a null pointer.  
-1             if the `wchar_t` value cannot be converted to a multi-byte character.  
1              in all other cases.

Notes        *Restriction*  
This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

See also      `mblen()`, `mbstowcs()`, `mbtowc()`, `wcstombs()`, `stdlib.h`.



## wctrans - define wide character mappings

**Syntax**      `#include <wctype.h>`

```
wctrans_t wctrans(const char *property);
```

**Description** `wctrans()` constructs a value of type `wctrans_t` from *property* that describes a mapping between wide characters.

The two strings "tolower" and "toupper" are permitted in all locales as a value of the *property* argument.

If *property* identifies a mapping that is valid according to the `LC_CTYPE` category of the current locale, `wctrans()` returns a value not equal to 0 that can be used as a valid second argument in the function `towctrans()`.

**Return value**

Value  $\neq$  0      if *property* identifies a valid mapping.

0                  otherwise.

**Notes**        This version of the C runtime system only supports 1-byte characters as wide character codes.

**See also**     `towctrans()`

## wctype - define wide character class

Syntax `#include <wchar.h>`

```
wctype_t wctype(const char *charclass);
```

Description `wctype()` is defined for valid character class names as defined in the current locale. The *charclass* is a string identifying a generic character class for which codeset-specific type information is required. The following character class names are defined in all locales: "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit".

Additional character class names defined in the locale definition file (category LC\_CTYPE) can also be specified.

The function returns a value of type `wctype_t`, which can be used as the second argument to subsequent calls of `iswctype()`. The `wctype()` function determines values of `wctype_t` according to the rules of the coded character set defined by character type information in the program's locale (category LC\_CTYPE). The values returned by `wctype()` are valid until a call to `setlocale()` that modifies the category LC\_CTYPE.

### Restriction

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

Return value

- 0 if the character class name is not valid for the current locale (category LC\_CTYPE).
- ≠ 0 An object of type `wctype_t` that can be used in calls to `iswctype()` is returned.

See also `iswctype()`, `wchar.h`.

## wcwidth - get number of column positions of wide character code

**Syntax**      `#include <wchar.h>`  
`int wcwidth(wint_t wc);`

**Description** `wcwidth()` determines the number of column positions required for the wide character `wc`. The value of `wc` must be a character representable as a `wchar_t`, and must be a wide character code corresponding to a valid character in the current locale.

**Return value**

|    |                                                                                |
|----|--------------------------------------------------------------------------------|
| -1 | if <code>wc</code> does not correspond to a representable wide character code. |
| 0  | if <code>wc</code> is a null wide-character code.                              |
| 1  | if <code>wc</code> corresponds to a representable wide character code.         |

### Notes

*Restriction*

This version of the C runtime system only supports 1-byte characters as wide character codes. They are of type `wchar_t` (see `stddef.h`). □

**See also**      `wchar.h`.

## wmemchr - search for wide character in a wide character string

**Syntax**      `#include <wchar.h>`

```
wchar_t *wmemchr( const wchar_t *ws, wchar_t *wc, size_t n);
```

**Description** `wmemchr()` searches for the first occurrence of the wide character `wc` in the first `n` bytes of the wide character string `ws` and returns a pointer to the desired position in `ws` if successful.

**Return value**

Pointer to the position of `wc` in `ws`  
if successful,

Null pointer      otherwise.

**Notes**      This version of the C runtime system only supports 1-byte characters as wide character codes.

The following two prototypes are valid in C++ for the function `wmemchr()`:

```
const wchar_t* wmemchr(const wchar_t *ws, wchar_t *wc, size_t n);  
wchar_t* wmemchr(      wchar_t *ws, wchar_t *wc, size_t n);
```

**See also**      `memchr()`, `wcsstr()`, `wmemcmp()`, `wmemcpy()`

## wmemcmp - compare two wide character strings

**Syntax**      `#include <wchar.h>`

```
int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

**Description** `wmemcmp()` compares the first *n* bytes of the two wide character strings *ws1* and *ws2* lexicographically.

**Return value**

`< 0`            *ws1* is lexicographically smaller than *ws2*.

`= 0`            *ws1* and *ws2* are lexicographically equal.

`> 0`            *ws1* is lexicographically larger than *ws2*.

**Notes**        This version of the C runtime system only supports 1-byte characters as wide character codes.

**See also**     `memcmp()`, `wcsstr()`, `wmemchr()`, `wmemcpy()`.

## wmemcpy - copy wide character string

**Syntax**      `#include <wchar.h>`

```
wchar_t *wmemcpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

**Description** `wmemcpy()` copies the first *n* bytes of the wide character string *ws2* to the first *n* bytes of the wide character string *ws1*.

**Return value**

Pointer to the wide character string *ws1*.

**Notes**        This version of the C runtime system only supports 1-byte characters as wide character codes.

**See also**     `memcpy()`, `wmemmove()`, `wmemset()`.

## wmemmove - copy wide character string in overlapping area

**Syntax**      `#include <wchar.h>`

```
wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

**Description** `wmemmove()` copies the first  $n$  bytes of the wide character string `ws2` to the first  $n$  bytes of the wide character string `ws1`. The copy is performed as if the  $n$  wide characters are first copied to a temporary array that does not overlap with `ws1` or `ws2`, and are then copied from this array to `ws1`.

**Return value**

Pointer to the wide character string `ws1`.

**Notes**        This version of the C runtime system only supports 1-byte characters as wide character codes.

**See also**    `memmove()`, `wmemcpy()`, `wmemset()`

## wmemset - set first $n$ wide characters in wide character string

**Syntax**      `#include <wchar.h>`

```
wchar_t *wmemset(wchar_t *ws, wchar_t *c, size_t n);
```

**Description** `wmemset()` sets the first  $n$  wide characters in the wide character string `ws` to the value `c`.

**Return value**

Pointer to `ws`.

**Notes**        This version of the C runtime system only supports 1-byte characters as wide character codes.

**See also**    `memset()`, `wmemcpy()`, `wmemmove()`

## wprintf - formatted output of wide characters

Syntax        `#include <wchar.h>`  
              `int wprintf(const wchar_t *format [, arglist]);`

Description A detailed description can be found under `fwprintf()`.

## write - write bytes to file

Syntax `#include <unistd.h>`

*BS2000*

`#include <stdio.h>` □

`ssize_t write(int fildes, const void *buf, size_t nbyte);`

Description `write()` attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the file descriptor *fildes*.

*BS2000*

SAM files are always processed as text files with elementary functions. □

On a file that is capable of seeking, the actual write operation proceeds from the position in the file indicated by the file offset (i.e. the file position indicator) associated with *fildes*.

Before a successful return from `write()`, the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the `O_SYNC` flag of the file status flags is set and *fildes* refers to a regular file, a successful `write()` does not return until the data is delivered to the underlying hardware.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the beginning of the `write()` operation.

If a `write()` requests that more bytes be written than the amount of available space (because of the `ulimit()` or the physical end of a medium, for instance), only as many bytes as can be accommodated will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20 in this case, and the next write with a non-zero number of bytes will return with an error (except in the cases noted below) and will sent the `SIGXFSZ` signal to the process.

If `write()` is interrupted by a signal before it has written the data, -1 is returned and `errno` is set to `EINTR`.

If `write()` is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.



The following applies following a successful `write()` to a regular file:

- Any successful `read()` from each byte position in the file that was modified by that write will return the data specified by the `write()` for that position until such byte positions are again modified.
- Any subsequent successful `write()` to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO will be handled the same as a regular file, with the following exceptions:

- There is no file offset associated with a pipe, so each write request will append to the end of the pipe.
- Write requests of `{PIPE_BUF}` bytes or less bytes will not be interleaved with data from other processes doing writes on the same file. Writes of greater than `{PIPE_BUF}` bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the `O_NONBLOCK` flag in the system file status byte is set.
- If the `O_NONBLOCK` flag is clear, a write request may cause the process to block, but on normal completion it will return *nbyte*.
- If the `O_NONBLOCK` flag is set, `write()` requests will be handled differently, in the following ways:
  - `write()` will not block the process.
  - A write request for `{PIPE_BUF}` or fewer bytes will have the following effects:
    - a) If there is sufficient space available in the pipe, `write()` will transfer all the data and return the number of bytes requested.
    - b) If there is not enough space available in the pipe, `write()` will transfer no data and return -1 with `errno` set to `EAGAIN`.
  - A write request for more than `{PIPE_BUF}` bytes will cause one of the following:
    - a) When at least one byte can be written, `write()` will transfer as many bytes as it can and return the number of bytes written. When all data previously written to the pipe is read, it will transfer at least `{PIPE_BUF}` bytes.
    - b) When no data can be written, `write()` will transfer no data and return -1 with `errno` set to `EAGAIN`.

If a request is for more than `{PIPE_BUF}` bytes and all data previously written to the file has been read, `write()` will transfer at least `{PIPE_BUF}` bytes.

The following occurs when attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-blocking writes:

- If the `O_NONBLOCK` flag is clear, `write()` will block until the data can be accepted.
- If the `O_NONBLOCK` flag is set, `write()` will not block the process. If some data can be written without blocking the process, `write()` will write as many bytes as it can and return the number of bytes written. Otherwise, it will return `-1` and `errno` will be set to `EAGAIN`.

Upon successful completion, where `nbyte` is greater than 0, `write()` will mark for update the `st_ctime` and `st_mtime` structure components of the file. The `S_ISUID` and `S_ISGID` bits of the file mode will be cleared if the process does not have appropriate privileges.

If `files` describes a STREAM, the write operation is determined by the minimum and maximum values for `nbyte` („packet size“) accepted by the STREAM. These values are defined by the highest level STREAM module.

If `nbyte` bytes is the permitted packet size, `nbyte` bytes are written.

If `nbyte` is in the permitted range for the packet size and the smallest packet size is equal to 0, `write()` divides the buffer up into segments of a size equal to the maximum packet size before the data is sent upstream (the last segment can be smaller).

If `nbyte` is not in the permitted range for the packet size and the smallest packet size is not equal to 0, `write()` fails and sets `errno` to `ERANGE`.

If a buffer of length 0 (`nbyte = 0`) is written to a STREAM, `write()` sends a message of length 0 and returns the value 0. However, if a buffer of length 0 is written to a STREAM-based pipe or a FIFO file, nothing is sent and 0 is returned. The process can use `I_SWROPT ioctl()` if messages of length 0 are to be sent through the pipe or FIFO file.

If `write()` writes to a STREAM, messages with the priority class 0 are generated.

The following rules apply if `write()` writes to a STREAM that is not a pipe or a FIFO file:

- If the `O_NONBLOCK` flag is clear and the STREAM does not accept any data (because the STREAM write queue is full due to internal control flow conditions), `write()` blocks until the data is accepted.
- If the `O_NONBLOCK` flag is set and the STREAM does not accept any data, `write()` fails, returns `-1` and sets `errno` to `EAGAIN`.
- If the `O_NONBLOCK` flag is set and `write()` has already written a portion of the buffer when a condition arises in which the STREAM does not accept any more data, `write()` terminates and returns the number of bytes actually written.

If threads are used, the function affects the process or a thread in the following manner:

– Write bytes to file

A write request for a pipe or FIFO is handled just like such a request for a normal file with the following exceptions:

- If the `O_NONBLOCK` flag is clear, a write request can block the thread, but returns the result *nbyte* if it terminates normally.
- If the `O_NONBLOCK` flag is set, the request from `write()` is handled differently: `write()` does not block the thread.

If an attempt is made to write to a file descriptor that is not a pipe or FIFO and supports non-blocking writes, the following occurs:

- If the `O_NONBLOCK` flag is clear, `write()` blocks the calling thread until the data is accepted.
  - `EAGAIN` - the `O_NONBLOCK` flag is set for the file descriptor and the thread would be stopped by the write operation.
- Furthermore, if an `EPIPE` error occurs, then `SIGPIPE` signal is not sent to the process, but to the calling thread instead.

### Return value

Number of bytes actually written

upon successful completion. This number will never be greater than *nbyte*.

- 0 if data was to be written to a regular file and *nbyte* is equal to 0. No data will be written.
- 1 if an error occurs. `write()` will not have written any data due to one of the following errors:
- A physical I/O error occurred.
  - *fdes* is not a valid file descriptor.
  - The file does not exist.
  - No write permission exists for the file.
  - The area containing the data was not correctly specified.
- `errno` is set to indicate the error.

|                  |                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Errors           | <code>write()</code> fails if the following applies:                                                                                                                                                                                                                                                                                                                             |
| EAGAIN           | The <code>O_NONBLOCK</code> flag is set for the file descriptor and the process would be delayed in the <code>write()</code> operation.                                                                                                                                                                                                                                          |
| EBADF            | <i>fd</i> is not a valid file descriptor open for writing.                                                                                                                                                                                                                                                                                                                       |
| EFBIG            | An attempt was made to write a file that exceeds the maximum possible file size or the process file size limit (see <code>getrlimit()</code> and <code>ulimit()</code> ).                                                                                                                                                                                                        |
| <i>Extension</i> |                                                                                                                                                                                                                                                                                                                                                                                  |
| EAGAIN           | The amount of system memory available for raw I/O is temporarily insufficient, or<br>an attempt was made to write to a stream that cannot accept data with the <code>O_NDELAY</code> or <code>O_NONBLOCK</code> flag set, or<br>an attempt was made to write <code>{PIPE_BUF}</code> or fewer bytes to a pipe or FIFO and less than <i>nbytes</i> of free space was available. □ |
| <i>Extension</i> |                                                                                                                                                                                                                                                                                                                                                                                  |
| EDEADLK          | The <code>write()</code> function is sleeping and causes a deadlock situation to occur.                                                                                                                                                                                                                                                                                          |
| EFAULT           | <i>buf</i> points outside the allocated address space of the process. □                                                                                                                                                                                                                                                                                                          |
| EINTR            | The write operation was terminated by a signal, and no data was transferred.                                                                                                                                                                                                                                                                                                     |
| <i>Extension</i> |                                                                                                                                                                                                                                                                                                                                                                                  |
| EINVAL           | An attempt was made to write to a stream associated with a multiplexer. □                                                                                                                                                                                                                                                                                                        |
| EIO              | A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the <code>SIGTTIN</code> signal or the process group of the process is orphaned.                                                                                                      |
| ENOSPC           | There was no free space remaining on the device containing the file.                                                                                                                                                                                                                                                                                                             |
| <i>Extension</i> |                                                                                                                                                                                                                                                                                                                                                                                  |
| ENOSR            | An attempt was made to write to a stream for which not enough space is available. □                                                                                                                                                                                                                                                                                              |
| ENXIO            | A request was made of a non-existent device, or the request was outside the capabilities of the device.                                                                                                                                                                                                                                                                          |
| EPIPE            | An attempt was made to access a non-existent device, or the request was outside the capabilities of the device.<br>The process gets a <code>SIGPIPE</code> signal.                                                                                                                                                                                                               |
| ERANGE           | An attempt was made to write to a stream with an <i>mbyte</i> value outside the prescribed minimum and maximum limits, and the minimum value is non-zero.                                                                                                                                                                                                                        |

|        |                                                                                                                         |
|--------|-------------------------------------------------------------------------------------------------------------------------|
| EINVAL | The stream or multiplexer referred to by <i>files</i> is directly or indirectly connected via a multiplexer downstream. |
| ENXIO  | An attempt was made to access a non-existent device, or the device was not capable of the request.                      |
| ENXIO  | A hang-up occurred during writing to the stream.                                                                        |

`write()` will also fail if an asynchronous error message appears at the STREAM head before the call. In this case, the value of `errno` does not refer to `write()` but to the previous STREAM error.

**Notes** The `sizeof()` function should be used to ensure that the value specified in *nbyte* does not exceed the size of the buffer.

#### *BS2000*

The number of bytes actually written should be verified after each call to `write()`:

- If the result is less than the value specified in *nbyte*, it generally means that an error has occurred.
- If the result is greater than the *nbyte* specification, tab characters (`\t`) were written to a text file; these tab characters were expanded to the appropriate spaces and included in the number of bytes returned.

The bytes are not written immediately to the external file but are stored in an internal C buffer (see section "Buffering streams" on page 77).

Control characters for white space (`\n`, `\t`, etc.) are converted to their appropriate effect when output to text files, depending on the type of text file (see section "White-space characters" on page 84). □

**See also** `creat()`, `dup()`, `fcntl()`, `lseek()`, `open()`, `pipe()`, `ulimit()`, `unistd.h`.

## writev - write to file

**Syntax**      `#include <sys/uio.h>`

```
ssize_t writev(int fildev, const struct iovec *iov, size_t nbyte);
```

**Description** `writev()` does the same as `write()`, but collects the output data of the *iovcnt* buffers that are defined by the members of the *iov* fields (*iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1]). The following must apply:  $0 < \textit{iovcnt} \leq \text{IOV\_MAX}$ .

For `writev()` the `iovec` structure contains the following elements:

```
    caddr_t   iov_base;
    int       iov_len;
```

Each `iovec` entry specifies the basic address and the length of the memory area from which the data is to be written. `writev()` always fills a whole area before proceeding to the next one.

If *fildev* identifies a regular file and all elements of the *iov* field have the value 0, `writev()` returns the value 0 and has no other effect.

If the sum of the *iov\_len* values exceeds `SSIZE_MAX`, `writev()` fails and no data is transferred.

For more details, see `write()`.

**Return value**

Number of bytes actually written if successful.

-1 otherwise. In this case the file pointer is not changed. `errno` is set to indicate the error.

**Errors** see `write()`. In addition to the errors specified there, `writev()` will fail if:

`EINVAL`      *iovcnt* was less than or equal to 0 or greater than or equal to 16, or one of the *iov\_len* values in the *iov* field was negative, or the sum of the *iov\_len* values in the *iov* field creates an overflow in the case of a 32-bit integer.

`EINVAL`      *fildev* is assigned to a BS2000 file.

`writev()` will also fail if an asynchronous error message appears at the STREAM head before the call. In this case, the value of `errno` does not refer to `writev()`, but to the previous STREAM error.

**See also** `chmod()`, `creat()`, `dup()`, `fcntl()`, `getrlimit()`, `lseek()`, `open()`, `pipe()`, `ulimit()`, `limits.h`, `stropts.h`, `sys/uio.h`, `unistd.h`.

## wscanf - formatted read

Syntax `#include <wchar.h>`  
`int wscanf(const wchar_t *format [, arglist]);`

Description A detailed description can be found under `fwscanf()`.

## y0, y1, yn - Bessel functions of the second kind

Syntax `#include <math.h>`  
`double y0(double x);`  
`double y1(double x);`  
`double yn(int n, double x);`

Description `y0()`, `y1()` and `yn()` compute the Bessel functions of the second kind for real arguments  $x (> 0)$  and the integral orders 0, 1 or  $n$  (only for `yn`).

Return value  
Value of the Bessel function of  $x$ , if  $x > 0$ .  
`-HUGE_VAL` for arguments  $\leq 0$ .  
`errno` is set to indicate the error.

Errors `y0()`, `y1()` and `yn()` will fail if:  
`EDOM` The value of  $x$  is negative.

See also `j0()`, `j1()`, `jn()`, `math.h`.





---

# Header files

This chapter describes the contents of the header files in which X/Open-conformant functions, macros, and external variables are defined or declared. The headers are presented in alphabetical order.

Header files also contain the definitions of symbolic constants, common structures, preprocessor macros and predefined data types. In order to use a function, specific header files must be included in an application; these header files are specified with each function and variable description in the chapter "Functions and variables in alphabetical order (a - m)". Headers are only required during the development process; they need not be present when applications are executed.

## Format of entries

Each description begins with a title, which contains the name of the header and some keywords to describe its functionality. The title is always followed by a uniform group of subsections:

- |             |                                                                                                                                                                                    |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Syntax      | Shows the syntax with which each header file is included in an application.                                                                                                        |
| Description | Describes the contents of the header file.                                                                                                                                         |
| Notes       | Typically includes explanations of concepts, information on interaction with other functions, and/or tips concerning application usage. This section may be omitted in some cases. |
| See also    | Contains cross-references to function descriptions, related sections in this manual and other relevant manuals.                                                                    |

Extensions and deviations with respect to the Standard are indicated by the following markers:

### *BS2000*

Provides information on extensions of the C runtime system describing functionality in connection with access to DMS and C runtime versions up to V2.1C (CRTE V1.0B). The end of each such section is indicated by the end marker. □

### *Extension*

Includes information on extensions of the C runtime system V2.2 (CRTE V 2.0). The end of each such section is likewise indicated by the end marker. □

## assert.h - verify program assertion

**Syntax**      `#include <assert.h>`

**Description** The `assert.h` header defines the `assert()` macro. It refers to the macro `NDEBUG` which is not defined in the header. If `NDEBUG` is defined as a macro name before the inclusion of this header, the `assert()` macro is defined simply as:

```
#define assert(ignore) ((void) 0)
```

Otherwise, the macro behaves as described in `assert()`.

The `assert()` macro is implemented as a macro, not as a function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

**See also**      `assert()`.

## cpio.h - cpio archive mode values

Syntax `#include <cpio.h>`

Description Values needed by the `c_mode` field of the `cpio` archive format. The values are described as follows:

| Name                  | Value (octal) | Description                   |
|-----------------------|---------------|-------------------------------|
| <code>C_IRUSR</code>  | 0000400       | Read permission for user      |
| <code>C_IWUSR</code>  | 0000200       | Write permission for user     |
| <code>C_IXUSR</code>  | 0000100       | Execute permission for user   |
| <code>C_IRGRP</code>  | 0000040       | Read permission for group     |
| <code>C_IWGRP</code>  | 0000020       | Write permission for group    |
| <code>C_IXGRP</code>  | 0000010       | Execute permission for group  |
| <code>C_IROTH</code>  | 0000004       | Read permission for others    |
| <code>C_IWOTH</code>  | 0000002       | Write permission for others   |
| <code>C_IXOTH</code>  | 0000001       | Execute permission for others |
| <code>C_ISUID</code>  | 0004000       | Set user ID                   |
| <code>C_ISGID</code>  | 0002000       | Set group ID                  |
| <code>C_ISVTX</code>  | 0001000       | Reserved                      |
| <code>C_ISDIR</code>  | 0040000       | Directory                     |
| <code>C_ISFIFO</code> | 0010000       | FIFO file                     |
| <code>C_ISREG</code>  | 0100000       | Regular file                  |
| <code>C_ISBLK</code>  | 0060000       | Block special file            |
| <code>C_ISCHR</code>  | 0020000       | Character special file        |
| <code>C_ISCTG</code>  | 0110000       | Reserved                      |
| <code>C_ISLNK</code>  | 0120000       | Reserved                      |
| <code>C_ISSOCK</code> | 0140000       | Reserved                      |

The header file defines the symbolic constant:

```
MAGIC "070707"
```

## ctype.h - character classification

**Syntax**      `#include <ctype.h>`

**Description** The following names are declared as functions and, with the exception of `isprint()`, `tolower()` and `toupper()`, also defined as macros:

```
int isalnum(int c);
int isalpha(int c);
int isascii(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int toascii(int c);
int tolower(int c);
int toupper(int c);
```

The following macro names are defined:

```
int _toupper(int c);
int _tolower(int c);
```

**See also** `isalnum()`, `isalpha()`, `isascii()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `toascii()`, `tolower()`, `toupper()`, `locale.h`.

## dirent.h - format of directory entries

**Syntax**      `#include <dirent.h>`

**Description** The following data type is defined through `typedef`:

`DIR`                      A type representing a directory stream.

The data types `ino_t` and `ino64_t` are defined as described in `sys/types.h`.

The structure `dirent` is defined with the following members:

`ino_t d_ino`              file serial number

`char d_name[]`          name of entry

The structure `dirent64` is defined with the following components:

`ino64_t d_ino`          sequential file number

`char d_name[]`          name of the entry

The `char` array `d_name` is of unspecified size, but the number of bytes preceding the terminating null byte should not exceed `{NAME_MAX}` (described in `limits.h`).

The following functions are declared:

```
int closedir(DIR *dirp)
```

```
void rewinddir(DIR *dirp)
```

```
DIR *opendir(const char *dirname)
```

```
void seekdir(DIR *dirp, long int *loc)
```

```
struct dirent *readdir(DIR *dirp)
```

```
struct dirent64 *readdir64(DIR *dirp);
```

```
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result)
```

```
long int telldir(DIR *dirp)
```

**See also**      `closedir()`, `opendir()`, `readdir()`, `readdir_r( )`, `rewinddir()`, `seekdir()`, `telldir()`, `sys/types.h`.

## errno.h - system error codes

Syntax `#include <errno.h>`

Description The `errno.h` header provides, among other things, a declaration for the variable `errno` and defines unique non-zero values for the following symbolic constants.

The general descriptions in the table below are more precisely defined in the "Errors" section of each respective function where these errors may occur. Only the symbolic names should be used in programs, since the actual values of the error codes are implementation-dependent and hence not portable.

Error codes that are defined in `errno.h` but not prescribed by the XPG4 Version 2 standard are marked as extensions.

| Symbolic error code                   | Error message                                   | Meaning                                                                                                                                                                                                                                                                |
|---------------------------------------|-------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| E2BIG                                 | Argument list too long                          | The sum of the number of bytes used by the argument list and environment list of the new process image is greater than the system-imposed limit of <code>{ARG_MAX}</code> bytes.                                                                                       |
| EACCES                                | Permission denied                               | An attempt was made to access a file by a method that is forbidden by its file access permissions.                                                                                                                                                                     |
| EADDRINUSE<br>( <i>Extension</i> )    | Address in use                                  | An attempt was made to use an address that is already in use, and the protocol does not allow this.                                                                                                                                                                    |
| EADDRNOTAVAIL<br>( <i>Extension</i> ) | Address not available                           | An attempt was made to create a transport endpoint with an address not on the current machine.                                                                                                                                                                         |
| EADV<br>( <i>Extension</i> )          | Advertise error                                 | This error relates to the use of RFS (remote file sharing). It occurs when an attempt is made to advertise a resource that has already been advertised, or to stop RFS when there are resources still advertised, or to unmount a device while it is still advertised. |
| EAFNOSUPPORT<br>( <i>Extension</i> )  | Address family not supported by protocol family | An address incompatible with the requested protocol was used.                                                                                                                                                                                                          |
| EAGAIN                                | Resource temporarily unavailable                | This is a temporary error condition. Later calls to the same function may complete normally. For example, it is possible that a system call failed because of insufficient memory.                                                                                     |

| <b>Symbolic error code</b>       | <b>Error message</b>          | <b>Meaning</b>                                                                                                                                                                                                                                                                             |
|----------------------------------|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EALREADY<br>( <i>Extension</i> ) | Operation already in progress | An operation was attempted on a non-blocking object that already had an operation in progress.                                                                                                                                                                                             |
| EBADE<br>( <i>Extension</i> )    | Bad exchange                  |                                                                                                                                                                                                                                                                                            |
| EBADF                            | Bad file descriptor           | A file descriptor argument is out of the permissible range or does not refer to an open file, or a read (write) request is made to a file that is only open for writing (reading).                                                                                                         |
| EBADFD<br>( <i>Extension</i> )   | File descriptor in bad state  | Either a file descriptor refers to no open file or a read request was made to a file that is open only for writing.                                                                                                                                                                        |
| EBADMSG<br>( <i>Extension</i> )  | Bad message                   | During a system call, e.g. a <code>read()</code> request to a STREAMS device, something arrived at the head of the queue that can't be processed as a data message. What that is will depend on the system call used: control or data messages or passed file descriptors may be involved. |
| EBADR<br>( <i>Extension</i> )    | Bad request descriptor        |                                                                                                                                                                                                                                                                                            |
| EBADRQC<br>( <i>Extension</i> )  | Bad request code              |                                                                                                                                                                                                                                                                                            |
| EBADSLT<br>( <i>Extension</i> )  | Bad slot                      |                                                                                                                                                                                                                                                                                            |
| EBFONT<br>( <i>Extension</i> )   | Invalid font file             |                                                                                                                                                                                                                                                                                            |
| EBUSY                            | Resource busy                 | An attempt was made to make use of a system resource that is not currently available, as it is being used by another process in a manner that would have conflicted with the request being made by this process.                                                                           |
| ECHILD                           | No child process present      | <code>wait()</code> or <code>waitpid()</code> was executed by a process that had no existing or unwaited-for child process.                                                                                                                                                                |
| ECHRNG<br>( <i>Extension</i> )   | Channel number out of range   |                                                                                                                                                                                                                                                                                            |
| ECOMM<br>( <i>Extension</i> )    | Communication error on send   | This error is related to the use of RFS (remote file sharing). It occurs when the current process is waiting for a message from a remote system, and the virtual connection fails.                                                                                                         |

| Symbolic error code                  | Error message                                                      | Meaning                                                                                                                                                                                                                                                 |
|--------------------------------------|--------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ECONNABORTED<br>( <i>Extension</i> ) | Connection aborted by software                                     | A connection abort was caused internally by the host system.                                                                                                                                                                                            |
| ECONNREFUSED<br>( <i>Extension</i> ) | Connection refused                                                 | An attempt was made to connect to a service that is inactive on the remote system.                                                                                                                                                                      |
| ECONNRESET<br>( <i>Extension</i> )   | Connection reset to default setting by peer                        | The connection was aborted by the peer. This normally results from a timeout or a reboot of the system.                                                                                                                                                 |
| EDEADLK                              | Risk of a deadlock                                                 | An attempt was made to lock a system resource that would have resulted in a deadlock situation.                                                                                                                                                         |
| EDEADLOCK<br>( <i>Extension</i> )    | File closed due to deadlock error                                  |                                                                                                                                                                                                                                                         |
| EDESTADDRREQ<br>( <i>Extension</i> ) | Destination address required                                       | The destination address for an operation on a transport endpoint was not specified.                                                                                                                                                                     |
| EDMS<br>( <i>Extension</i> )         | dms error on <code>%.8s</code> ,<br>error code = <code>%.8s</code> |                                                                                                                                                                                                                                                         |
| EDOM                                 | Domain error                                                       | An input argument is outside the defined domain of the mathematical function. (defined in the <i>ISO C Standard</i> ).                                                                                                                                  |
| EOF<br>( <i>Extension</i> )          | Attempt to read beyond end of file (cmd <code>%.8s</code> )        |                                                                                                                                                                                                                                                         |
| EEXIST                               | File exists                                                        | An existing file was specified in an inappropriate context, e.g. when calling the <code>link()</code> function.                                                                                                                                         |
| EFAULT                               | Bad address                                                        | A program has referred to data that is out of the permitted address space. Since the reliable detection of this error cannot be guaranteed, it may result in the generation of a signal, indicating an address violation, which is sent to the process. |
| EFBIG                                | File too large                                                     | The size of a file would exceed the maximum file size or <code>{FCHR_MAX}</code> .                                                                                                                                                                      |
| EHOSTDOWN<br>( <i>Extension</i> )    | Host is down                                                       | A transport provider operation failed because the destination host was down.                                                                                                                                                                            |
| EHOSTUNREACH<br>( <i>Extension</i> ) | Host is unreachable (no known route)                               | A transport provider operation was attempted to an unreachable host.                                                                                                                                                                                    |
| EIDRM                                | Identifier removed                                                 | An identifier was removed from the system during interprocess communication.                                                                                                                                                                            |
| EIKEYOFLW<br>( <i>Extension</i> )    | ISAM-KEY overflow                                                  |                                                                                                                                                                                                                                                         |



| Symbolic error code                 | Error message                    | Meaning                                                                                                                                                                                                                                                                          |
|-------------------------------------|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EILSEQ                              | Illegal byte sequence            | A wide character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide character code.                                                                                                                             |
| EINPROGRESS<br>( <i>Extension</i> ) | Operation now in progress        | An operation that takes a long time to complete (such as a <code>connect</code> ) was attempted on a non-blocking object.                                                                                                                                                        |
| EINTR                               | Interrupted system call          | A signal was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call may return this condition (see also <code>signal.h</code> ).                                                 |
| EINVAL                              | Invalid argument                 | An invalid argument was used, e.g. an undefined signal for <code>signal()</code> or <code>kill()</code> .                                                                                                                                                                        |
| EINVAL<br>( <i>Extension</i> )      | Access to a BS2000 file          | A function which expects a POSIX file as an argument was passed to a BS2000 file.                                                                                                                                                                                                |
| EIO                                 | I/O error                        | A physical I/O error has occurred. In some cases, this error may not appear until the next system call (e.g. an error during <code>write()</code> may be reported after <code>close()</code> ). Any other error-causing operation may cause the EIO error indication to be lost. |
| EIORESID<br>( <i>Extension</i> )    | Block not fully transferred      |                                                                                                                                                                                                                                                                                  |
| EISCONN<br>( <i>Extension</i> )     | Socket is connected              | A connect request has been made, or a <code>sendto</code> or <code>sendmsg</code> specified a destination address, even though the endpoint was already connected.                                                                                                               |
| EISDIR                              | Is a directory instead of a file | An attempt was made to open a directory with write mode specified.                                                                                                                                                                                                               |
| EISNAM<br>( <i>Extension</i> )      | Named type file                  |                                                                                                                                                                                                                                                                                  |
| EL2HLT<br>( <i>Extension</i> )      | Level 2 halted                   |                                                                                                                                                                                                                                                                                  |
| EL2NSYNC<br>( <i>Extension</i> )    | Level 2 not synchronized         |                                                                                                                                                                                                                                                                                  |
| EL3HLT<br>( <i>Extension</i> )      | Level 3 halted                   |                                                                                                                                                                                                                                                                                  |
| EL3RST<br>( <i>Extension</i> )      | Level 3 reset                    |                                                                                                                                                                                                                                                                                  |

| Symbolic error code               | Error message                                                      | Meaning                                                                                                                                                                                                                                                                                  |
|-----------------------------------|--------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ELIBACC<br>( <i>Extension</i> )   | Cannot access a needed shared library                              | An attempt was made to start an executable file that requires a shared library which doesn't exist or for which the user has no access permission.                                                                                                                                       |
| ELIBBAD<br>( <i>Extension</i> )   | Accessing a corrupted shared library                               | An attempt was made to start an executable file that requires a shared library. This was not possible, since the shared library is probably corrupted.                                                                                                                                   |
| ELIBEXEC<br>( <i>Extension</i> )  | Cannot directly execute a shared library directly                  | An attempt was made to <code>exec()</code> a shared library directly.                                                                                                                                                                                                                    |
| ELIBMAX<br>( <i>Extension</i> )   | Maximum limit for shared libraries exceeded                        | An attempt was made to start an executable file that requires more shared libraries than is allowed on the current configuration of the system.                                                                                                                                          |
| ELIBSCN<br>( <i>Extension</i> )   | <code>.lib</code> section in <code>a.out</code> corrupted          | An attempt was made to start an executable file that requires a shared library, and there was erroneous data in the <code>.lib</code> section of the <code>a.out</code> . <code>exec</code> uses the <code>.lib</code> section to obtain the information required from shared libraries. |
| ELNRNG<br>( <i>Extension</i> )    | Link number out of range                                           |                                                                                                                                                                                                                                                                                          |
| ELOOP<br>( <i>Extension</i> )     | Too many symbolic links                                            | The number of symbolic links encountered on traversing a path exceeds <code>{MAXSYMLINKS}</code> .                                                                                                                                                                                       |
| EMACRO<br>( <i>Extension</i> )    | Macro error: the macro <code>%.8s</code> returns <code>%.8s</code> |                                                                                                                                                                                                                                                                                          |
| EMFILE                            | Too many open files                                                | An attempt was made to open more than the maximum number of <code>{OPEN_MAX}</code> file descriptors allowed in this process.                                                                                                                                                            |
| EMLINK                            | Too many links                                                     | An attempt was made to have the link count of a single file exceed <code>{LINK_MAX}</code> .                                                                                                                                                                                             |
| EMODE<br>( <i>Extension</i> )     | Invalid open mode<br>( <code>cmd %.8s</code> )                     |                                                                                                                                                                                                                                                                                          |
| EMSGSIZE<br>( <i>Extension</i> )  | Message too long                                                   | A message sent on a transport provider was larger than the internal message buffer or some other network limit.                                                                                                                                                                          |
| EMULTIHOP<br>( <i>Extension</i> ) | Multi-hops not possible                                            | An attempt was made when using RFS (remote file sharing) to access remote resources that cannot be accessed directly.                                                                                                                                                                    |
| ENAME<br>( <i>Extension</i> )     | Invalid file name<br>( <code>cmd %.8s</code> )                     |                                                                                                                                                                                                                                                                                          |

| Symbolic error code                 | Error message                               | Meaning                                                                                                                                                                                   |
|-------------------------------------|---------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ENAMETOOLONG                        | Pathname too long                           | The length of a pathname exceeds <code>{PATH_MAX}</code> , or a pathname component is longer than <code>{NAME_MAX}</code> , and <code>{_POSIX_NO_TRUNC}</code> is in effect for the file. |
| ENAVAIL<br>( <i>Extension</i> )     | No XENIX semaphores available               |                                                                                                                                                                                           |
| ENETDOWN<br>( <i>Extension</i> )    | Network is down                             | An operation encountered a dead network.                                                                                                                                                  |
| ENETRESET<br>( <i>Extension</i> )   | Network dropped connection because of reset | The host crashed and was rebooted.                                                                                                                                                        |
| ENETUNREACH<br>( <i>Extension</i> ) | Network is unreachable                      | An operation was attempted to an unreachable network.                                                                                                                                     |
| ENFILE                              | Too many files open in system               | The system has reached its predefined limit for simultaneously open files and temporarily cannot accept requests to open another one.                                                     |
| ENOANO<br>( <i>Extension</i> )      | No anode                                    |                                                                                                                                                                                           |
| ENOBUFS<br>( <i>Extension</i> )     | No buffer space available                   | An operation on a transport endpoint was not performed because the system lacked sufficient buffer space or because a queue was full.                                                     |
| ENOCSI<br>( <i>Extension</i> )      | No CSI structure available                  |                                                                                                                                                                                           |
| ENODATA<br>( <i>Extension</i> )     | No data available                           |                                                                                                                                                                                           |
| ENODEV                              | Device unknown                              | An attempt was made to apply an inappropriate function to a device; for example, trying to read from a write-only device such as a printer.                                               |
| ENOENT                              | No such file or directory                   | A component of a specified pathname does not exist, or the pathname is an empty string.                                                                                                   |
| ENOEXEC                             | Executable file format error                | A request is made to execute a file that, although it has the appropriate permissions, is not in the format required for executable files.                                                |
| ENOLCK                              | No record locks available                   | A system-imposed limit on the number of simultaneous file and record locks has been reached and no more are currently available.                                                          |
| ENOLINK<br>( <i>Extension</i> )     | Virtual link lost                           | When using RFS (remote file sharing), the link (virtual) connecting to a remote system is lost.                                                                                           |

| Symbolic error code                | Error message                  | Meaning                                                                                                                                                                                                                   |
|------------------------------------|--------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ENOMEM                             | Not enough space               | The new process image requires more memory than is available.                                                                                                                                                             |
| ENOMSG                             | No message of the desired type | The message queue does not contain a message of the required type during interprocess communication.                                                                                                                      |
| ENONET<br>( <i>Extension</i> )     | System is not on the network   | When using RFS (remote file sharing), an attempt was made to advertise, unadvertise, mount, or unmount remote resources while the machine has not (yet) executed the proper startup procedures to connect to the network. |
| ENOPKG<br>( <i>Extension</i> )     | Package not installed          | An attempt was made to use a system call that is part of a package which has not been installed.                                                                                                                          |
| ENOPROTOPT<br>( <i>Extension</i> ) | Protocol not available         | A bad version or level was specified when getting or setting options for a protocol.                                                                                                                                      |
| ENOSPC                             | No space left on a device      | During the <code>write()</code> function on a regular file or when extending a directory, there is no free space left on the device.                                                                                      |
| ENOSR<br>( <i>Extension</i> )      | Out of stream resources        | During an <code>open()</code> , either no STREAMS queues or no STREAMS header structures were available. This is a temporary condition; one may recover from it if other processes release resources.                     |
| ENOSTR<br>( <i>Extension</i> )     | Device not a stream            | An attempt was made to apply a <code>putmsg()</code> or <code>getmsg()</code> on a file that is not a stream.                                                                                                             |
| ENOSUBSYS<br>( <i>Extension</i> )  | POSIX subsystem not ready      |                                                                                                                                                                                                                           |
| ENOSYS                             | Function not implemented       | An attempt was made to use a function that is not available under the current implementation.                                                                                                                             |
| ENOTBLK<br>( <i>Extension</i> )    | Not a block-special file       | A non-block-special file was specified, but a block device was required.                                                                                                                                                  |
| ENOTCONN<br>( <i>Extension</i> )   | Socket not connected           | A request to send or receive data was disallowed because the transport endpoint was not connected and (when sending a datagram) no address was supplied.                                                                  |
| ENOTDIR                            | Not a directory                | A component of the specified pathname exists, but it is not a directory, when a directory was expected.                                                                                                                   |
| ENOTEMPTY                          | Directory not empty            | A directory with entries other than dot and dot-dot was supplied when an empty directory was expected.                                                                                                                    |

| <b>Symbolic error code</b>              | <b>Error message</b>                          | <b>Meaning</b>                                                                                                                                                                                           |
|-----------------------------------------|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ENOTNAM<br>( <i>Extension</i> )         | Not a XENIX named type file                   |                                                                                                                                                                                                          |
| ENOTSOCK<br>( <i>Extension</i> )        | Socket operation on non-socket                |                                                                                                                                                                                                          |
| ENOTTY                                  | Inappropriate I/O control operation           | A control function has been attempted for a file or special file for which the operation is inappropriate.                                                                                               |
| ENOTUNIQ<br>( <i>Extension</i> )        | Name not unique on network                    |                                                                                                                                                                                                          |
| ENXIO                                   | Device or address not available               | Input or output on a special file referred to a device that does not exist, or made a request beyond the limits of the device. This error may also occur when, for example, a tape drive is not on-line. |
| EOPNOTSUPP<br>( <i>Extension</i> )      | Operation not supported on transport endpoint | An attempt was made, e.g. to accept a connection on a datagram transport endpoint.                                                                                                                       |
| EOPR<br>( <i>Extension</i> )            | Illegal operation<br>(cmd % . 8s)             |                                                                                                                                                                                                          |
| EOVERFLOW<br>( <i>Extension</i> )       | Value too large for defined data type         |                                                                                                                                                                                                          |
| EPERM                                   | Not a system administrator                    | An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resource.                                                                |
| EPFNOSUPPORT<br>( <i>Extension</i> )    | Protocol family not supported                 | The protocol family has not been configured in the system or no implementation for it exists. This error occurs with Internet protocols.                                                                 |
| EPIPE                                   | Broken pipe                                   | An attempt was made to write on a pipe or FIFO for which there is no process to read the data.                                                                                                           |
| EPROTO<br>( <i>Extension</i> )          | Protocol error                                | This error is device-specific, but is generally not related to a hardware failure.                                                                                                                       |
| EPROTONOSUPPORT<br>( <i>Extension</i> ) | Protocol not supported                        | The protocol has not been configured in the current system or no support for it exists.                                                                                                                  |
| EPROTOTYPE<br>( <i>Extension</i> )      | Invalid protocol type for socket              | A protocol that does not support the semantics of the required socket type was specified.                                                                                                                |
| ERANGE                                  | Result too large or too small                 | The result of the function is too large or too small to be represented in the available space (defined in the <i>ISO C Standard</i> ).                                                                   |
| EREMCHG<br>( <i>Extension</i> )         | Address for remote access changed             |                                                                                                                                                                                                          |

| Symbolic error code                     | Error message                               | Meaning                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EREMOTE<br>( <i>Extension</i> )         | Resource is not local                       | This error may occur for the following reasons when using remote file sharing (RFS): <ul style="list-style-type: none"> <li>– An attempt is made to advertise a resource which is not on the local system.</li> <li>– An attempt is made to mount/unmount a device (or pathname) that is on a remote system.</li> <li>– An attempt is made to mount/unmount a pathname that is on a remote system.</li> </ul> |
| EREMOTEIO<br>( <i>Extension</i> )       | Remote I/O error                            |                                                                                                                                                                                                                                                                                                                                                                                                               |
| EREPL<br>( <i>Extension</i> )           | Error that only occurs on opening           |                                                                                                                                                                                                                                                                                                                                                                                                               |
| ERESTART<br>( <i>Extension</i> )        | Restartable system call                     |                                                                                                                                                                                                                                                                                                                                                                                                               |
| EROFS                                   | Read-only file system                       | An attempt was made to update a file or directory on a file system that is read-only.                                                                                                                                                                                                                                                                                                                         |
| ESHUTDOWN<br>( <i>Extension</i> )       | Cannot send after socket is closed          | A request to send data was disallowed because the transport endpoint has already been shut down.                                                                                                                                                                                                                                                                                                              |
| ESOCKTNOSUPPORT<br>( <i>Extension</i> ) | Socket type not supported                   | Support for the socket type is not configured in the current system or no support for it exists.                                                                                                                                                                                                                                                                                                              |
| ESPIPE                                  | Invalid seek statement                      | An attempt was made to access the file offset associated with a pipe or FIFO.                                                                                                                                                                                                                                                                                                                                 |
| ESRCH                                   | No such process                             | No process can be found corresponding to that specified by the given process ID.                                                                                                                                                                                                                                                                                                                              |
| ESRMNT<br>( <i>Extension</i> )          | Srmount error                               | This error relates to the use of RFS (remote file sharing). It occurs when an attempt is made to stop RFS while resources are still mounted on remote systems, or when an attempt is made to advertise a resource with a client list that does not include a remote system on which the resource is currently mounted.                                                                                        |
| ESSNOTAVAIL<br>( <i>Extension</i> )     | Subsystem not available                     |                                                                                                                                                                                                                                                                                                                                                                                                               |
| ESTALE<br>( <i>Extension</i> )          | Stale NFS file handling                     |                                                                                                                                                                                                                                                                                                                                                                                                               |
| ERESTART<br>( <i>Extension</i> )        | Interrupted system call should be restarted |                                                                                                                                                                                                                                                                                                                                                                                                               |

| Symbolic error code                  | Error message                                | Meaning                                                                                                                                                                                                                                                                                               |
|--------------------------------------|----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ESTRPIPE<br>( <i>Extension</i> )     | Streams pipe error (not externally visible)  |                                                                                                                                                                                                                                                                                                       |
| ETIME<br>( <i>Extension</i> )        | Timer expired                                | The timer set for <code>ioctl()</code> has expired. The cause of this error is device-specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too low for the specific function. The status of the <code>ioctl()</code> function will be indeterminate. |
| ETIMEDOUT<br>( <i>Extension</i> )    | Connection timed out                         | A connect or send request failed because the communication partner did not respond appropriately within a specific period of time. The timeout period will depend on the communication protocol.                                                                                                      |
| ETOOMANYREFS<br>( <i>Extension</i> ) | Too many references: connection not possible |                                                                                                                                                                                                                                                                                                       |
| ETXTBSY                              | Text file busy                               | An attempt was made to execute a pure procedure that is currently open for writing or to write to a pure procedure that is being executed.                                                                                                                                                            |
| EUCLEAN<br>( <i>Extension</i> )      | Structure requires cleanup                   |                                                                                                                                                                                                                                                                                                       |
| EUNATCH<br>( <i>Extension</i> )      | Protocol driver not installed                |                                                                                                                                                                                                                                                                                                       |
| EUSERS<br>( <i>Extension</i> )       | Too many users (for POSIX)                   |                                                                                                                                                                                                                                                                                                       |
| EWouldBlock<br>( <i>Extension</i> )  | See EAGAIN                                   |                                                                                                                                                                                                                                                                                                       |
| EXDEV                                | Improper link                                | A link to a file on another file system was attempted.                                                                                                                                                                                                                                                |
| EXFULL<br>( <i>Extension</i> )       | Exchange full                                |                                                                                                                                                                                                                                                                                                       |

See also `errno`, `perror()`, `strerror()`, section "Error handling" on page 126.

## fcntl.h - file control options

**Syntax**      `#include <fcntl.h>`

**Description** The `fcntl.h` header defines the following requests and parameters for use by the functions `fcntl()` and `open()`.

Unique values for commands specified with `fcntl()`:

|                         |                                                                               |
|-------------------------|-------------------------------------------------------------------------------|
| <code>F_DUPFD</code>    | Duplicate file descriptor                                                     |
| <code>F_GETFD</code>    | Get file descriptor flags                                                     |
| <code>F_SETFD</code>    | Set file descriptor flags                                                     |
| <code>F_GETFL</code>    | Get file status flags                                                         |
| <code>F_SETFL</code>    | Set file status flags                                                         |
| <code>F_GETLK</code>    | Get record locking information                                                |
| <code>F_GETLK64</code>  | Get record locking information using the flock64 structure                    |
| <code>F_SETLK</code>    | Set record locking information                                                |
| <code>F_SETLK64</code>  | Set record locking information using the flock64 structure                    |
| <code>F_SETLKW</code>   | Set record locking information; wait if blocked                               |
| <code>F_SETLKW64</code> | Set record locking information; wait if blocked (using the flock64 structure) |

File status flags used for `open()` and `fcntl()`:

|                          |                                                                                                                            |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>O_LARGEFILE</code> | Maximum offset in the open file descriptor is the maximum value that can be represented by an <code>off64_t</code> object. |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------|

File descriptor flags used for `fcntl()`:

|                         |                                                                                                      |
|-------------------------|------------------------------------------------------------------------------------------------------|
| <code>FD_CLOEXEC</code> | Close the file descriptor upon execution of an <code>exec</code> function (see <code>exec()</code> ) |
|-------------------------|------------------------------------------------------------------------------------------------------|

Unique values for `l_type` used for record locking with `fcntl()`:

|                      |                         |
|----------------------|-------------------------|
| <code>F_RDLCK</code> | Shared or read lock     |
| <code>F_UNLCK</code> | Unlock                  |
| <code>F_WRLCK</code> | Exclusive or write lock |



The values used for `l_whence`, `SEEK_SET`, `SEEK_CUR` and `SEEK_END` are defined as described in `unistd.h`.

The following sets of values for the *oflag* argument to `open()` are bit-wise distinct:

`O_CREAT`      Create file if it does not exist  
`O_EXCL`        Exclusive use  
`O_NOCTTY`     Do not assign controlling terminal  
`O_TRUNC`       Truncate file length

File status flags used for `open()` and `fcntl()`:

`O_APPEND`     Set append mode  
`O_NONBLOCK`   Non-blocking mode (POSIX)  
`O_SYNC`        Synchronous writes

Mask for use with file access modes:

`O_ACCMODE`    Mask for file access modes

File access modes used for `open()` and `fcntl()`:

`O_RDONLY`     Open for reading only  
`O_RDWR`        Open for reading and writing  
`O_WRONLY`     Open for writing only

The symbolic names of the values of `mode_t` are defined as described in `sys/types.h`.

The structure  `flock` describes a file lock. It includes the following members:

```
short  l_type;      /* Type of lock */
short  l_whence;   /* Offset type */
off_t   l_start;   /* Relative offset in bytes */
off_t   l_len;     /* Size; if 0 then until EOF (end of file) */
long    l_sysid;   /* Returned with F_GETLK */
pid_t   l_pid;    /* Process ID of the process holding the lock;
                  /* Returned with F_GETLK */
```

The structure `flock64` is a 64-bit version of `flock` and describes a file lock. It includes the following members:

```
short  l_type;      /* Type of lock */
short  l_whence;    /* Offset type */
off64_t l_start;    /* Relative offset in bytes */
off64_t l_len;      /* Size; if 0 then until EOF (end of file) */
long   l_sysid;     /* Returned with F_GETLK */
pid_t  l_pid;       /* Process ID of the process holding the lock;
                    /* Returned with F_GETLK */
```

The `mode_t`, `off_t` and `pid_t` types are defined as described in `sys/stat.h`.

The following are declared as functions:

```
int creat(const *path, mode_t mode);
int creat64(const char *path, mode_t mode)
int fcntl(int fildes, int cmd, ...);
int open(const char *path, int oflag, ...);
int open64(const char *path, int oflag, ...);
```

## float.h - types for floating-point numbers

Syntax `#include <float.h>`

Description The characteristics of floating types are defined in terms of a model which describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.

The following parameters are used to define the model for each floating-point type:

- s* Sign ( $\pm 1$ )
- b* Base or radix of exponent representation (an integer  $> 1$ )
- e* Exponent (an integer between a minimum  $e_{min}$  and a maximum  $e_{max}$ )
- p* Precision (the number of significant digits to the base *b*)
- $f_k$  Non-negative integers less than *b* (the significant digits)

A normalized floating-point number  $x$  ( $f_l > 0$  if  $x \neq 0$ ) is defined by the following model:

$$x = s \times b^e \times \sum_{k=1}^p f_k \times b^{-k}, e_{min} \leq e \leq e_{max}$$

FLT\_RADIX is a constant expression suitable for use in the `#if` preprocessing directives. All expressions except FLT\_RADIX and FLT\_ROUNDSD have separate names for all three floating-point types. The floating-point model representation is provided for all macro names except FLT\_ROUNDSD.

The rounding mode for floating-point addition is characterized by the value of FLT\_ROUNDSD :

- 1 Indeterminable
- 0 Toward zero
- 1 To nearest number
- 2 Toward positive infinity
- 3 Toward negative infinity

All other values for FLT\_ROUNDSD characterize implementation-dependent rounding behavior. FLT\_ROUNDSD is defined with the value 0 in `float.h`.

The macro names in the following list are defined as expressions:

| Name            | Value            | Description                                                                                                                                                                                                                                                                                                                                                              |
|-----------------|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FLT_RADIX       | 16               | Radix of exponent representation $b$                                                                                                                                                                                                                                                                                                                                     |
| FLT_MANT_DIG    | 6                | Number of significant digits to the base FLT_RADIX, $p$                                                                                                                                                                                                                                                                                                                  |
| DBL_MANT_DIG    | 14               |                                                                                                                                                                                                                                                                                                                                                                          |
| LDBL_MANT_DIG   | ANSI 28<br>KR 14 |                                                                                                                                                                                                                                                                                                                                                                          |
| FLT_DIG         | 6                | Number of decimal digits, $q$ , such that any floating-point number with $q$ decimal digits can be rounded into a floating-point number with $p$ digits to the base $b$ and back again without change to the $q$ decimal digits.<br>$[(p - 1) \times \log_{10} b] + k$<br>The following applies:<br>$k = 1$ , if $b$ is an integral power of 10; otherwise,<br>$k = 0$ . |
| DBL_DIG         | 15               |                                                                                                                                                                                                                                                                                                                                                                          |
| LDBL_DIG        | ANSI 32<br>KR 15 |                                                                                                                                                                                                                                                                                                                                                                          |
| FLT_MIN_EXP     | -64              |                                                                                                                                                                                                                                                                                                                                                                          |
| DBL_MIN_EXP     | -64              | Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number, $e_{min}$                                                                                                                                                                                                                                               |
| LDBL_MIN_EXP    | -64              |                                                                                                                                                                                                                                                                                                                                                                          |
| FLT_MIN_10_EXP  | -79              |                                                                                                                                                                                                                                                                                                                                                                          |
| DBL_MIN_10_EXP  | -79              | Minimum negative integer such that 10 raised to that power minus 1 is in the range of normalized floating point numbers.<br>$\left[ \log_{10} b^{e_{min} - 1} \right]$                                                                                                                                                                                                   |
| LDBL_MIN_10_EXP | -79              |                                                                                                                                                                                                                                                                                                                                                                          |
| FLT_MAX_EXP     | 63               |                                                                                                                                                                                                                                                                                                                                                                          |
| DBL_MAX_EXP     | 63               | Maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number, $e_{max}$                                                                                                                                                                                                                                              |
| LDBL_MAX_EXP    | 63               |                                                                                                                                                                                                                                                                                                                                                                          |
| FLT_MAX_10_EXP  | 75               |                                                                                                                                                                                                                                                                                                                                                                          |
| DBL_MAX_10_EXP  | 75               | Maximum integer such that 10 raised to that power minus 1 is in the range of representable finite floating-point numbers.<br>$[\log_{10}((1 - b^{-p}) \times b^{e_{max}})]$                                                                                                                                                                                              |
| LDBL_MAX_10_EXP | 75               |                                                                                                                                                                                                                                                                                                                                                                          |

The particularly large and small values specified here are approximations of the expressions defined in `float.h`.

| Name     | Value     | Description                                                                             |
|----------|-----------|-----------------------------------------------------------------------------------------|
| FLT_MAX  | 7,2370e75 | Maximum representable finite floating-point number<br>$(1 - b^{-p}) \times b^{e_{max}}$ |
| DBL_MAX  | 7,2370e75 |                                                                                         |
| LDBL_MAX | 7,2370e75 |                                                                                         |

The particularly large and small values specified here are approximations of the expressions defined in `float.h`.

| Name         | Value                          | Description                                                                                                                         |
|--------------|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| FLT_EPSILON  | 9,53674e-7                     | The difference between 1.0 and the least value greater than 1 that is representable in the given floating-point type<br>$b^{(1-p)}$ |
| DBL_EPSILON  | 2,22045e-16                    |                                                                                                                                     |
| LDBL_EPSILON | 3,08149e-33<br>ANSI 9<br>KR 16 |                                                                                                                                     |
| FLT_MIN      | 5,3976054e-79                  | Approximated minimum normalized positive floating-point number representable with <code>float</code><br>$b^{(e_{min} - 1)}$         |
| DBL_MIN      | 5,397605346934028e-79          | Minimum positive floating-point number representable with <code>double</code> and <code>long double</code><br>$b^{(e_{min} - 1)}$   |
| LDBL_MIN     | 5,3976053469340278909e-79      |                                                                                                                                     |

## fmtmsg.h - structure of message display

Syntax `#include <fmtmsg.h>`

Description `fmtmsg.h` defines the following macros, which, when expanded, form integral-constant expressions:

|                         |                                                                      |
|-------------------------|----------------------------------------------------------------------|
| <code>MM_HARD</code>    | Origin of a state is the hardware.                                   |
| <code>MM_SOFT</code>    | Origin of a state is the software.                                   |
| <code>MM_FIRM</code>    | Origin of a state is the firmware.                                   |
| <code>MM_APPL</code>    | Message origin in an application.                                    |
| <code>MM_UTIL</code>    | Message origin in a utility routine.                                 |
| <code>MM_OPSYS</code>   | Message origin in the operating system.                              |
| <code>MM_RECOVER</code> | Recoverable error.                                                   |
| <code>MM_NRECOV</code>  | Nonrecoverable error.                                                |
| <code>MM_HALT</code>    | Critical error which halts processing of the application.            |
| <code>MM_ERROR</code>   | Application has detected a non-critical error.                       |
| <code>MM_WARNING</code> | Application has detected an unusual state (which could be an error). |
| <code>MM_INFO</code>    | Information on a state which does not indicate an error.             |
| <code>MM_NOSEV</code>   | There is no warning level for this message.                          |
| <code>MM_PRINT</code>   | Output message to standard error output.                             |
| <code>MM_CONSOLE</code> | Output message to system console.                                    |

The following table shows the null values and identifiers for the arguments of `fmtmsg()`:

| Argument        | Type                  | Null value            | Identifier              |
|-----------------|-----------------------|-----------------------|-------------------------|
| <i>label</i>    | <code>char*</code>    | <code>(char*)0</code> | <code>MM_NULLLBL</code> |
| <i>severity</i> | <code>int</code>      | <code>0</code>        | <code>MM_NULLSEV</code> |
| <i>class</i>    | <code>long int</code> | <code>0L</code>       | <code>MM_NULLMC</code>  |
| <i>text</i>     | <code>char*</code>    | <code>(char*)0</code> | <code>MM_NULLTXT</code> |
| <i>action</i>   | <code>char*</code>    | <code>(char*)0</code> | <code>MM_NULLACT</code> |
| <i>tag</i>      | <code>char*</code>    | <code>(char*)0</code> | <code>MM_NULLTAG</code> |

`fmtmsg.h` defines the following macros as return values of `fmtmsg()`:

|                       |                                                                                                       |
|-----------------------|-------------------------------------------------------------------------------------------------------|
| <code>MM_OK</code>    | The function was successful.                                                                          |
| <code>MM_NOTOK</code> | The function was not successful.                                                                      |
| <code>MM_NOMSG</code> | The function could not generate a message on the standard error output, but was otherwise successful. |
| <code>MM_NOCON</code> | The function could not generate a message on the system console, but was otherwise successful.        |

The following function is defined:

```
int fmtmsg(long classification, const char *label, int severity,  
           const char *text, const char *action, const char *tag);
```

See also `fmtmsg()`.

## ftw.h - file tree traversal

Syntax `#include <ftw.h>`

Description The `ftw.h` header defines codes for use as the third argument to the user-defined function that is passed as the second argument to `ftw()`:

|         |                                          |
|---------|------------------------------------------|
| FTW_F   | File                                     |
| FTW_D   | Directory                                |
| FTW_DNR | Directory without read permission        |
| FTW_NS  | Unknown type; <code>stat()</code> failed |

The following names are declared as functions:

```
int ftw(const char *path,  
        int (*fn) (const char *, const struct stat *, int), int ndirs);  
int ftw64(const char *path,  
          int (*fn) (const char *, const struct stat64 *, int), int ndirs);  
int nftw64(const char *path,  
           int (*fn) (const char *, const struct stat64 *, int, struct FTW *),  
           int depth, int flags);
```

`ftw.h` defines the `stat` structure and the symbolic names for `st_mode` and the file type test macros as described in `sys/stat.h`.

Notes Inclusion of the `ftw.h` header may also make all symbols from `sys/stat.h` visible.

See also `ftw()`, `sys/stat.h`.

## grp.h - group structure

**Syntax**     `#include <grp.h>`

**Description** The `grp.h` header declares the structure `struct group`, which includes the following members:

|                    |                       |                                                                          |
|--------------------|-----------------------|--------------------------------------------------------------------------|
| <code>char</code>  | <code>*gr_name</code> | Name of the group                                                        |
| <code>gid_t</code> | <code>gr_gid</code>   | Numerical group ID                                                       |
| <code>char</code>  | <code>**gr_mem</code> | Pointer to a null-terminated array of character pointers to member names |

The `gid_t` type is defined as described in `sys/types.h`.

The following names are declared as functions:

```
struct group *getgrgid(gid_t gid);
struct group *getgrgid_r(gid_t gid, struct group *gr, char *name2,
                        size_t siz, struct group **grp, );
struct group *getgrnam(const char *name);
struct group *getgrnam_r(const char *name, struct group *gr, char *name2,
                        size_t siz, struct group **grp);
```

**See also**    `getgrgid()`, `getgrgid()_r`, `getgrnam()`, `getgrnam()_r`, `sys/types.h`.



## iconv.h - codeset conversion facility

Syntax `#include <iconv.h>`

Description The following data type is defined through typedef :

`iconv_t`        Identifies the conversion from one codeset to another.

The following names are declared as functions:

```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

```
size_t iconv(iconv_t cd, char **inbuf, size_t *inbytesleft, char **outbuf,  
              size_t *outbytesleft);
```

```
int iconv_close(iconv_t cd);
```

See also `iconv_open()`, `iconv()`, `iconv_close()`.

## iso646.h - alternative ways of writing operators

Syntax `#include <iso646.h>`

Description The header file `iso646.h` contains the following 11 macros that are expanded to the symbols to the right of the macro and that therefore represent alternative ways of writing the operators:

|                     |                         |                     |                 |                     |                 |
|---------------------|-------------------------|---------------------|-----------------|---------------------|-----------------|
| <code>and</code>    | <code>&amp;&amp;</code> | <code>compl</code>  | <code>~</code>  | <code>or_eq</code>  | <code> =</code> |
| <code>and_eq</code> | <code>&amp;=</code>     | <code>not</code>    | <code>!</code>  | <code>xor</code>    | <code>^</code>  |
| <code>bitand</code> | <code>&amp;</code>      | <code>not_eq</code> | <code>!=</code> | <code>xor_eq</code> | <code>^=</code> |
| <code>bitor</code>  | <code> </code>          | <code>or</code>     | <code>  </code> |                     |                 |

## langinfo.h - language information constants

Syntax `#include <langinfo.h>`

Description `langinfo.h` contains the constants used to identify items of `langinfo` data (see also `nl_langinfo()`). The type of these constants is given in `nl_types.h`.

The following constants are defined on all X/Open-conformant systems. The entries in the "Category" column indicate in which `setlocale()` category each item is defined.

| Constant   | Category | Meaning                                         |
|------------|----------|-------------------------------------------------|
| CODESET    | LC_CTYPE | Codeset name                                    |
| D_T_FMT    | LC_TIME  | String for formatting date and time             |
| D_FMT      | LC_TIME  | Date format string                              |
| T_FMT      | LC_TIME  | Time format string                              |
| T_FMT_AMPM | LC_TIME  | Time format string for a.m. or p.m.             |
| AM_STR     | LC_TIME  | Ante Meridian affix (a.m.)                      |
| PM_STR     | LC_TIME  | Post Meridian affix (p.m.)                      |
| DAY_1      | LC_TIME  | Name of the first day of the week (e.g. Sunday) |
| DAY_2      | LC_TIME  | Name of second day of the week (e.g. Monday)    |
| DAY_3      | LC_TIME  | Name of third day of the week (e.g. Tuesday)    |
| DAY_4      | LC_TIME  | Name of fourth day of the week (e.g. Wednesday) |
| DAY_5      | LC_TIME  | Name of fifth day of the week (e.g. Thursday)   |
| DAY_6      | LC_TIME  | Name of sixth day of the week (e.g. Friday)     |
| DAY_7      | LC_TIME  | Name of seventh day of the week (e.g. Saturday) |
| ABDAY_1    | LC_TIME  | Abbreviated name of the first day of the week   |
| ABDAY_2    | LC_TIME  | Abbreviated name of the second day of the week  |
| ABDAY_3    | LC_TIME  | Abbreviated name of the third day of the week   |
| ABDAY_4    | LC_TIME  | Abbreviated name of the fourth day of the week  |
| ABDAY_5    | LC_TIME  | Abbreviated name of the fifth day of the week   |
| ABDAY_6    | LC_TIME  | Abbreviated name of the sixth day of the week   |
| ABDAY_7    | LC_TIME  | Abbreviated name of the seventh day of the week |
| MON_1      | LC_TIME  | Name of the first month of the year             |
| MON_2      | LC_TIME  | Name of the second month                        |
| MON_3      | LC_TIME  | Name of the third month                         |
| MON_4      | LC_TIME  | Name of the fourth month                        |
| MON_5      | LC_TIME  | Name of the fifth month                         |

| <b>Constant</b> | <b>Category</b> | <b>Meaning</b>                                                                                         |
|-----------------|-----------------|--------------------------------------------------------------------------------------------------------|
| MON_6           | LC_TIME         | Name of the sixth month                                                                                |
| MON_7           | LC_TIME         | Name of the seventh month                                                                              |
| MON_8           | LC_TIME         | Name of the eighth month                                                                               |
| MON_9           | LC_TIME         | Name name of the ninth month                                                                           |
| MON_10          | LC_TIME         | Name of the tenth month                                                                                |
| MON_11          | LC_TIME         | Name of the eleventh month                                                                             |
| MON_12          | LC_TIME         | Name of the twelfth month                                                                              |
| ABMON_1         | LC_TIME         | Abbreviated name of the first month                                                                    |
| ABMON_2         | LC_TIME         | Abbreviated name of the second month                                                                   |
| ABMON_3         | LC_TIME         | Abbreviated name of the third month                                                                    |
| ABMON_4         | LC_TIME         | Abbreviated name of the fourth month                                                                   |
| ABMON_5         | LC_TIME         | Abbreviated name of the fifth month                                                                    |
| ABMON_6         | LC_TIME         | Abbreviated name of the sixth month                                                                    |
| ABMON_7         | LC_TIME         | Abbreviated name of the seventh month                                                                  |
| ABMON_8         | LC_TIME         | Abbreviated name of the eighth month                                                                   |
| ABMON_9         | LC_TIME         | Abbreviated name of the ninth month                                                                    |
| ABMON_10        | LC_TIME         | Abbreviated name of the tenth month                                                                    |
| ABMON_11        | LC_TIME         | Abbreviated name of the eleventh month                                                                 |
| ABMON_12        | LC_TIME         | Abbreviated name of the twelfth month                                                                  |
| ERA             | LC_TIME         | Era description segments                                                                               |
| ERA_D_FMT       | LC_TIME         | Era date format string                                                                                 |
| ERA_D_T_FMT     | LC_TIME         | Era date and time format string                                                                        |
| ERA_T_FMT       | LC_TIME         | Era time format string                                                                                 |
| ALT_DIGITS      | LC_TIME         | Alternative symbols for digits                                                                         |
| RADIXCHAR       | LC_NUMERIC      | Radix character                                                                                        |
| THOUSEP         | LC_NUMERIC      | Separator for thousands                                                                                |
| YESEXPR         | LC_MESSAGES     | Affirmative response expression                                                                        |
| NOEXPR          | LC_MESSAGES     | Negative response expression                                                                           |
| YESSTR          | LC_MESSAGES     | Affirmative response for yes/no queries. Will no longer be supported by the X/Open-Standard in future. |
| NOSTR           | LC_MESSAGES     | Negative response for yes/no queries. Will no longer be supported by the X/Open-Standard in future.    |

| Constant | Category    | Meaning                                                                                                                                                                                  |
|----------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CRNCYSTR | LC_MONETARY | Currency symbol; preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character. |

The `langinfo.h` header declares the following as a function:

```
char *nl_langinfo(nl_item);
```

**Notes** Wherever possible, users are advised to use functions compatible with those in the ISO C standard to access items of `langinfo` data. In particular, the `strftime()` function should be used to access date and time information defined in the `LC_TIME` category. The `localeconv()` function should be used to access information corresponding to `RADIXCHAR`, `THOUSEP` and `CRNCYSTR`.

**See also** `nl_langinfo()`, `localeconv()`, `strfmon()`, `strftime()`, section "Locale" on page 53.

## libgen.h - function definitions for pattern matching

**Syntax** `#include <libgen.h>`

**Description** `libgen.h` defines the following external variable:

```
extern char* __loc1
```

(TO BE WITHDRAWN)

The variable is used by `regex()` to specify the address of a pattern.

The following functions are defined:

```
char *regcmp(const char *string1, ... );
char *basename (const char *path);
char *dirname (const char *path);
char *regex(const char *rel, const char *subject, ... );
```

**See also** `regcmp()`.

## limits.h - implementation-dependent constants

Syntax `#include <limits.h>`

Description `limits.h` defines various symbolic names. These names are grouped into different categories in the tables below. The names represent various limits on resources that the system imposes on applications.

Symbolic constant names that begin with `_POSIX` are described under `unistd.h`.

Applications should not assume any particular value for a limit. To achieve maximum portability, an application should not require more of a resource than the quantity listed in the "Minimum acceptable value" column.

However, if an application wishes to avail itself of the full amount of a resource available in an implementation, it should make use of the symbolic name defined in `limits.h` which contains the relevant maximum value of the implementation (see column 1 of the table below). It should be noted, however, that many of the limits listed in `limits.h` are not invariant, and at runtime, the actual value of the limit may differ from the value given in this header file, for the following reasons:

- The limit is pathname-dependent.
- The limit differs on the compile and runtime machines.

For these reasons, an application may use `fpathconf()`, `pathconf()` and `sysconf()` to determine the actual value of a limit at runtime.

The items in the list ending in `_MIN` give the most negative values that the mathematical types are guaranteed to be capable of representing. Numbers of a more negative value may be supported on some systems, as indicated by the `limits.h` header on the system, but applications requiring such numbers are not guaranteed to be portable to all systems.

The symbol "\*" in the "Minimum acceptable value" column indicates that there is no guaranteed value across all X/Open-conformant systems.

**Runtime invariant values**

| <b>Name</b>                   | <b>Description</b>                                                                                                                                         | <b>Minimum acceptable value</b>                  |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------|
| ARG_MAX                       | Maximum length of argument to the <code>exec</code> functions including environment data                                                                   | <code>_POSIX_ARG_MAX</code>                      |
| CHILD_MAX                     | Maximum number of processes                                                                                                                                | 25                                               |
| OPEN_MAX                      | Maximum number of simultaneously open files per process                                                                                                    | 20                                               |
| PASS_MAX                      | Maximum number of significant bytes in a password, not including the terminating null byte (Will no longer be supported by the X/Open-Standard in future.) | 8                                                |
| PTHREAD_DESTRUCTOR_ITERATIONS | Maximum number of attempts taken to delete the thread-specific data value of a thread at the end of the thread.                                            | <code>_POSIX_THREAD_DESTRUCTOR_ITERATIONS</code> |
| PTHREAD_KEYS_MAX              | Maximum number of data keys that can be created by a process.                                                                                              | <code>_POSIX_THREAD_KEYS_MAX</code>              |
| PTHREAD_STACK_MIN             | Minimum stack size (in bytes).                                                                                                                             | 0                                                |
| PTHREAD_THREADS_MAX           | Maximum number of threads that can be created per process.                                                                                                 | <code>_POSIX_THREAD_THREADS_MAX</code>           |
| STREAM_MAX                    | Maximum number of streams that one process can have open at one time. Equivalent to the value of <code>FOPEN_MAX</code> (see <code>stdio.h</code> )        | <code>_POSIX_STREAM_MAX</code>                   |
| TZNAME_MAX                    | Maximum number of bytes supported for the name of a timezone (not of the <code>TZ</code> variable)                                                         | <code>_POSIX_TZNAME_BUF</code>                   |

**Pathname variable values**

| <b>Name</b> | <b>Description</b>                                                            | <b>Minimum acceptable value</b> |
|-------------|-------------------------------------------------------------------------------|---------------------------------|
| LINK_MAX    | Maximum number of links to a single file                                      | _POSIX_LINK_MAX                 |
| MAX_CANON   | Maximum number of bytes in a terminal input line for default input processing | _POSIX_MAX_CANON                |
| MAX_INPUT   | Maximum number of bytes allowed in the input queue of a terminal              | _POSIX_MAX_INPUT                |
| NAME_MAX    | Maximum number of bytes in a filename, not including terminating null byte    | _POSIX_NAME_MAX                 |
| PATH_MAX    | Maximum number of bytes in a pathname, including the terminating null byte    | _POSIX_PATH_MAX                 |
| PIPE_BUF    | Maximum number bytes that is guaranteed to be atomic when writing to a pipe   | _POSIX_PIPE_BUF                 |

## Values that can be increased at runtime

The magnitude limitations in the following table are fixed by the respective implementations. An application should assume that the value supplied by the implementation in `limits.h` is the minimum that applies for all applications which run under that implementation. However, instances of this implementation may increase the value specified in `limits.h` at runtime. The actual value supported by a specific instance is provided by the `sysconf()` function.

| Name             | Description                                                                                                                                                                                               | Minimum acceptable value |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| BC_BASE_MAX      | Maximum <i>ibase</i> values allowed by the <code>bc</code> command.                                                                                                                                       | _POSIX2_BC_BASE_MAX      |
| BC_DIM_MAX       | Maximum number of elements permitted in an array by the <code>bc</code> command.                                                                                                                          | _POSIX2_BC_DIM_MAX       |
| BC_SCALE_MAX     | Maximum <i>scale</i> value allowed by the <code>bc</code> command                                                                                                                                         | _POSIX2_BC_SCALE_MAX     |
| BC_STRING_MAX    | Maximum length of a string constant accepted by the <code>bc</code> command.                                                                                                                              | _POSIX2_BC_STRING_MAX    |
| COLL_WEIGHTS_MAX | Maximum number of weights that can be assigned to an entry of the <code>LC_COLLATE</code> order keyword in the locale definition file (see also section "Locale" on page 53)                              | _POSIX2_COLL_WEIGHTS_MAX |
| EXPR_NEST_MAX    | Maximum nesting level for parenthesized expressions in the <code>expr</code> command.                                                                                                                     | _POSIX2_EXPR_NEST_MAX    |
| LINE_MAX         | Maximum length, in bytes, of a command input line (either standard input or another file) if the command permits text processing. The length includes room for the trailing newline.                      | _POSIX2_LINE_MAX         |
| NGROUPS_MAX      | Maximum number of simultaneously present group IDs per process.                                                                                                                                           | 8                        |
| RE_DUP_MAX       | Maximum number of repeated occurrences of a regular expression permitted when using the interval notation <code>{m,n}</code> (see also "Regular expressions" in the manual "POSIX Commands (BS2000/OSD)") | _POSIX2_RE_DUP_MAX       |



## Minimum values

The symbolic constants in the following table are defined in `limits.h` with the values shown. These are symbolic names for the most restrictive values for certain features on systems conforming to the X/Open standard. Related symbolic constants are defined elsewhere in this manual with the actual implementation taken into account. An X/Open-conformant implementation must support minimum values that are not less than those specified in the table. A portable application must not require a larger value for correct operation. The following constants are specified for POSIX and are always defined in `limits.h`. They are fixed.

| Name                               | Description                                                                                            | Value  |
|------------------------------------|--------------------------------------------------------------------------------------------------------|--------|
| <code>_POSIX_ARG_MAX</code>        | Length of the argument string to the <code>exec</code> functions, in bytes, including environment data | 4 096  |
| <code>_POSIX_CHILD_MAX</code>      | Maximum number of simultaneous processes per real user ID                                              | 6      |
| <code>_POSIX_LINK_MAX</code>       | Maximum value for the link count of a file                                                             | 8      |
| <code>_POSIX_MAX_CANON</code>      | Number of bytes in the terminal canonical input queue for (standard input processing)                  | 255    |
| <code>_POSIX_MAX_INPUT</code>      | Number of bytes allowed in a terminal input queue                                                      | 255    |
| <code>_POSIX_NAME_MAX</code>       | Number of bytes in a filename, not including terminating null byte                                     | 14     |
| <code>_POSIX_NGROUPS_MAX</code>    | Number of simultaneous supplementary group IDs per process                                             | 0      |
| <code>_POSIX_OPEN_MAX</code>       | Number of files that one process can have open at any one time                                         | 16     |
| <code>_POSIX_PATH_MAX</code>       | Number of bytes in a pathname                                                                          | 255    |
| <code>_POSIX_PIPE_BUF</code>       | Number of bytes that is guaranteed to be atomic when writing to a pipe                                 | 512    |
| <code>_POSIX_SSIZE_MAX</code>      | Value that can be assigned to an object of type <code>ssize_t</code>                                   | 32 767 |
| <code>_POSIX_STREAM_MAX</code>     | Number of streams that one process can have open at one time                                           | 8      |
| <code>_POSIX_TZNAME_BUF</code>     | Maximum number of bytes supported for the name of a time zone (not of the <code>TZ</code> variable)    | 3      |
| <code>_POSIX2_BC_BASE_MAX</code>   | Maximum <i>obase</i> values allowed by the <code>bc</code> command                                     | 99     |
| <code>_POSIX2_BC_DIM_MAX</code>    | Maximum number of elements permitted in an array by the <code>bc</code> command                        | 2 048  |
| <code>_POSIX2_BC_SCALE_MAX</code>  | Maximum <i>scale</i> value allowed by the <code>bc</code> command                                      | 99     |
| <code>_POSIX2_BC_STRING_MAX</code> | Maximum length of a string constant allowed by the <code>bc</code> command                             | 1 000  |

| Name                     | Description                                                                                                                                                                                      | Value |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|
| _POSIX2_COLL_WEIGHTS_MAX | Maximum number of weights that can be assigned to an entry of the LC_COLLATE order keyword in the locale definition file (see section "Locale" on page 53)                                       | 2     |
| _POSIX2_EXPR_NEST_MAX    | Maximum number of expressions that can be nested within parentheses by the <code>expr</code> command                                                                                             | 32    |
| _POSIX2_LINE_MAX         | Maximum length, in bytes, of a command input line (either standard input or another file) if the command allows text processing. The length includes room for the trailing newline               | 2 048 |
| _POSIX2_RE_DUP_MAX       | Maximum number of repeated occurrences of a regular expression permitted when using the interval notation $\{m,n\}$ (see also "Regular expressions" in the manual "POSIX Commands (BS2000/OSD)") | 255   |

## Numerical limits

The values in the following tables are defined in `limits.h` and will be constant expressions suitable for use in `#if` preprocessing directives. Moreover, except for `CHAR_BIT`, `EXDBL_DIG`, `DBL_MAX`, `FLT_DIG`, `FLT_MAX`, `LONG_BIT`, `WORD_BIT` and `MB_LEN_MAX`, the symbolic names will be defined as expressions of the correct type. If the value of an object of type `char` is treated as a signed integer when used in an expression, the value of `CHAR_MIN` is the same as that of `SCHAR_MIN`, and the value of `CHAR_MAX` is the same as that of `SCHAR_MAX`. Otherwise, the value of `CHAR_MIN` is 0, and the value of `CHAR_MAX` is the same as that of `UCHAR_MAX`.

### Maximum values

| Name                  | Description                                                                                                          | Minimum acceptable value                               |
|-----------------------|----------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------|
| <code>CHAR_BIT</code> | Number of bits in a type <code>char</code>                                                                           | 8                                                      |
| <code>CHAR_MAX</code> | Maximum value of a type <code>char</code>                                                                            | <code>UCHAR_MAX</code><br>or<br><code>SCHAR_MAX</code> |
| <code>DBL_DIG</code>  | Significant digits for a type <code>double</code><br>(Will no longer be supported by the X/Open-Standard in future.) | 10                                                     |
| <code>DBL_MAX</code>  | Maximum decimal value of a <code>double</code><br>(Will no longer be supported by the X/Open-Standard in future.)    | 1E+37                                                  |
| <code>FLT_DIG</code>  | Significant digits for a type <code>float</code><br>(Will no longer be supported by the X/Open-Standard in future.)  | 6                                                      |

| Name       | Description                                                                                                      | Minimum acceptable value |
|------------|------------------------------------------------------------------------------------------------------------------|--------------------------|
| FLT_MAX    | Maximum decimal value of a <code>float</code><br>(Will no longer be supported by the X/Open-Standard in future.) | 1E+37                    |
| INT_MAX    | Maximum value of a type <code>int</code>                                                                         | 32 767                   |
| LONG_BIT   | Number of bits in a <code>long</code>                                                                            | 32                       |
| LONG_MAX   | Maximum value of a type <code>long</code>                                                                        | +2 147 483 647           |
| MB_LEN_MAX | Maximum number of bytes in a character, for any supported locale                                                 | 1                        |
| SCHAR_MAX  | Maximum value of a type <code>signed char</code>                                                                 | +127                     |
| SHRT_MAX   | Maximum value of a type <code>short</code>                                                                       | +32 767                  |
| SSIZE_MAX  | Maximum value of a type <code>ssize_t</code>                                                                     | _POSIX_SSIZE_MAX         |
| UCHAR_MAX  | Maximum value of a type <code>unsigned char</code>                                                               | 255                      |
| UINT_MAX   | Maximum value of a type <code>unsigned int</code>                                                                | 65 535                   |
| ULONG_MAX  | Maximum value of a type <code>unsigned long</code>                                                               | 4 294 967 295            |
| USHRT_MAX  | Maximum value of a type <code>unsigned short</code>                                                              | 65 535                   |
| WORD_BIT   | Number of bits in a word or type <code>int</code>                                                                | 16                       |

*Minimum value*

| Name      | Description                                      | Maximum acceptable value |
|-----------|--------------------------------------------------|--------------------------|
| CHAR_MIN  | Minimum value of a type <code>char</code>        | SCHAR_MIN<br>or<br>0     |
| INT_MIN   | Minimum value of a type <code>int</code>         | -32 767                  |
| LONG_MIN  | Minimum value of a type <code>long</code>        | -2 147 483 647           |
| SCHAR_MIN | Minimum value of a type <code>signed char</code> | -127                     |
| SHRT_MIN  | Minimum value of a type <code>short</code>       | -32 767                  |

## Other invariant values

The following constants are defined on all systems in `limits.h`.

| Name               | Description                                                                                                                           | Minimum acceptable value |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------|--------------------------|
| CHARCLASS_NAME_MAX | Maximum number of bytes in a character class name                                                                                     | 14                       |
| NL_ARGMAX          | Maximum value of digit in calls to the <code>printf()</code> and <code>scanf()</code> functions                                       | 9                        |
| NL_LANGMAX         | Maximum number of bytes in a LANG name                                                                                                | 14                       |
| NL_MSGMAX          | Maximum message number                                                                                                                | 32 767                   |
| NL_NMAX            | Maximum number of bytes in an N-to-1 collation mapping                                                                                | *                        |
| NL_SETMAX          | Maximum set number                                                                                                                    | 255                      |
| NL_TEXTMAX         | Maximum number of bytes in a message string                                                                                           | _POSIX2_LINE_MAX         |
| NZERO              | Default process priority                                                                                                              | 20                       |
| TMP_MAX            | Maximum number of unique pathnames generated by <code>tmpnam()</code> (Will no longer be supported by the X/Open-Standard in future.) | 10 000                   |

**Notes** `TMP_MAX` is also defined in `stdio.h` to align with the ISO C standard.

`DBL_DIG`, `DBL_MAX`, `DBL_MIN`, `FLT_DIG`, `FLT_MAX` and `FLT_MIN` are also defined in `float.h`.

**See also** `fpathconf()`, `pathconf()`, `sysconf()`, `unistd.h`.

## locale.h - category macros

**Syntax**     **#include <locale.h>**

**Description** `locale.h` defines the structure `lconv`, which includes at least the following members:

```
char *currency_symbol
char *decimal_point
char frac_digits
char *grouping
char *int_curr_symbol
char int_frac_digits
char *mon_decimal_point
char *mon_grouping
char *mon_thousands_sep
char *negative_sign
char n_cs_precedes
char n_sep_by_space
char n_sign_posn
char *positive_sign
char p_cs_precedes
char p_sep_by_space
char p_sign_posn
char *thousands_sep
```

`locale.h` defines `NULL` and at least the following as macros:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
```

These macros expand to distinct integral-constant expressions for use as the first argument to the `setlocale()` function.

The following names are declared as functions:

```
struct lconv *localeconv(void);
```

```
char setlocale(int category, const char *locale);
```

**See also** `localeconv()`, `setlocale()`, section "Environment variables" on page 71.

## math.h - mathematical functions and constants

Syntax `#include <math.h>`

Description This header file contains the declarations of all math library functions and various C library functions that return floating-point values.

### *Extension*

HUGE Maximum value of a single-precision floating-point number  $\square$

The following mathematical constants are defined as described below to facilitate usage in applications:

|            |                                                                     |
|------------|---------------------------------------------------------------------|
| M_E        | Base of the natural logarithm ( $e$ )                               |
| M_LOG2E    | Logarithm of $e$ to the base 2                                      |
| M_LOG10E   | Logarithm of $e$ to the base 10                                     |
| M_LN2      | Natural logarithm of 2                                              |
| M_LN10     | Natural logarithm of 10                                             |
| M_PI       | $\pi$ , i.e. ratio of the circumference of a circle to its diameter |
| M_PI_2     | $\pi/2$                                                             |
| M_PI_4     | $\pi/4$                                                             |
| M_1_PI     | $1/\pi$                                                             |
| M_2_PI     | $2/\pi$                                                             |
| M_2_SQRTPI | $2/\text{square root of } \pi$                                      |
| M_SQRT2    | Positive square root of 2                                           |
| M_SQRT1_2  | $1/\text{positive square root of } 2$                               |

The following mathematical constants are also defined in this header file:

|          |                                                           |
|----------|-----------------------------------------------------------|
| MAXFLOAT | Maximum value of a single-precision floating-point number |
| HUGE_VAL | Positive infinity of type <code>double</code>             |

The `HUGE_VAL` macro is defined to represent error values that are returned by the math functions. Since the value  $+\infty$  cannot be represented on the hardware, `HUGE_VAL = {DBL_MAX}`.

The following names are declared as functions:

```
double acos(double x);
double asin(double x);
```

```
double atan2(double x);
double atan(double x);
double ceil(double x);
double cos(double x);
double cosh(double x);
double exp(double x);
double fabs(double x);
double floor(double x);
double fmod(double x);
double frexp(double x);
double ldexp(double x);
double log10(double x);
double log(double x);
double modf(double x);
double pow(double x);
double sin(double x);
double sinh(double x);
double sqrt(double x);
double tan(double x);
double tanh(double x);
double erf(double x);
double erfc(double x);
double gamma(double x);
double hypot(double x, double y);
double j0(double x);
double j1(double x);
double jn(int n, double x);
double lgamma(double x);
double y0(double x);
```



```
double y1(double x);  
double yn(int n, double x);  
int isnan(double x);
```

The following external variable is defined:

```
extern int signgam;
```

For the definitions of various machine-dependent constants, see `values()`.

**See also** `matherr()`, `values()`, `acos()`, `asin()`, `atan()`, `atan2()`, `ceil()`, `cos()`, `cosh()`, `erf()`, `exp()`, `fabs()`, `floor()`, `fmod()`, `frexp()`, `gamma()`, `hypot()`, `isnan()`, `j0()`, `ldexp()`, `lgamma()`, `log()`, `log10()`, `modf()`, `pow()`, `sin()`, `sinh()`, `sqrt()`, `tan()`, `tanh()`, `y0()`.

## monetary.h - types for monetary values

**Syntax**     `#include <monetary.h>`

**Description** The `monetary.h` header defines the following data types through typedef:

`size_t`        As described in `stddef.h`.

`ssize_t`       As described in `sys/types.h`.

The following name is declared as a function:

```
ssize_t strfmon(char *s, size_t maxsize, const char *format, ...);
```

**See also**     `strfmon()`.

## ndbm.h - definitions of operations for ndbm database

**Syntax**     `#include <ndbm.h>`

**Description** `ndbm.h` defines the type `date` as the following structure:

```
typedef struct {
    void *dptr;           /* Pointer to the application data */
    size_t dsize;        /* Length of object to which dptr points */
} date
```

`ndbm.h` defines the type `DBM` with the following structure:

```
typedef struct {
    int dbm_dirf;        /* Open directory */
    int dbm_pagf;        /* Open page file */
    int dbm_flags;       /* Flags, see below */
    long dbm_maxbno;     /* Last 'bit' in directory */
    long dbm_bitno;     /* Number of the current bit */
    long dbm_hmask;     /* Hash mask */
    long dbm_blkptr;    /* Current block for dbm_nextkey */
    int dbm_keyptr;     /* Current key for dbm_nextkey */
    long dbm_blkno;     /* Current page for read or write */
    long dbm_pagbno;    /* Current page in page buffer */
    char dbm_pagbuf[_PBLKSIZ]; /* Page file block buffer */
    long dbm_dirbno;    /* Current block in dirbuf */
    char dbm_dirbuf[_DBLKSIZ]; /* Block buffer for directory */
} DBM;
```

Furthermore, the following constants are defined, which indicate possible values of the *store\_mode* parameter of `dbm_store()`:

`DBM_INSERT` only new entries may be inserted

`DBM_REPLACE` existing entries may be replaced

The following functions are defined:

```
int dbm_clearerr(DBM *db);
void dbm_close(DBM *db);
int dbm_delete(DBM *db, date key);
int dbm_error(DBM *db);
date dbm_fetch(DBM *db, date key);
date dbm_firstkey(DBM *db);
date dbm_nextkey(DBM *db);
DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
int dbm_store(DBM *db, date key, date content, int store_mode);
```

The following functions are also offered as macros:

```
int dbm_clearerr(DBM *db);
int dbm_error(DBM *db);
```

**See also** `dbm_clearerr()`.

## nl\_types.h - native language data types

Syntax `#include <nl_types.h>`

Description The following data types are defined by typedef:

`nl_catd` Used by the message catalog functions `catopen()`, `catgets()` and `catclose()` to identify a catalog descriptor.

`nl_item` Used by `nl_langinfo()` to identify `langinfo` data. Values of objects of type `nl_item` are defined in `langinfo.h`.

The following constants are defined:

`NL_CAT_LOCALE` Value passed to `catopen()` as *oflag* argument to ensure that message catalog selection depends on the `LC_MESSAGES` locale category and not directly on the `LANG` environment variable.

`NL_SETD` Used by `gencat` when no `$set` directive is specified in a message text source file. This constant can be passed as the value of `set_id` on subsequent calls to `catgets()` to get messages from the default message set. The value of `NL_SETD` is implementation-dependent.

The following names are declared as functions:

```
int catclose(nl_catd catd);
```

```
char *catgets(nl_catd catd, int set_id, int msg_id, const char *str);
```

```
nl_catd catopen(const char *name, int oflag);
```

See also `catclose()`, `catgets()`, `catopen()`, `nl_langinfo()`, `langinfo.h`, and the command `gencat` in the manual "POSIX Commands (BS2000/OSD)".

## poll.h - definitions for poll() function

Syntax `#include <poll.h>`

`poll.h` defines the `pollfd` structure:

```
struct pollfd {
int fd;                /* File descriptor */
short events;          /* Requested events */
short revents;         /* Reported events */
};
```

`poll.h` defines the following type with typedef:

`nfds_t` Unsigned integral data type which specifies the number of file descriptors.

Through ORing of the following symbolic constants in any combination, the `events` and `revents` parameter are constructed in the `pollfd` structure (0 can also be specified):

|                         |                                                                         |
|-------------------------|-------------------------------------------------------------------------|
| <code>POLLIN</code>     | As <code>POLLRDNORM   POLLRDBAND</code> .                               |
| <code>POLLRDNORM</code> | Data with priority = 0 can be read.                                     |
| <code>POLLRDBAND</code> | Data with priority > 0 can be read.                                     |
| <code>POLLPRI</code>    | Data with a high priority can be read.                                  |
| <code>POLLOUT</code>    | As <code>POLLWRNORM</code> .                                            |
| <code>POLLWRNORM</code> | Data with priority = 0 can be written.                                  |
| <code>POLLERR</code>    | There is an error message (valid for <code>revents</code> only).        |
| <code>POLLHUP</code>    | Connection to device interrupted (valid for <code>revents</code> only). |
| <code>POLLNVAL</code>   | Invalid fd element (valid for <code>revents</code> only).               |

`poll.h` defines the following functions:

```
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

See also `poll()`.

## pwd.h - password structure

Syntax `#include <pwd.h>`

Description The `pwd.h` header provides a definition for the structure `struct passwd`, which includes the following members:

|                    |                          |                                                                   |
|--------------------|--------------------------|-------------------------------------------------------------------|
| <code>char</code>  | <code>*pw_name</code>    | Login name                                                        |
| <code>uid_t</code> | <code>pw_uid</code>      | User ID                                                           |
| <code>gid_t</code> | <code>pw_gid</code>      | Group ID                                                          |
| <code>char</code>  | <code>*pw_dir</code>     | HOME directory                                                    |
| <code>char</code>  | <code>*pw_shell</code>   | Startup program                                                   |
| <code>char</code>  | <code>*pw_passwd</code>  | LOGON password (BS2000)                                           |
| <code>char</code>  | <code>*pw_age</code>     | Password age (SINIX; supported for compatibility reasons)         |
| <code>char</code>  | <code>*pw_comment</code> | Comment line (e.g.: address, room no.)                            |
| <code>char</code>  | <code>*pw_gecos</code>   | Protection attribute (SINIX; supported for compatibility reasons) |

The following names are declared as functions:

```
struct passwd *getpwnam(const char *name);
```

```
int getpwnam_r(const char *name, struct passwd *pw, char *name1, size_t siz, struct passwd **ppw);
```

```
struct passwd *getpwuid(uid_t uid);
```

```
int getpwuid_r(uid_t uid, struct passwd *pw, char *name1, size_t siz, struct passwd **ppw);
```

## re\_comp.h - comparison of regular expressions with re\_comp

Syntax `#include <re_comp.h>`

Description The following functions are declared:

```
char *re_comp(const char *string);
int re_exec(const char *string);
```

Note This header file will no longer be available in the next version.

See also `re_comp()`.

## regexp.h - declarations for regular expressions

Syntax `#include <regexp.h>`

Description The following names are declared as functions:

```
int advance(const char *string, const char *expbuf);
char *compile(char *instring, const char *expbuf);
int step(const char *string, const char *expbuf);
```

The following external variables are defined:

```
extern char *loc1;
extern char *loc2;
extern char *locs;
```

Notes Will no longer be supported by the X/Open-Standard in future.

See also `regexp()`.

## search.h - search tables

```
#include <search.h>
```

**Description** The `search.h` header provides a typedef `ENTRY` for the data type `struct entry` containing the following members:

```
char      *key;
char      *data;
```

The enumeration data types `ACTION` and `VISIT` are defined through typedef as follows:

```
enum { FIND, ENTER } ACTION;
enum { preorder, postorder, endorder, leaf } VISIT;
```

The following names are declared as functions:

```
int hcreate(size_t nel);
void hdestroy(void);
ENTRY *hsearch(ENTRY item, ACTION action);
void *lfind(const void *key, const void *base, size_t *nelp, size_t width,
            int (*compar)(const void *, const void*));
void *lsearch(const void *key, const void *base, size_t *nelp, size_t
              width, int (*compar)(const void *, const void*))
void *tsearch(const void *key, void **rootp,
              int (*compar)(const void *, const void*))
void *tdelete(const void *key, void **rootp,
              int (*compar)(const void *, const void*))
void *tfind(const void *key, void **rootp,
            int (*compar)(const void *, const void*))
void *twalk(const void *root, void (*action)(const void *, VISIT, int))
```

**See also** `hsearch()`, `lsearch()`, `tsearch()`, `sys/types.h`.



## setjmp.h - stack environment declarations

**Syntax**      `#include <setjmp.h>`

**Description** The `setjmp.h` header contains type definitions with `typedef` for the data types `jmp_buf` and `sigjmp_buf`.

`longjmp()` and `siglongjmp()` are declared as functions:

```
void longjmp(jmp_buf env, int val);
```

```
void siglongjmp(sigjmp_buf env, int val);
```

`setjmp()` is declared as a function, and `sigsetjmp()` is defined as a macro:

```
int setjmp(jmp_buf env);
```

```
int sigsetjmp(sigjmp_buf env, int savemask);
```

**See also**      `longjmp()`, `setjmp()`, `siglongjmp()`, `sigsetjmp()`.

## signal.h - signals

Syntax `#include <signal.h>`

Description `signal.h` declares constants to represent signals that occur in the system. These constants begin with the letters `SIG`. Each signal has a unique, positive integral value. The value 0 is reserved for the null signal. (see also `kill()`). Additional signal that may occur in the system may be defined by an implementation.

The following symbolic constants are defined as signal actions:

|                       |                                                          |
|-----------------------|----------------------------------------------------------|
| <code>SIG_DFL</code>  | Request for default signal handling                      |
| <code>SIG_ERR</code>  | Return value from <code>signal()</code> in case of error |
| <code>SIG_HOLD</code> | Request that signal be held                              |
| <code>SIG_IGN</code>  | Request that signal be ignored                           |

Each of these constants expands to a unique expression of type `void (*)(int)` with a value that matches no declarable function.

The following data types are defined through `typedef`:

|                           |                                                                                                                      |
|---------------------------|----------------------------------------------------------------------------------------------------------------------|
| <code>sig_atomic_t</code> | Integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts |
| <code>sigset_t</code>     | Integral or structure type of an object used to represent sets of signals                                            |
| <code>pid_t</code>        | See <code>sys/types.h</code>                                                                                         |

The following symbolic names for signal numbers - called signals for short - are supported by the library functions described in this manual:

| Signal                                  | Description                                | Default signal handling                                                                                                                                                                                          |
|-----------------------------------------|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>SIGABRT</code>                    | Process abort                              | Abnormal process termination; signal-handling functions may also be executed before termination                                                                                                                  |
| <code>SIGALRM</code>                    | Alarm clock expired<br>STXIT class: RTIMER | Abnormal process termination with all consequences of <code>_exit()</code> , except that the status for <code>wait()</code> is available and <code>waitpid()</code> indicates abnormal termination by the signal |
| <code>SIGBUS</code><br><i>Extension</i> | Address error                              | Abnormal process termination and core dump                                                                                                                                                                       |
| <code>SIGCHLD</code>                    | Child process terminated or stopped        | Ignore signal                                                                                                                                                                                                    |

| <b>Signal</b>               | <b>Description</b>                                                                                                                                                                                                                    | <b>Default signal handling</b>                           |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| SIGCONT                     | Continue executing, if stopped                                                                                                                                                                                                        | Continue stopped process, otherwise, ignore signal       |
| SIGDVZ<br><i>Extension</i>  | Division by 0;<br>(is mapped to SIGFPE)<br>STXIT class: PROCHK                                                                                                                                                                        | See SIGABRT                                              |
| SIGEMT<br><i>Extension</i>  | Interrupted for emulator                                                                                                                                                                                                              | Abnormal process termination and core dump               |
| SIGFPE                      | Error on floating-point operation<br>STXIT class: PROCHK<br>STXIT weights:<br>0x64: Exponent overflow<br>0x68: Division error<br>0x6c: Mantissa=0<br>0x70: Exponent underflow<br>0x74: Decimal overflow<br>0x78: Fixed-point overflow | See SIGABRT                                              |
| SIGHUP                      | Connection to terminal interrupted (hangup)<br>STXIT class: ABEND                                                                                                                                                                     | See SIGALRM                                              |
| SIGILL                      | Illegal instruction<br>STXIT class: PROCHK                                                                                                                                                                                            | See SIGABRT; this signal is not reset after being caught |
| SIGINT                      | Interrupt from terminal<br>STXIT class: ESCPBRK                                                                                                                                                                                       | See SIGALRM                                              |
| SIGIO<br><i>Extension</i>   | I/O on socket possible (equivalent to SIGPOLL)                                                                                                                                                                                        | Ignore signal                                            |
| SIGIOT<br><i>Extension</i>  | Abort                                                                                                                                                                                                                                 | Abnormal process termination and core dump               |
| SIGKILL                     | Unconditional abort that can be neither caught nor ignored (kill)                                                                                                                                                                     | See SIGALRM                                              |
| SIGPIPE                     | Write on a pipe that is not readable                                                                                                                                                                                                  | See SIGALRM                                              |
| SIGPOLL<br><i>Extension</i> | Pollable event                                                                                                                                                                                                                        | Ignore signal                                            |
| SIGPROF<br><i>Extension</i> | Profiling timer (for runtime analyses) expired                                                                                                                                                                                        | Exit                                                     |
| SIGPWR<br><i>Extension</i>  | Power interrupt/Restart                                                                                                                                                                                                               | Ignore signal                                            |
| SIGQUIT                     | Terminal quit signal                                                                                                                                                                                                                  | See SIGABRT                                              |
| SIGSEGV                     | Invalid access to memory segment<br>STXIT class: ERROR                                                                                                                                                                                | See SIGABRT                                              |

| Signal                        | Description                                                    | Default signal handling                                                                    |
|-------------------------------|----------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| SIGSTOP                       | Stop executing (cannot be caught or ignored)                   | Stop process                                                                               |
| SIGSYS<br><i>Extension</i>    | Invalid system call<br>STXIT weight:<br>0xb0: Validation error | Abnormal process termination and core dump                                                 |
| SIGTERM                       | Process termination<br>STXIT class: TERM                       | See SIGALRM                                                                                |
| SIGTIM<br><i>Extension</i>    | Timer;<br>(is mapped to SIGVTALRM)<br>STXIT class: TIMER       | Exit                                                                                       |
| SIGTRAP<br><i>Extension</i>   | Interrupt for tracing                                          | Abnormal process termination and core dump;<br>this signal is not reset after being caught |
| SIGTSTP                       | Stop signal for terminal                                       | Stop process                                                                               |
| SIGTTIN                       | Background process attempting to read                          | Stop process                                                                               |
| SIGTTOU                       | Background process attempting to write                         | Stop process                                                                               |
| SIGURG<br><i>Extension</i>    | Urgent case on socket                                          | Ignore signal                                                                              |
| SIGUSR1                       | User-defined signal 1                                          | See SIGALRM                                                                                |
| SIGUSR2                       | User-defined signal 2                                          | See SIGALRM                                                                                |
| SIGVTALRM<br><i>Extension</i> | STXIT weight:<br>0x20: SETIC interval for CPU-time expired     | Exit                                                                                       |
| SIGWINCH<br><i>Extension</i>  | Change window size                                             | Ignore signal.                                                                             |
| SIGXCPU<br><i>Extension</i>   | CPU time limit exceeded<br>STXIT class: RUNOUT                 | Abnormal process termination and core dump                                                 |
| SIGXFSZ<br><i>Extension</i>   | File size limit exceeded                                       | Abnormal process termination and core dump                                                 |

signal.h declares the structure `sigaction`, which contains at least the following members:

```
void (*sa_handler)(int)           Signal-handling function
sigset_t sa_mask                 Signal set that is blocked during execution of the signal-handling
int sa_flags                     Special flags
```

The following symbolic constants are defined as flag bits:

|              |                                                                                                        |
|--------------|--------------------------------------------------------------------------------------------------------|
| SA_NOCLDSTOP | Do not generate SIGCHLD when children stop.                                                            |
| SIG_BLOCK    | The resulting set is the union of the current set and the signal set pointed to by <i>set</i> .        |
| SIG_UNBLOCK  | The resulting set is the intersection of the current set and the signal set pointed to by <i>set</i> . |
| SIG_SETMASK  | The resulting set is the signal set pointed to by <i>set</i> .                                         |

The following names are declared as functions:

```
int kill(pid_t pid, int sig);
int pthread_kill(pthread_t pthr, int signo);
int pthread_sigmask(int signo, const sigset_t *cset, sigset_t *set);
int raise(int sig);
int sigaction(int sig, const struct sigactions *act, struct sigaction *oact);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigismember(const sigset_t *set, int signo);
void (*signal(int sig, void(*func)(int)))(int);
int sigpending(sigset_t *set);
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int sigsuspend(const sigset_t *sigmask);
```

**See also** alarm(), kill(), pthread\_kill(), pthread\_sigmask(), raise(), sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), signal(), sigpending(), sigprocmask(), sigsuspend(), wait(), sys/types.h.

## stdarg.h - edit variable argument list

Syntax `#include <stdarg.h>`

Syntax `void va_start(va_list ap, argN);`  
`type va_arg(va_list ap, type);`  
`void va_end(va_list ap);`

Description `stdarg.h` contains macros that permit the development of portable procedures that accept variable numbers of arguments of different types. Routines that have variable argument lists (such as `printf()`) but do not use these macros are inherently non-portable, since different systems use different argument-passing conventions.

`va_list` is defined for variables used to traverse the list.

The `va_start()` macro is executed before any access to the unnamed arguments. This macro initializes `ap` for subsequent use by `va_arg()` and `va_end()`. `argN` is the identifier of the right-most parameter in the variable parameter list in the function definition. If this parameter is declared with the `register` storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument handling, the behavior is undefined.

The `va_arg()` macro is evaluated as an expression that has the type and value of the next argument in the call. `ap` should have been previously initialized by `va_start()`. Each invocation of `va_arg()` modifies `ap` so that the values of successive arguments are returned in turn. The parameter `type` is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a `*` to `type`. If there is no next argument, or if `type` is not compatible with the type of the next argument (as promoted according to the default argument handling), the behavior is undefined.

The `va_end()` function is used to clean up; it invalidates `ap` `va_start()` is invoked again.

Multiple traversals, each bracketed by `va_start()` and `va_end()`, are possible.

**Example** This example reads pointers to strings into an array via an arguments list. This is done using the function `f1()`, which accepts a maximum of `MAXARGS` arguments. The array is then passed as a single argument to the function `f2()`. The number of pointers is specified by the first argument to `f1()`.

```
#include <stdarg.h>
#define MAXARGS 31

int f2(int, char *);
void f1(int n_ptrs, ...)
{
    va_list ap;
    char *array[MAXARGS];
    int ptr_no = 0;

    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    va_start(ap, n_ptrs);
    while (ptr_no < n_ptrs)
        array[ptr_no++] = va_arg(ap, char*);
    va_end(ap);
    f2(n_ptrs, array);
}
```

The definition of the function or a declaration such as `void f1(int, ...)` should be available at each call to `f1()`.

**Notes** It is up to the calling routine to specify how many arguments are passed, since it is not always possible to determine the number of arguments from the stack frame. For example, `exec1()` is passed a null pointer to signal the end of the list. `printf()` can determine the number of arguments from the format. It is non-portable to specify a second argument of `char`, `short`, or `float` to `va_arg()`, because arguments seen by the called function are not `char`, `short`, or `float`. **C converts `char` and `short` arguments to `int` and converts `float` arguments to `double` before passing them to a function.**

**See also** `vprintf()`.

## stddef.h - standard type definitions

Syntax `#include <stddef.h>`

Description `stddef.h` defines the following symbolic names:

`NULL` Null pointer

`offsetof(type, member-designator)` ;

Integral constant expression of type `size_t`, the value of which is the offset in bytes to the structure member (*member-designator*), from the beginning of its structure (*type*).

The following data types are defined by `typedef`:

`ptrdiff_t` Signed integral type of the result of subtracting two pointers.

`wchar_t` Integral type whose range of values can represent distinct wide character codes for all members of the largest character set specified among the locales supported by the compilation environment; the null character has the code value 0, and each member of the portable character set has a code value equal to its value when used as the lone character in an integer character constant.

`size_t` Unsigned integral type of the result of the `sizeof` operator.

See also `wchar.h`, `sys/types.h`.



## stdio.h - standard buffered input/output

Syntax `#include <stdio.h>`

Description `stdio.h` defines the following macro names as positive integral constants:

|                           |                                                                                                                                                                                                               |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>BUFSIZ</code>       | Size of buffers for <code>stdio.h</code>                                                                                                                                                                      |
| <code>FILENAME_MAX</code> | Maximum size in bytes of the longest filename string that the implementation guarantees can be opened.                                                                                                        |
| <code>FOPEN_MAX</code>    | Maximum number of files which the implementation guarantees can be open simultaneously.                                                                                                                       |
| <code>_IOFBF</code>       | Input/output fully buffered.                                                                                                                                                                                  |
| <code>_IOLBF</code>       | Input/output line buffered.                                                                                                                                                                                   |
| <code>_IONBF</code>       | Input/output unbuffered.                                                                                                                                                                                      |
| <code>L_ctermid</code>    | Maximum size of character array to hold <code>ctermid()</code> output.                                                                                                                                        |
| <code>L_cuserid</code>    | Maximum size of character array to hold <code>cuserid()</code> output (Will no longer be supported by the X/Open-Standard in future.).                                                                        |
| <code>L_tmpnam</code>     | Maximum size of character array to hold <code>tmpnam()</code> output.                                                                                                                                         |
| <code>SEEK_CUR</code>     | Seek relative to current position.                                                                                                                                                                            |
| <code>SEEK_END</code>     | Seek relative to end-of-file.                                                                                                                                                                                 |
| <code>SEEK_SET</code>     | Seek relative to start-of-file.                                                                                                                                                                               |
| <code>TMP_MAX</code>      | Minimum number of unique filenames generated by <code>tmpnam()</code> .<br>Maximum number of times an application can call <code>tmpnam()</code> reliably. The value must be greater than or equal to 10.000. |

The following macro names are defined:

|                       |                                                                                    |
|-----------------------|------------------------------------------------------------------------------------|
| <code>EOF</code>      | Return value for end-of-file, defined as a negative integral constant.             |
| <code>NULL</code>     | Null pointer.                                                                      |
| <code>P_tmpdir</code> | Default directory prefix for <code>tmpnam()</code> , defined as a string constant. |
| <code>stderr</code>   | Standard error output stream, defined a data type pointer for <code>FILE</code> .  |
| <code>stdin</code>    | Standard input stream, defined a data type pointer for <code>FILE</code> .         |
| <code>stdout</code>   | Standard output stream, defined a data type pointer for <code>FILE</code> .        |

The following data types are defined by typedef:

|          |                                                                                                                                                                                 |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FILE     | A structure containing information about a file.                                                                                                                                |
| fpos_t   | Type containing all information needed to uniquely identify every position within a file.                                                                                       |
| fpos64_t | Type containing all information needed to uniquely identify every position within a file in which the maximum offset is determined by the range of values of an off64_t object. |
| va_list  | See stdarg.h                                                                                                                                                                    |
| size_t   | See stddef.h                                                                                                                                                                    |

The following names are declared as functions:

```
void clearerr(FILE *stream); (also defined as a macro)
char *ctermid(char *s);
char *cuserid(char *s); (To be withdrawn from the X/Open standard.)
int fclose(FILE *stream);
FILE *fdopen(int fildev, const char *mode);
int feof(FILE *stream); (also defined as a macro)
int ferror(FILE *stream); (also defined as a macro)
void lockfile(FILE *stream);
int fflush(FILE *stream);
int fgetc(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fgetpos64(FILE *stream, fpos64_t *pos);
char *fgets(char *s, int n, FILE *stream);
int fileno(FILE *stream); (also defined as a macro)
FILE *fopen(const char *filename, const char *mode);
FILE *fopen64(const char *filename, const char *mode);
int fprintf(FILE *stream, const char *format, ...);
int fputc(int c, FILE *stream);
int fputs(char *s, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
FILE *freopen64(const char *filename, const char *mode, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fseek(FILE *stream, long int offset, int whence);
int fseek64(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
int fsetpos64(FILE *stream, const fpos64_t *pos);
long int ftell(FILE *stream);
off64_t ftello64(FILE *stream);
void ftrylockfile(FILE *stream);
void funlockfile(FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
int getc(FILE *stream); (also defined as a macro)
int getc_unlocked (FILE *stream); (also defined as a macro)
int getchar(void); (also defined as a macro)
int getchar_unlocked(void); (also defined as a macro)
int getopt(int argc, char * const argv[], const char *optstring);
char *gets(char *s);
int getw(FILE *stream);
int pclose(FILE *stream);
void perror(const char *s);
FILE *popen(const char *command, const char *type);
int printf(const char *format, ...);
int putc(int c, FILE *stream); (also defined as a macro)
int fputc_unlocked(int c, FILE *stream); (also defined as a macro)
int putchar(int c); (also defined as a macro)
int fputc_unlocked(int c); (also defined as a macro)
int puts(const char *s);
int putw(int w, FILE *stream);
```

```
int remove(const char *path);
int rename(const char *old, const char *new);
void rewind(FILE *stream);
int scanf(const char *format, ...);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
char *tempnam(const char *dir, const char *pfx);
FILE *tmpfile(void);
FILE *tmpfile64(void);
char *tmpnam(char *s);
int ungetc(int c, FILE *stream);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vprintf(const char *format, va_list ap);
int vsprintf(char *s, const char *format, va_list ap);
```

The following external variables are defined:

```
extern char *optarg; (Will no longer be supported by the X/Open-Standard in future.)
extern int opterr; (Will no longer be supported by the X/Open-Standard in future.)
extern int optind; (Will no longer be supported by the X/Open-Standard in future.)
extern int optopt; (Will no longer be supported by the X/Open-Standard in future.)
```

## Notes

Inclusion of the `stdio.h` header may also make the symbols defined in `stddef.h` visible.

### *BS2000*

The following header files are included automatically together with `stdio.h` for compatibility with earlier versions of the C runtime systems:

```
stdio.common.h
stdio.bs21.h
stdio.bs22.h
iobuf.h
```

**See also** `clearerr()`, `ctermid()`, `cuserid()`, `fclose()`, `fdopen()`, `feof()`, `ferror()`, `fflush()`, `fgetc()`, `fgetpos()`, `fgets()`, `fileno()`, `fopen()`, `fputc()`, `fputs()`, `fread()`, `freopen()`, `fseek()`, `fsetpos()`, `ftell()`, `fwrite()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, `getopt()`, `gets()`, `getw()`, `getwchar()`, `getws()`, `pclose()`, `perror()`, `popen()`, `printf()`, `putc()`, `putc_unlocked()`, `putchar_unlocked()`, `puts()`, `putw()`, `putwchar()`, `remove()`, `rename()`, `rewind()`, `scanf()`, `setbuf()`, `setvbuf()`, `sscanf()`, `stdin`, `system()`, `tempnam()`, `tmpfile()`, `tmpnam()`, `ungetc()`, `vprintf()`, `stdarg.h`, `stddef.h`, `sys/types.h`.

## stdlib.h - standard library definitions

Syntax `#include <stdlib.h>`

Description The following constants are defined:

|                           |                                                                                    |
|---------------------------|------------------------------------------------------------------------------------|
| <code>EXIT_FAILURE</code> | Unsuccessful termination for <code>exit()</code> ; evaluates to a value $\neq 0$ . |
| <code>EXIT_SUCCESS</code> | Successful termination for <code>exit()</code> ; evaluates to the value 0.         |
| <code>NULL</code>         | Null pointer                                                                       |
| <code>RAND_MAX</code>     | Maximum for the return value <code>rand()</code> ; at least 32,767.                |
| <code>MB_CUR_MAX</code>   | Maximum number of bytes in a character specified by the current locale.            |

The following data types are defined by typedef:

|                      |                                                              |
|----------------------|--------------------------------------------------------------|
| <code>div_t</code>   | Structure type returned by the <code>div()</code> function.  |
| <code>ldiv_t</code>  | Structure type returned by the <code>ldiv()</code> function. |
| <code>size_t</code>  | See <code>stddef.h</code>                                    |
| <code>wchar_t</code> | See <code>stddef.h</code>                                    |

In addition, the following symbolic names and macros are defined as in `sys/wait.h`, for use in decoding the return value from `system()`:

`WNOHANG`  
`WUNTRACED`  
`WEXITSTATUS()`  
`WIFEXITED()`  
`WIFSIGNALED()`  
`WIFSTOPPED()`  
`WSTOPSIG()`  
`WTERMSIG()`

The following names are declared as functions:

```
void abort(void);
int abs(int i);
int atexit(void (*func)(void));
double atof(const char *str);
int atoi(const char *str);
long int atol(const char *str);
void *bsearch(const void *key, const void *base, size_t nel,
size_t width, int (*compar)(const void *,
const void *));
void *calloc(size_t nelem, size_t elsize);
div_t div(int numer, int denom);
double drand48(void);
double erand48(unsigned short int xsubi[3]);
void exit(int status);
void free(void *ptr);
char *getenv(const char name);
long int jrand48(unsigned short int xsubi[3]);
long int labs(long int j);
void lcong48(unsigned short int param[7]);
ldiv_t ldiv(long int numer, long int denom);
long int lrand48(void);
void *malloc(size_t size);
int mblen(const char *s, size_t n);
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
int mbtowlc(wchar_t *pwc, const char *s, size_t n);
long int mrand48(void);
long int nrand48(unsigned short int xsubi[3]);
int putenv(const char *str);
```

```
void qsort(void *base, size_t nel, size_t width,
int (*compar)(const void *, const void *));
int rand(void);
void *realloc(void *ptr, size_t size);
unsigned short int *seed48(unsigned short int seed16v[3]);
void setkey(const char *key);
void srand(unsigned int seed);
void srand48(long int seedval);
double strtod(const char *str, char **ptr);
long int strtol(const char *str, char **ptr, int base);
unsigned short int strtoul(const char *str, char **ptr, int base);
int system(const char *str);
int unlockpt(int n);
size_t wcstombs(const char *str, const wchar_t *pwcs, size_t n);
int wctomb(char *s, wchar_t wchar);
```

**Notes**      **Inclusion of the `stdlib.h` header may also make all the symbols in `stddef.h`, `limits.h`, `math.h` and `sys/wait.h` visible.**

**See also**    `abort()`, `abs()`, `atexit()`, `atof()`, `atoi()`, `atol()`, `bsearch()`, `calloc()`, `div()`, `drand48()`, `erand48()`, `exit()`, `free()`, `getenv()`, `jrand48()`, `labs()`, `lcong48()`, `ldiv()`, `lrand48()`, `malloc()`, `mblen()`, `mbstowcs()`, `mbtowc()`, `rand48()`, `putenv()`, `qsort()`, `rand()`, `realloc()`, `srand()`, `srand48()`, `strtod()`, `strtol()`, `strtoul()`, `wcstombs()`, `wctomb()`, `stddef.h`, `limits.h`, `math.h`, `sys/wait.h`.



## string.h - string operations

**Syntax**      `#include <string.h>`

**Description** `string.h` defines the following symbolic names:

|                     |                             |
|---------------------|-----------------------------|
| <code>NULL</code>   | Null pointer                |
| <code>size_t</code> | See <code>stddef.h</code> . |

The following names are declared as functions:

```
void *memcpy(void *s1, const void *s2, int c, size_t n);
void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, int c, size_t n);
void *strcpy(void *s1, const void *s2, int c, size_t n);
void *memmove(void *s1, const void *s2, int c, size_t n);
void *memset(void *s, int c, size_t n);
char *strcat(char *s1, const char *s2);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
char *strcpy(char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);
char *strerror(int errnum);
size_t strlen(const char *s);
char *strncat(char *s1, const char *s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
char *strtok_r(char *s1, const char *s2);
size_t strxfrm(char *s1, const char *s2, size_t n);
```

**Notes**      Inclusion of the `string.h` header may also make all the symbols defined in `stddef.h` visible.

**See also**    `memcpy()`, `memchr()`, `memcmp()`, `memcpy()`, `memmove()`, `memset()`, `strcat()`, `strchr()`, `strcmp()`, `strcoll()`, `strcpy()`, `strcspn()`, `strerror()`, `strlen()`, `strncat()`, `strncmp()`, `strncpy()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`, `strtok()`, `strtok_r()`, `strxfrm()`, `sys/types.h`.

## strings.h - string operations

**Syntax**      `#include <strings.h>`

**Description** The following functions are declared:

```
int      bcmp(const void *s1, const void *s2, size_t n);
void     bcopy(const void *s1, void *s2, size_t n);
void     bzero(void *s, size_t n);
int      ffs(int i);
char     *index(const char *s, int c);
char     *rindex(const char *s, int c);
int      strcasecmp(const char *s1, const char *s2);
int      strncasecmp(const char *s1, const char *s2, size_t n);
```

**See also**    `bcmp()`, `bcopy()`, `bzero()`, `ffs()`, `index()`, `rindex()`, `strcasecmp()`.

## stropts.h - STREAMS interface

Syntax `#include <stropts.h>`

Description The header `stropts.h` defines the following structures:

The `bandinfo` structure for the `I_FLUSHBAND` priority classes of the `ioctl()` function:

```
struct bandinfo {
    unsigned char bi_pri;
    int bi_flag;
};
```

The `strpeek` structure for the `I_PEEK` stream format of the `ioctl()` function:

```
struct strpeek {
    struct strbuf ctlbuf;
    struct strbuf databuf;
    long flags;
};
```

The `strbuf` structure for `putmsg` and `getmsg` system calls:

```
struct strbuf {
    int maxlen;                /* Maximum length of the buffer */
    int len;                   /* Data length */
    char *buf;                 /* Pointer to the buffer */
};
```

The `strfdinsert` structure for the `I_FDINSERT` stream format of the `ioctl()` function:

```
struct strfdinsert {
    struct strbuf ctlbuf;
    struct strbuf databuf;
    long flags;
    int fildes;
    int offset;
};
```

The `strioc1` structure for user formats of the `ioctl()` function which go downstream (`I_STR`):

```
struct strioc1 {
    int ic_cmd;                /* Command */
    int ic_timeout;           /* Timeout value */
    int ic_len;               /* Data length */
    char *ic_dp;              /* Pointer to the data */
};
```

The `strrecvfd` structure for the compatibility mode of the user. `E_OVERFLOW` is returned if `uid` or `gid` exceeds the `ushort` limit:

```
struct strrecvfd {
    int fd;
    o_uid_t uid;
```

```
    o_uid_t gid;
    char fill[8];
};
```

The structures `str_mlist` and `str_list` for `I_LIST` of the `ioctl()` function:

```
struct str_mlist {
    char l_name[FMNAMESZ+1];
};

struct str_list {
    int sl_nmods;
    struct str_mlist *sl_modlist;
};
```

The following macros are defined as the *request* argument for `ioctl()`:

- |                          |                                                                                                                                                                                                                                                                                                                                                           |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>I_PUSH</code>      | Pushes the STREAMS module right to the top of the current stream, just below the STREAM head.                                                                                                                                                                                                                                                             |
| <code>I_POP</code>       | Removes the STREAMS module from just below the STREAM head.                                                                                                                                                                                                                                                                                               |
| <code>I_LOOK</code>      | Calls the name of the module just below the STREAM head and stores it in a string. At least the following macro is defined as the <i>arg</i> argument:                                                                                                                                                                                                    |
|                          | <code>FMNAMESZ</code> Minimum size of the buffer (in bytes) that is addressed by the <i>arg</i> argument.                                                                                                                                                                                                                                                 |
| <code>I_FLUSH</code>     | Deletes all input or output queues (depending on the value of the <i>arg</i> argument). At least the following macros are defined as the <i>arg</i> argument:                                                                                                                                                                                             |
|                          | <code>FLUSHR</code> Delete the read queues.                                                                                                                                                                                                                                                                                                               |
|                          | <code>FLUSHW</code> Delete the write queues.                                                                                                                                                                                                                                                                                                              |
|                          | <code>FLUSHRW</code> Delete the read and write queues.                                                                                                                                                                                                                                                                                                    |
| <code>I_FLUSHBAND</code> | Deletes only the specified band.                                                                                                                                                                                                                                                                                                                          |
| <code>I_SETSIG</code>    | <p>Notifies the STREAM head that the process wants to have the <code>SIGPOLL</code> signal issued (see <code>signal()</code> and <code>sigset()</code>) if a specific event occurs within the context of the STREAM.</p> <p>The header file <code>stropts.h</code> defines the following values for <i>arg</i> if <code>I_SETSIG</code> is specified:</p> |

---

|                        |                                                                                                                                                                                                                                 |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>S_RDNORM</code>  | Normal message (priority class = 0) has arrived at the beginning of a read queue of the <code>STREAM</code> head.                                                                                                               |
| <code>S_RDBAND</code>  | A message (priority class not equal to 0) has arrived at the beginning of the read queue of the <code>STREAM</code> head.                                                                                                       |
| <code>S_INPUT</code>   | A message which does not have high priority has arrived at the beginning of a read queue of the <code>STREAM</code> head.                                                                                                       |
| <code>S_HIPRI</code>   | There is a message with high priority in a read queue of the <code>STREAM</code> head.                                                                                                                                          |
| <code>S_OUTPUT</code>  | The write queue for regular data (priority class 0) just below the <code>STREAM</code> head is no longer full. This informs the process that there is room in the queue for sending (or writing) regular data.                  |
| <code>S_WRNORM</code>  | As <code>S_OUTPUT</code> .                                                                                                                                                                                                      |
| <code>S_WRBAND</code>  | The write queue for a priority class not equal to 0 just below the <code>STREAM</code> head is no longer full.                                                                                                                  |
| <code>S_MSG</code>     | A <code>STREAMS</code> signal message with the <code>SIGPOLL</code> signal has reached the beginning of the read queue of the <code>STREAM</code> head.                                                                         |
| <code>S_ERROR</code>   | A message concerning an error condition has reached the <code>STREAM</code> head.                                                                                                                                               |
| <code>S_HANGUP</code>  | The message concerning a hang-up has reached the <code>STREAM</code> head.                                                                                                                                                      |
| <code>S_BANDURG</code> | If this is used together with <code>S_RDBAND</code> , then <code>SIGURG</code> is generated instead of <code>SIGPOLL</code> if a message with priority reaches the beginning of the read queue of the <code>STREAM</code> head. |
| <code>I_GETSIG</code>  | Returns the events for which the calling process is currently to receive a <code>SIGPOLL</code> signal.                                                                                                                         |
| <code>I_FIND</code>    | Compares the names of all modules currently in the <code>STREAM</code> with the name to which <i>arg</i> points.                                                                                                                |
| <code>I_PEEK</code>    | A process can call up the data in first message of the read queue of the <code>STREAM</code> head without the message being removed from the queue. The following macro is defined as the <i>arg</i> argument:                  |
| <code>RS_HIPRI</code>  | Only messages with a high priority are sought.                                                                                                                                                                                  |

|                    |                                                                                                                                                                                               |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>I_SRDOPT</b>    | <b>Sets the read mode. The following macros are defined as the <i>arg</i> argument:</b>                                                                                                       |
|                    | <b>RNORM</b> <b>Byte-STREAM mode (default).</b>                                                                                                                                               |
|                    | <b>RMSGD</b> <b>Message-discard mode.</b>                                                                                                                                                     |
|                    | <b>RMSGN</b> <b>Message-nondiscard mode.</b>                                                                                                                                                  |
|                    | <b>RPROTNORM</b> <b>read() with EBADMSG fails if there is a message with a control section at the beginning of the read queue of the STREAM head.</b>                                         |
|                    | <b>RPROTDAT</b> <b>The control section of a message is delivered as data if a process issues read().</b>                                                                                      |
|                    | <b>RPROTDIS</b> <b>The control section of a message is discarded but the data is delivered if a process issues read().</b>                                                                    |
| <b>I_GRDOPT</b>    | <b>Returns the current setting of the read mode.</b>                                                                                                                                          |
| <b>I_NREAD</b>     | <b>Counts the number of data bytes in data blocks in the first message of the read queue of the STREAM head.</b>                                                                              |
| <b>I_FDINSERT</b>  | <b>Generates messages from the specified buffer(s), adds information via a different STREAM and sends the message downstream.</b>                                                             |
| <b>I_STR</b>       | <b>Generates an internal STREAMS <code>ioctl()</code> message and sends it downstream.</b>                                                                                                    |
| <b>I_SWROPT</b>    | <b>Sets the write mode. The following macro is defined as the <i>arg</i> argument:</b>                                                                                                        |
|                    | <b>SNDZERO</b> <b>Sends a message of length 0 downstream if a <code>write()</code> with 0 bytes occurs.</b>                                                                                   |
| <b>I_GWROPT</b>    | <b>Returns the current setting of the write mode.</b>                                                                                                                                         |
| <b>I_SENDFD</b>    | <b>Requests that the STREAM associated with <i>fd</i> sends a message with a file pointer to the STREAM head at the other end of a STREAMS pipe.</b>                                          |
| <b>I_RECVFD</b>    | <b>Calls the file descriptor associated with the message that was sent by an <code>I_SENDFD</code> <code>ioctl()</code> via a STREAMS pipe.</b>                                               |
| <b>I_LIST</b>      | <b>The process can list all module names of the STREAM (up to and including the highest driver name).</b>                                                                                     |
| <b>I_ATMARK</b>    | <b>The process can see whether the current message in the read queue of the STREAM head was “marked” downstream by a module. The following macros are defined as the <i>arg</i> argument:</b> |
|                    | <b>ANYMARK</b> <b>Checks whether the message is marked.</b>                                                                                                                                   |
|                    | <b>LASTMARK</b> <b>Checks whether the message is the last one marked in the queue.</b>                                                                                                        |
| <b>I_CKBAND</b>    | <b>Checks whether there is a message of a particular priority class in the read queue of the STREAM head.</b>                                                                                 |
| <b>I_GETBAND</b>   | <b>Returns the priority class of the first message in the read queue of the STREAM head.</b>                                                                                                  |
| <b>I_CANPUT</b>    | <b>Checks whether a specific priority class can be written to.</b>                                                                                                                            |
| <b>I_SETCLTIME</b> | <b>The process can specify how long the delay for the STREAM head is if a STREAM is terminated when there is still data in the write queues.</b>                                              |

`I_GETCLTIME` Returns the delay for the termination.

`I_LINK` Links two STREAMs.

`I_UNLINK` The link between the two STREAMs is canceled. The header defines the following value for *all*:

`MUXID_ALL` The link is canceled for all STREAMs that are associated with the STREAM assigned to *fd*.

`I_PLINK` Links two STREAMs with an existing association.

`I_PUNLINK` Disconnects the two STREAMs that were linked with a persistent association.

For `getmsg()`, `getpmsg()`, `putmsg()` and `putpmsg()` the following macros are defined:

`MSG_ANY` Receives any message.

`MSG_BAND` Receives message of a specific priority class.

`MSG_HIPRI` Sends/receives messages with high priority.

`MORECTL` More control data is left in the message.

`MOREDATA` More data is left in the message.

`stropts.h` can make all symbols in `unistd.h` visible. The following functions are declared as functions in the `stropts.h` header:

```
int  isastream(int fildes);

int  getmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,
           int *flagsp);

int  getpmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,
            int *bandp, int *flagsp);

int  ioctl(int fildes, int request, ... );

int  putmsg(int fd, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int flags);

int  putpmsg(int fd, const struct strbuf *ctlptr,
             const struct strbuf *dataptr, int band, int flags);

int  fattach(int fildes, const char *path);

int  fdetach(const char *path);
```

**See also** `close()`, `fcntl()`, `getmsg()`, `ioctl()`, `open()`, `pipe()`, `read()`, `poll()`, `putmsg()`, `signal()`, `sigset()`, `write()`.

## sys/ipc.h - interprocess communication access structure

Syntax `#include <sys/ipc.h>`

Description The `sys/ipc.h` header is used by three mechanisms for interprocess communication (IPC): messages, semaphores and shared memory. All use a common structure type, `ipc_perm`, to pass information used in determining permission to perform an IPC operation.

The structure `ipc_perm` contains the following members:

|                     |                   |                       |
|---------------------|-------------------|-----------------------|
| <code>uid_t</code>  | <code>uid</code>  | Owner's user ID       |
| <code>gid_t</code>  | <code>gid</code>  | Owner's group ID      |
| <code>uid_t</code>  | <code>cuid</code> | Creator's user ID     |
| <code>gid_t</code>  | <code>cgid</code> | Creator's group ID    |
| <code>mode_t</code> | <code>mode</code> | Read/write permission |

Definitions are given for the following constants:

Mode bits:

|                         |                                    |
|-------------------------|------------------------------------|
| <code>IPC_CREAT</code>  | Create entry if key does not exist |
| <code>IPC_EXCL</code>   | Fail if key exists                 |
| <code>IPC_NOWAIT</code> | Error if request must wait         |

Keys:

|                          |             |
|--------------------------|-------------|
| <code>IPC_PRIVATE</code> | Private key |
|--------------------------|-------------|

Control commands:

|                       |                   |
|-----------------------|-------------------|
| <code>IPC_RMID</code> | Remove identifier |
| <code>IPC_SET</code>  | Set options       |
| <code>IPC_STAT</code> | Get options       |



## syslog.h - definitions for logging system error messages

Syntax `#include <syslog.h>`

`syslog.h` defines the following symbolic constants. ORing these constants in any combination produces the `logopt` option of `openlog()`:

|                         |                                                                             |
|-------------------------|-----------------------------------------------------------------------------|
| <code>LOG_PID</code>    | Logs the process ID with each message.                                      |
| <code>LOG_CONS</code>   | Outputs error messages to the system console.                               |
| <code>LOG_NDELAY</code> | Opens the connection to <code>syslog</code> immediately.                    |
| <code>LOG_ODELAY</code> | Delays the opening of the connection until <code>syslog()</code> is called. |
| <code>LOG_NOWAIT</code> | Does not wait for child processes.                                          |

The following symbolic constants are defined as values for the *facility* parameter of `openlog()`:

|                         |                                                               |
|-------------------------|---------------------------------------------------------------|
| <code>LOG_KERN</code>   | Reserved for system messages.                                 |
| <code>LOG_USER</code>   | Messages generated by user processes.                         |
| <code>LOG_MAIL</code>   | Reserved for messages of the mail system.                     |
| <code>LOG_NEWS</code>   | Reserved for messages of the USENET network news system.      |
| <code>LOG_UUCP</code>   | Reserved for messages of the UUCP system.                     |
| <code>LOG_DAEMON</code> | Reserved for messages of the system daemons.                  |
| <code>LOG_AUTH</code>   | Reserved for messages of the authorization system ("daemon"). |
| <code>LOG_CRON</code>   | Reserved for messages of the "clock daemon".                  |
| <code>LOG_LPR</code>    | Reserved for messages of the printer system.                  |
| <code>LOG_LOCAL0</code> | Reserved for local use.                                       |
| <code>LOG_LOCAL1</code> | Reserved for local use.                                       |
| <code>LOG_LOCAL2</code> | Reserved for local use.                                       |
| <code>LOG_LOCAL3</code> | Reserved for local use.                                       |
| <code>LOG_LOCAL4</code> | Reserved for local use.                                       |
| <code>LOG_LOCAL5</code> | Reserved for local use.                                       |
| <code>LOG_LOCAL6</code> | Reserved for local use.                                       |
| <code>LOG_LOCAL7</code> | Reserved for local use.                                       |

The following macros serve to represent the *maskpri* parameter of `setlogmask()`.

`LOG_MASK(pri)` Computes the mask for a given priority *pri*.

`LOG_UPTO(pri)` Outputs the mask for all priorities up to *pri*.

The following macros are defined as values of the *priority* parameter of `syslog()`:

|                          |                                                                                |
|--------------------------|--------------------------------------------------------------------------------|
| <code>LOG_EMERG</code>   | "Panic" condition reported to all user processes.                              |
| <code>LOG_ALERT</code>   | Condition that should be corrected immediately (e.g. damaged system database). |
| <code>LOG_CRIT</code>    | Critical condition (e.g. hard disk error).                                     |
| <code>LOG_ERR</code>     | Error message.                                                                 |
| <code>LOG_WARNING</code> | Warning message.                                                               |
| <code>LOG_NOTICE</code>  | Condition which requires special steps.                                        |
| <code>LOG_INFO</code>    | Information message.                                                           |
| <code>LOG_DEBUG</code>   | Message with information that is used during program debugging.                |

The following functions are defined:

```
void  closelog(void);
void  openlog(const char *id, int logopt, int facility);
int   setlogmask(int maskpri);
void  syslog(int priority, const char *format, ...);
```

**See also** `closelog()`.

## sys/mman.h - definitions for memory management

**Syntax**      `#include <sys/mman.h>`

**Description** The following protection options are available:

|                         |                          |
|-------------------------|--------------------------|
| <code>PROT_READ</code>  | Page can be read.        |
| <code>PROT_WRITE</code> | Page can be written.     |
| <code>PROT_EXEC</code>  | Page can be executed.    |
| <code>PROT_NONE</code>  | Page cannot be accessed. |

The following `flag` options are defined:

|                          |                                |
|--------------------------|--------------------------------|
| <code>MAP_SHARED</code>  | Make changes shareable.        |
| <code>MAP_PRIVATE</code> | Keep changes private.          |
| <code>MAP_FIXED</code>   | Interpret <i>addr</i> exactly. |

The following flags (bit patterns) are defined for `msync()`:

|                            |                                      |
|----------------------------|--------------------------------------|
| <code>MS_ASYNC</code>      | Perform asynchronous write accesses. |
| <code>MS_SYNC</code>       | Perform synchronous write accesses.  |
| <code>MS_INVALIDATE</code> | Invalidate mappings (links).         |

The data types `size_t` and `off_t` are defined and described in `sys/types.h`.

The following functions are declared in `sys/mman.h`:

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t off);
int mprotect(void *addr, size_t len, int prot);
int msync(void *addr, size_t len, int flags);
int munmap(void *addr, size_t len);
```

**See also**      `mmap()`, `mprotect()`, `msync()`, `munmap()`.

## sys/msg.h - message queue structures

**Syntax**      `#include <sys/msg.h>`

**Description** The `sys/msg.h` header defines the following constants and members of the structure `msqid_ds`.

**Message processing flag:**

`MSG_NOERROR`      No error if big message

The structure `msqid_ds` contains the following members:

|                              |                         |                                          |
|------------------------------|-------------------------|------------------------------------------|
| <code>struct ipc_perm</code> | <code>msg_perm</code>   | Operation permission structure           |
| <code>unsigned short</code>  | <code>msg_qnum</code>   | Number of messages currently on queue    |
| <code>unsigned short</code>  | <code>msg_qbytes</code> | Maximum number of bytes allowed on queue |
| <code>pid_t</code>           | <code>msg_lspid</code>  | Process ID of last <code>msgsnd()</code> |
| <code>pid_t</code>           | <code>msg_lrpid</code>  | Process ID of last <code>msgrcv()</code> |
| <code>time_t</code>          | <code>msg_stime</code>  | Time of last <code>msgsnd()</code>       |
| <code>time_t</code>          | <code>msg_rtime</code>  | Time of last <code>msgrcv()</code>       |
| <code>time_t</code>          | <code>msg_ctime</code>  | Time of last change                      |

The following names are declared as functions

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

```
int msgget(key_t key, int msgflg);
```

```
int msgrcv(int msqid, void *msgq, size_t msgsz, long int msgtyp,
           int msgflg);
```

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

In addition, all of the symbols from `sys/ipc.h` will be defined when `sys/msg.h` is included in the source text.

**See also**      `msgctl()`, `msgget()`, `msgrcv()`, `msgsnd()`, `sys/types.h`.

## sys/resource.h - definitions of operations for XSI resources

Syntax `#include <sys/resource.h>`

Description `sys/resource.h` defines the following symbolic constants as possible values for the *which* argument of the `getpriority()` and `setpriority()` functions:

`PRIO_PROCESS`        Identifies *who* as the process ID.  
`PRIO_PGRP`            Identifies *who* as the process group ID.  
`PRIO_USER`            Identifies *who* as the user ID.

The following symbolic constant is defined as follows:

`RLIM_INFINITY`        The value of `rlim_t` is infinity.  
`RLIM64_INFINITY`      The value of `rlim64_t` is infinity.  
`RLIM_SAVED_MAX`       The value of `rlim_t` is infinity.  
`RLIM64_SAVED_MAX`    The value of `rlim64_t` is infinity.  
`RLIM_SAVED_CUR`       Current value of the `rlim_t` type.  
`RLIM64_SAVED_CUR`    Current value of the `rlim64_t` type.

`RLIM64_SAVED_MAX` and `RLIM64_SAVED_CUR` need not differ from `RLIM64_INFINITY` in implementations in which all resource limits can be represented by an object of `rlim64_t` type.

The following symbolic constants are possible values for the *who* argument of the `getrusage()` function:

`RUSAGE_SELF`         Outputs information on the current process.  
`RUSAGE_CHILDREN`     Outputs information on the child processes of the current process.

`rlim_t` is defined by typedef as an unsigned, integral data type used for limits:

```
struct rlimit {
    rlim_t rlim_cur;           /* Current limit */
    rlim_t rlim_max;         /* Maximum value for rlim_cur */
};
```

`rlim64_t` is defined by typedef as an extended, unsigned, integral data type that can represent every non-negative number in the range of values of an `off64_t` object and that is used for limits:

```
struct rlimit64 {
    rlim64_t rlim_cur;           /* Current (soft) limit */
    rlim64_t rlim_max;         /* Maximum value for
rlim_cur*/
```

`rusage` is defined as follows:

```
struct rusage {
    struct timeval ru_utime;    /* Time used in user mode */
    struct timeval ru_stime;    /* Time used in system mode */
    long ru_maxrss;
#define ru_first      ru_ixrss
    long ru_ixrss;             /* Size of shareable memory */
    long ru_idrss;             /* Size of non-shared data */
    long ru_isrss;             /* Size of non-shared stack */
    long ru_minflt;           /* Page error without I/O */
    long ru_majflt;           /* Page error with I/O */
    long ru_nswap;            /* swaps */
    long ru_inblock;          /* Block input operations */
    long ru_oublock;          /* Block output operations */
    long ru_msgsnd;           /* Messages sent */
    long ru_msgrcv;           /* Messages received */
    long ru_nsignals;         /* Signals received */
    long ru_nvcsw;            /* Voluntary change of context */
    long ru_nivcsw;           /* Involuntary change of context */
    long ru_totcsw;           /* Total no. of context changes */
/* pyramid additions */
    long ru_zerofill;         /* Pages to be zero-padded */
    long ru_pffincr;          /* No. of RSS increases */
    long ru_pffdecr;          /* No. of RSS decreases */
    long ru_syscall;          /* Number of system calls */
    long ru_lread;            /* Read calls of the system */
    long ru_lwrite;           /* Write calls of the system */
    long ru_phread;           /* Raw read accesses */
    long ru_phwrite;          /* Raw write accesses */
#define ru_last      ru_write
    long ru_spare[5];         /* Round up to 32-bit values */
};
```

The `timeval` structure is described in `sys/time.h`.

The following symbolic constants are possible values of the *resource* argument of the `getrlimit()` and `setrlimit()` functions:

|                            |                                                                  |
|----------------------------|------------------------------------------------------------------|
| <code>RLIMIT_CORE</code>   | Maximum size of a core dump file.                                |
| <code>RLIMIT_CPU</code>    | Maximum CPU time that can be used by a process.                  |
| <code>RLIMIT_DATA</code>   | Maximum size of the heap memory of a process.                    |
| <code>RLIMIT_FSIZE</code>  | Maximum length of a file.                                        |
| <code>RLIMIT_NOFILE</code> | Maximum number of open file descriptors that a process can have. |
| <code>RLIMIT_STACK</code>  | Maximum size of the process stack.                               |
| <code>RLIMIT_AS</code>     | Maximum length of the address area of a process.                 |

The following functions are defined:

```
int  getpriority(int which, id_t who);
int  getrlimit(int resource, struct rlimit *rlp);
int  getrlimit64(int resource, struct rlimit64 *rlp);
int  getrusage(int who, struct rusage *r_usage);
int  setpriority(int which, id_t who, int priority);
int  setrlimit(int resource, const struct rlimit *rlp);
int  setrlimit64(int resource, const struct rlimit64 *rlp);
```

The header file `sys/resource.h` can also make all contents of `sys/time.h` visible.

**See also** `getpriority()`, `getrusage()`, `getrlimit()`.

## sys/sem.h - semaphore structures

Syntax `#include <sys/sem.h>`

Description The `sys/sem.h` header defines the following constants and structures.

Semaphore operation flags:

`SEM_UNDO`      Automatic release of semaphores at process exit

Command definitions for the function `semctl()`:

|                      |                                             |
|----------------------|---------------------------------------------|
| <code>GETNCNT</code> | <b>Get</b> <code>semcnt</code>              |
| <code>GETPID</code>  | <b>Get</b> <code>sempid</code>              |
| <code>GETVAL</code>  | <b>Get</b> <code>semval</code>              |
| <code>GETALL</code>  | <b>Get all cases of</b> <code>semval</code> |
| <code>GETZCNT</code> | <b>Get</b> <code>semzcnt</code>             |
| <code>SETVAL</code>  | <b>Set</b> <code>semval</code>              |
| <code>SETALL</code>  | <b>Set all cases of</b> <code>semval</code> |

The structure `semid_ds` contains the following members:

|                              |                        |                                            |
|------------------------------|------------------------|--------------------------------------------|
| <code>struct ipc_perm</code> | <code>sem_perm</code>  | Operation permission structure             |
| <code>unsigned short</code>  | <code>sem_nsems</code> | Number of semaphores in set                |
| <code>time_t</code>          | <code>sem_otime</code> | Last execution of <code>semop()</code>     |
| <code>time_t</code>          | <code>sem_ctime</code> | Last time changed by <code>semctl()</code> |

The `pid_t`, `time_t` and `size_t` types are defined as described in `sys/types.h`.

`sem_nsems` is the number of semaphores in a set; the individual semaphores in a set are numbered sequentially from 0 to `sem_nsems-1`, and `sem_num` is the semaphore number.

A semaphore is represented by an anonymous structure containing the following members:

|                             |                      |                                                                                          |
|-----------------------------|----------------------|------------------------------------------------------------------------------------------|
| <code>unsigned short</code> | <code>semval</code>  | Semaphore value                                                                          |
| <code>pid_t</code>          | <code>sempid</code>  | Process ID of last operation                                                             |
| <code>unsigned short</code> | <code>semcnt</code>  | Number of processes waiting for <code>semval</code> to become greater than current value |
| <code>unsigned short</code> | <code>semzcnt</code> | Number of processes waiting for <code>semval</code> to become 0                          |



The structure *sembuf* contains the following members:

|                |         |                     |
|----------------|---------|---------------------|
| unsigned short | sem_num | Semaphore number    |
| short          | sem_op  | Semaphore operation |
| short          | sem_flg | Operation flags     |

The following names are declared as functions:

```
int semctl(int semid, int semnum, int cmd, ...);  
int semget(key_t key, int nsems, int semflg);  
int semop(int semid, struct sembuf *sops, size_t nsops);
```

**In addition, all of the symbols from `sys/ipc.h` will be defined when `sys/mem.h` is included in the source text.**

**See also** `semctl()`, `semget()`, `semop()`, `sys/types.h`.

## sys/shm.h - shared memory definitions

**Syntax**     **#include** <sys/shm.h>

**Description** The `sys/shm.h` header defines the following constants and a structure.

**Flags for operations:**

|            |                                             |
|------------|---------------------------------------------|
| SHM_RDONLY | Attach as read-only (otherwise: read-write) |
| SHMLBA     | Segment low boundary address multiple       |
| SHM_RND    | Round attach address to SHMLBA              |

The following data type is defined through typedef:

|          |                                                                                                                                                                                                                        |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| shmatt_t | Unsigned integer used for the number of currently attached areas, <code>shm_nattch</code> , which must be able to store values that at least correspond to the value range of the type <code>unsigned integer</code> . |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

The `pid_t`, `time_t`, `key_t` and `size_t` types are defined as described in `sys/types.h`.

The structure `shmids` contains the following members:

|                 |            |                                              |
|-----------------|------------|----------------------------------------------|
| struct ipc_perm | shm_perm   | Permission structure                         |
| int             | shm_segsz  | Size of segment in bytes                     |
| pid_t           | shm_lpid   | Process ID of last shared memory operation   |
| pid_t           | shm_cpid   | Process ID of creator                        |
| shmatt_t        | shm_nattch | Number of current attaches                   |
| time_t          | shm_atime  | Time of last <code>shmat()</code>            |
| time_t          | shm_dtime  | Time of last <code>shmdt()</code>            |
| time_t          | shm_ctime  | Time of last change by <code>shmctl()</code> |

The following names are declared as functions:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmctl(int shmid, int cmd, struct shmids *buf);
int shmdt(const void *shmaddr);
int shmget(key_t key, size_t size, int shmflg);
```

In addition, all of the symbols from `sys/ipc.h` will be defined when `sys/shm.h` is included in the source text.

**See also**     `shmat()`, `shmctl()`, `shmdt()`, `shmget()`, `sys/types.h`.

## sys/stat.h - data for file status

Syntax `#include <sys/stat.h>`

Description The `sys/stat.h` header defines the structure of the data returned by the functions `stat()` and `fstat()`.

The structure `stat` contains at least the following members:

| Data type               | Structure members      | Meaning                                        |
|-------------------------|------------------------|------------------------------------------------|
| <code>dev_t</code>      | <code>st_dev</code>    | ID of device containing file                   |
| <code>ino_t</code>      | <code>st_ino</code>    | File serial number                             |
| <code>ino64_t</code>    | <code>st_ino</code>    | File serial number                             |
| <code>mode_t</code>     | <code>st_mode</code>   | Mode of file (see below)                       |
| <code>nlink_t</code>    | <code>st_nlink</code>  | Number of links to the file                    |
| <code>uid_t</code>      | <code>st_uid</code>    | User ID of owner                               |
| <code>gid_t</code>      | <code>st_gid</code>    | Group ID of owner                              |
| <code>dev_t</code>      | <code>st_rdev</code>   | Device ID (if character or block special file) |
| <code>off_t</code>      | <code>st_size</code>   | File size in bytes (for regular files only)    |
| <code>off64_t</code>    | <code>st_size</code>   | File size in bytes                             |
| <code>time_t</code>     | <code>st_atime</code>  | Time of last access                            |
| <code>time_t</code>     | <code>st_mtime</code>  | Time of last modification                      |
| <code>time_t</code>     | <code>st_ctime</code>  | Time of last status change                     |
| <code>blkcnt_t</code>   | <code>st_blocks</code> | Number of blocks allocated for the object      |
| <code>blkcnt64_t</code> | <code>st_blocks</code> | Number of blocks allocated for the object      |

The file serial number and device ID of the device containing the file uniquely identify the file within the system. The `dev_t`, `ino_t`, `nlink_t`, `uid_t`, `gid_t`, `off_t` and `time_t` types are defined as described in `sys/types.h`. Times are given in seconds since the beginning of the Epoch.

The following symbolic names are defined for the values of `st_mode`:

| Symbolic names        | Meaning                                     |
|-----------------------|---------------------------------------------|
| <code>S_IFMT</code>   | Type of file                                |
| <code>S_IFBLK</code>  | Block special file                          |
| <code>S_IFCHR</code>  | Character special file                      |
| <code>S_IFDIR</code>  | Directory                                   |
| <code>S_IFIFO</code>  | FIFO special file                           |
| <code>S_IFREG</code>  | Regular file                                |
| <code>S_IFSOCK</code> | Socket                                      |
| <code>S_IRWXU</code>  | Access permissions for owner                |
| <code>S_IRUSR</code>  | Read permission for owner                   |
| <code>S_IWUSR</code>  | Write permission for owner                  |
| <code>S_IXUSR</code>  | Execute/Search permission for owner         |
| <code>S_IRWXG</code>  | Access permission for group                 |
| <code>S_IRGRP</code>  | Read permission for group                   |
| <code>S_IWGRP</code>  | Write permission for group                  |
| <code>S_IXGRP</code>  | Execute/Search permission for group         |
| <code>S_IRWXO</code>  | Access permissions for others               |
| <code>S_IROTH</code>  | Read permission for others                  |
| <code>S_IWOTH</code>  | Write permission for others                 |
| <code>S_IXOTH</code>  | Execute/Search permission for others        |
| <code>S_ISUID</code>  | Set-User-ID on execution (s-bit for owner)  |
| <code>S_ISGID</code>  | Set-Group-ID on execution (s-bit for group) |

The bits defined by `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRGRP`, `S_IWGRP`, `S_IXGRP`, `S_IROTH`, `S_IWOTH`, `S_IXOTH`, `S_ISUID` and `S_ISGID` are unique.

`S_IRWXU` is the bit-wise OR of `S_IRUSR`, `S_IWUSR` and `S_IXUSR`.

`S_IRWXG` is the bit-wise OR of `S_IRGRP`, `S_IWGRP` and `S_IXGRP`.

The following macros for testing file types determine whether the addressed file matches the expected type. *m* corresponds to the value of the `st_mode` structure component from the `stat` structure. The macros evaluate to a non-zero value if the test is true.

`S_ISBLK(m)` Test for a block special file  
`S_ISCHR(m)` Test for a character special file  
`S_ISDIR(m)` Test for a directory  
`S_ISFIFO(m)` Test for a FIFO special file  
`S_ISREG(m)` Test for a regular file

The following names are declared as functions:

```
int chmod(const char *path, mode_t mode);
int fstat(int fildes, struct stat *buf);
int mkdir(const char *path, mode_t mode);
int mkfifo(const char *path, mode_t mode);
mode_t umask(mode_t cmask);

int fstat64(int fildes, struct stat64 *buf);
int lstat64(const char *path, struct stat64 *buf);
int stat64(const char *path, struct stat64 *buf);
```

Notes Use of the macros is recommended for determining the type of a file.

## sys/statvfs.h - structure of VFS file system information

Syntax `#include <sys/statvfs.h>`

Description `sys/statvfs.h` defines the `statvfs` structure:

```
typedef struct statvfs {
    unsigned long f_bsize;      /* Block size of the file system */
    unsigned long f_frsize;    /* Fragment size */
    unsigned long f_blocks;    /* # blocks in file system with size f_frsize */
    unsigned long f_bfree;     /* # free blocks with size f_frsize */
    unsigned long f_bavail;    /* # available free blocks for non-superusers */
    unsigned long f_files;     /* # file nodes (inodes) */
    unsigned long f_ffree;     /* # free file nodes (inodes) */
    unsigned long f_favail;    /* # available free inodes for non-superusers */
    unsigned long f_fsid;      /* File system ID (number) */
    char f_basetype[FSYPSZ];   /* Name of destination file system, null-terminated
    */
    unsigned long f_flag;      /* Bit mask of the options of f_flag */
    unsigned long f_namemax;   /* Maximum length of file names */
    char f_fstr[32];          /* File-system-specific string */
    unsigned long f_filler[16]; /* Reserved for future versions */
} statvfs_t;
```

`sys/statvfs.h` defines the `statvfs64` structure:

```
typedef struct statvfs64 {
    unsigned long f_bsize;      /* Block size of the file system */
    unsigned long f_frsize;    /* Fragment size */
    fsblkcnt64_t f_blocks;     /* # blocks in file system with size f_frsize */
    fsblkcnt64_t f_bfree;     /* # free blocks with size f_frsize */
    fsblkcnt64_t f_bavail;    /* # available free blocks for non-superusers */
    fsfilcnt64_t f_files;     /* # file nodes (inodes) */
    fsfilcnt64_t f_ffree;     /* # free file nodes (inodes) */
    fsfilcnt64_t f_favail;    /* # available free inodes for non-superusers */
    unsigned long f_fsid;      /* File system ID (number) */
    char f_basetype[FSYPSZ];   /* Name of destination file system, null-terminated
    */
    unsigned long f_flag;      /* Bit mask of the options of f_flag */
    unsigned long f_namemax;   /* Maximum length of file names */
    char f_fstr[32];          /* File-system-specific string */
    unsigned long f_filler[16]; /* Reserved for future versions */
} statvfs64_t;
```

The following options are defined for the `f_flag` component:

|                        |                                           |
|------------------------|-------------------------------------------|
| <code>ST_RDONLY</code> | Readable file system only                 |
| <code>ST_NOSUID</code> | setgid/setuid semantics are not supported |

The following functions are defined in `sys/statvfs.h`:

```
int statvfs(const char *path, struct statvfs64 *buf);
int statvfs64(const char *path, struct statvfs *buf);
int fstatvfs(int fildes, struct statvfs *buf);
int fstatvfs64(int fildes, struct statvfs64 *buf);
```

**See also** `fstatvfs()`, `statvfs()`.

## sys/time.h - time types

Syntax `#include <sys/time.h>`

Description `sys/time.h` defines the `timeval` structure:

```
struct timeval {
    long tv_sec;        /* Seconds since January 1, 1970 */
    long tv_usec;      /* Microseconds */
};
```

`sys/time.h` also defines the `itimerval` structure:

```
struct itimerval {
    struct timeval it_interval; /* Timer interval */
    struct timeval it_value;    /* Current value */
};
```

`sys/time.h` also defines the `fd_set` structure, which defines a bit mask with descriptions for opening files:

```
typedef struct fd_set {
    fd_mask fds_bits[howmany(FD_SETSIZE,NFDBITS)];
} fd_set;
```

`sys/time.h` defines the following values for the *which* argument of `getitimer()` and `setitimer()`:

|                |                                                                                          |
|----------------|------------------------------------------------------------------------------------------|
| ITIMER_REAL    | Decrements in real time.                                                                 |
| ITIMER_VIRTUAL | Decrements in the virtual processor time.                                                |
| ITIMER_PROF    | Decrements both in virtual process time if the system is running because of the process. |

The following macros are defined:

```
void FD_CLR(int fd, fd_set *fdset)
    Removes fd from the file descriptor set fdset.

int FD_ISSET(int fd, fd_set *fdset)
    Returns a non-zero value if fd is an element from the file descriptor set fdset; otherwise returns the value zero.

void FD_SET(int fd, fd_set *fdset)
    Inserts the file descriptor fd into the descriptor set fdset.
```



```
void FD_ZERO(fd_set *fdset)
```

Initializes a descriptor set *fdset* with the null set.

```
FD_SETSIZE
```

Maximum number of file descriptors in an *fd\_set* structure.

The following functions are defined:

```
int getitimer(int which, struct itimerval *value);
```

```
int setitimer(int which, const struct itimerval *value,  
             struct itimerval *ovalue);
```

```
int gettimeofday(struct timeval *tp, void *tzp);
```

```
int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *errorfds,  
          struct timeval *timeout);
```

```
int utimes(const char *path, const struct timeval times[2]);
```

**See also** `getitimer()`, `gettime()`, `select()`, `setitimer()`, `utimes()`.

## sys/timeb.h - additional definitions for date and time

**Syntax**     **#include <sys/timeb.h>**

**Description** `sys/timeb.h` defines the `timeb` structure:

```
struct timeb {
time_t time;           /* Current time in sec. since 00:00:00 1.1.1970 */
unsigned short millitm; /* Time in milliseconds */
short timezone;       /* Local time zone, In minutes west of Greenwich */
short dstflag;        /* Summer time variable. >0 for summer time,
                       0 for winter time , <0 if no information */
};
```

The data type `time_t` is described in `sys/types.h`.

The following function is defined in `sys/timeb.h`:

```
int ftime(struct timeb *tp);
```

**See also**     `ftime()`, `time.h`.

## sys/times.h - structure for file times

**Syntax**     **#include <sys/times.h>**

**Description** The `sys/time.h` header defines the `struct tms` structure, which includes the following members:

|                      |                         |                                               |
|----------------------|-------------------------|-----------------------------------------------|
| <code>clock_t</code> | <code>tms_utime</code>  | User CPU time                                 |
| <code>clock_t</code> | <code>tms_stime</code>  | System CPU time                               |
| <code>clock_t</code> | <code>tms_cutime</code> | User CPU time of terminated child processes   |
| <code>clock_t</code> | <code>tms_cstime</code> | System CPU time of terminated child processes |

The `clock_t` type is defined as described in `sys/types.h`.

The following name is declared as a function:

```
clock_t times(struct tms *buffer);
```

**See also**     `times()`, `sys/types.h`.

## sys/types.h - data types

Syntax `#include <sys/types.h>`

Description `sys/types.h` defines data types and includes definitions for at least the following type:

|                           |                                                                                                                                                                                           |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>clock_t</code>      | for the system time in units of <code>CLK_TCK</code> or <code>CLOCKS_PER_SEC</code> (see <code>time.h</code> )                                                                            |
| <code>blkcnt_t</code>     | for block numbers in files                                                                                                                                                                |
| <code>blkcnt64_t</code>   | for block numbers in files                                                                                                                                                                |
| <code>dev_t</code>        | for device IDs                                                                                                                                                                            |
| <code>fsblkcnt_t</code>   | for block numbers in file systems                                                                                                                                                         |
| <code>fsblkcnt64_t</code> | for block numbers in file systems                                                                                                                                                         |
| <code>fsfilcnt_t</code>   | for file serial numbers in file systems                                                                                                                                                   |
| <code>fsfilcnt64_t</code> | for file serial numbers in file systems                                                                                                                                                   |
| <code>gid_t</code>        | for group IDs                                                                                                                                                                             |
| <code>ino_t</code>        | for file serial numbers                                                                                                                                                                   |
| <code>key_t</code>        | for interprocess communication                                                                                                                                                            |
| <code>mode_t</code>       | for some file attributes                                                                                                                                                                  |
| <code>nlink_t</code>      | for link counts                                                                                                                                                                           |
| <code>off_t</code>        | for file sizes; the type is a signed integer.                                                                                                                                             |
| <code>off64_t</code>      | for file sizes; the type is a signed integer.                                                                                                                                             |
| <code>pid_t</code>        | for process IDs and process group IDs                                                                                                                                                     |
| <code>size_t</code>       | for sizes of objects; the type is an integral integer.                                                                                                                                    |
| <code>ssize_t</code>      | for a count of bytes or for error numbers. The type is a signed integer and may include values between <code>-1</code> and <code>SSIZE_MAX</code> inclusive (see <code>limits.h</code> ). |
| <code>time_t</code>       | for the time in seconds                                                                                                                                                                   |
| <code>uid_t</code>        | for user IDs                                                                                                                                                                              |

The data types `blkcnt64_t` and `off64_t` are defined as extended data types (signed).

The data types `fsblkcnt64_t`, `fsfilcnt64_t` and `ino64_t` are defined as unsigned extended data types.

The other data types except `key_t` are defined as arithmetic types of an appropriate length.

**See also**

`bsearch()`, `chmod()`, `chown()`, `closedir()`, `creat()`, `fcntl()`, `fstat()`, `getegid()`, `geteuid()`, `getgid()`, `getgroups()`, `getpgrp()`, `getpid()`, `getppid()`, `getuid()`, `kill()`, `lseek()`, `mkdir()`, `mkfifo()`, `msgctl()`, `msgget()`, `msgrcv()`, `msgsnd()`, `open()`, `opendir()`, `readdir()`, `rewinddir()`, `semctl()`, `semget()`, `semop()`, `setgid()`, `setpgid()`, `setsid()`, `setuid()`, `shmat()`, `shmctl()`, `shmdt()`, `shmget()`, `stat()`, `tcgetpgrp()`, `tcsetpgrp()`, `umask()`, `utime()`, `limits.h`, `time.h`.

## sys/uio.h - definitions for array I/O operations

**Syntax**      `#include <sys/uio.h>`

**Description** `sys/uio.h` defines the `iovec` structure:

```
typedef struct iovec {
    caddr_t iov_base; /* Basic address of the I/O memory area */
    int iov_len;      /* Size of the iov_base memory area */
} iovec_t;
```

The following functions are defined:

```
ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

**See also**    `read()`, `write()`.

## sys/utsname.h - system name structure

**Syntax**      `#include <sys/utsname.h>`

**Description** The `sys/utsname.h` header defines the structure `utsname`, which includes at least the following members:

|                              |                                                                                                  |
|------------------------------|--------------------------------------------------------------------------------------------------|
| <code>char sysname[]</code>  | Name of the operating system implemented on the local processor                                  |
| <code>char nodename[]</code> | Name of the node for the local processor within an implementation-specific communication network |
| <code>char release[]</code>  | Release number of the system implementation                                                      |
| <code>char version[]</code>  | Date of the release                                                                              |
| <code>char machine[]</code>  | Name of the hardware type on which the system is running                                         |

The character arrays are of unspecified size, but the data stored in them is terminated by a null byte.

The following name is declared as a function:

```
int uname(struct utsname *name);
```

**See also**    `uname()`.

## sys/wait.h - declarations for waiting

**Syntax**     **#include** <sys/wait.h>

**Description** The `sys/wait.h` header defines the following symbolic constants for use with `waitpid()`

**Description**The `sys/wait.h` header defines the following symbolic constants for use with `waitpid()`:

`WNOHANG`     Do not hang if no status is available; return immediately

`WUNTRACED`   Report status of stopped child process

and the following macros for analysis of process status values:

`WEXITSTATUS()`

    Returns exit status

`WIFEXITED()` True if child process exited normally

`WIFSIGNALED()`

    True if child process exited due to uncaught signal

`WIFSTOPPED()`

    True if child process is currently stopped

`WSTOPSIG()` Returns signal number that caused process to stop

`WTERMSIG()` Returns signal number that caused process to terminate

### *Extension*

`WIFCONTINUED()`

    Returns a non-zero value if the status of a continued child process was returned.

`WCOREDUMP()` If the value of `WIFSIGNALED()` is not equal to zero, the macro will return a non-zero value if a core dump was generated for the terminated child process. □

The following names are declared as functions:

```
pid_t wait(int *stat_loc);
```

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

**See also**     `exit()`, `wait()`, `waitpid()`, `sys/types.h`.

## tar.h - extended tar definitions

Syntax `#include <tar.h>`

Description The definitions for the header block are as follows:

General definitions:

| Name     | Value   | Description              |
|----------|---------|--------------------------|
| TMAGIC   | "ustar" | ustar plus null byte     |
| TMAGLEN  | 6       | Length of TMAGIC (ustar) |
| TVERSION | "00"    | 00 without a null byte   |
| TVERSLN  | 2       | Length of TVERSION (00)  |

Type flag field definitions:

| Name     | Value | Description            |
|----------|-------|------------------------|
| REGTYPE  | '0'   | Regular file           |
| AREGTYPE | '\0'  | Regular file           |
| LNKTYPE  | '1'   | Link                   |
| SYMTYPE  | '2'   | Reserved               |
| CHRTYPE  | '3'   | Character special file |
| BLKTYPE  | '4'   | Block special file     |
| DIRTYPE  | '5'   | Directory              |
| FIFOTYPE | '6'   | FIFO special file      |
| CONTTYPE | '7'   | Reserved               |

Mode field bit definitions (octal):

| <b>Name</b> | <b>Value</b> | <b>Description</b>                   |
|-------------|--------------|--------------------------------------|
| TSUID       | 04000        | Set UID on execution                 |
| TSGID       | 02000        | Set GID on execution                 |
| TSVTX       | 01000        | Reserved                             |
| TUREAD      | 00400        | Read permission for user             |
| TUWRITE     | 00200        | Write permission for user            |
| TUEXEC      | 00100        | Execute/Search permission for user   |
| TGREAD      | 00040        | Read permission for group            |
| TGWRITE     | 00020        | Write permission for group           |
| TGEXEC      | 00010        | Execute/Search permission for group  |
| TOREAD      | 00004        | Read permission for others           |
| TOWRITE     | 00002        | Write permission for others          |
| TOEXEC      | 00001        | Execute/Search permission for others |



## termios.h - define values for termios

**Syntax**     `#include <termios.h>`

**Description** The `termios.h` header contains the definitions used by the `termios` interface.

A number of unsigned type definitions exist for:

`cc_t`                    `speed_t`                `tcflag_t`

The structure `termios` contains the following members:

|                       |                         |                     |
|-----------------------|-------------------------|---------------------|
| <code>tcflag_t</code> | <code>c_iflag</code>    | Input processing    |
| <code>tcflag_t</code> | <code>c_oflag</code>    | Output processing   |
| <code>tcflag_t</code> | <code>c_cflag</code>    | Hardware attributes |
| <code>tcflag_t</code> | <code>c_lflag</code>    | Local processing    |
| <code>cc_t</code>     | <code>c_cc[NCCS]</code> | Control characters  |

A definition is given for:

`NCCS`                    Size of the array `c_cc` for control characters

Special control characters are defined by the array `c_cc`:

Subscript values:

| Standard                | Raw    | Description     |
|-------------------------|--------|-----------------|
| <b>input processing</b> |        |                 |
| VEOF                    |        | EOF character   |
| VEOL                    |        | EOL character   |
| VERASE                  |        | ERASE character |
| VINTR                   | INTR   | INTR character  |
| VKILL                   |        | KILL character  |
|                         | VMIN   | MIN value       |
| VQUIT                   | VQUIT  | QUIT character  |
| VSTART                  | VSTART | START character |
| VSTOP                   | VSTOP  | STOP character  |
| VSUSP                   | VSUSP  | SUSP character  |
|                         | VTIME  | TIME character  |

The subscript values are unique, except that the `VMIN` and `VTIME` subscripts may have the same values as `VEOF` and `VEOL`, respectively.

### Input processing

The `c_iflag` field describes the basic terminal input control:

|                     |                                        |
|---------------------|----------------------------------------|
| <code>BRKINT</code> | Signal interrupt on break              |
| <code>ICRNL</code>  | Map CR to NL on input                  |
| <code>IGNBRK</code> | Ignore break condition                 |
| <code>IGNCR</code>  | Ignore CR                              |
| <code>IGNPAR</code> | Ignore characters with parity errors   |
| <code>INLCR</code>  | Map NL to CR on input                  |
| <code>INPCK</code>  | Enable input parity check              |
| <code>ISTRIP</code> | Strip character                        |
| <code>IUCLC</code>  | Map uppercase to lowercase on input    |
| <code>IXANY</code>  | Enable any character to restart output |
| <code>IXOFF</code>  | Enable start/stop input control        |
| <code>IXON</code>   | Enable start/stop output control       |
| <code>PARMRK</code> | Mark parity errors                     |

### Output processing modes

The `c_oflag` field specifies how the system handles output:

|                     |                                       |
|---------------------|---------------------------------------|
| <code>OPOST</code>  | Post-process output                   |
| <code>OLCUC</code>  | Map lowercase to uppercase on output  |
| <code>ONLCR</code>  | Map NL to CR-NL on output             |
| <code>OCRNL</code>  | Map CR to NL on output                |
| <code>ONOCR</code>  | No CR output at column 0              |
| <code>ONLRET</code> | NL performs CR function               |
| <code>OFILL</code>  | Use fill characters for delays        |
| <code>OFDEL</code>  | Fill character is DEL; otherwise, NUL |
| <code>NLDLY</code>  | Select delays for NL:                 |

|        |                                    |
|--------|------------------------------------|
| NL0    | Newline character type 0           |
| NL1    | Newline character type1            |
| CRDLY  | Select delays for CR:              |
| CR0    | CR delay type 0                    |
| CR1    | CR delay type 1                    |
| CR2    | CR delay type 2                    |
| CR3    | CR delay type 3                    |
| TABDLY | Select delays for horizontal tabs: |
| TAB0   | Horizontal-tab delay type 0        |
| TAB1   | Horizontal-tab delay type 1        |
| TAB2   | Horizontal-tab delay type 2        |
| TAB3   | Expand tabs to spaces              |
| BSDLY  | Select delays for backspace:       |
| BS0    | Backspace-delay type 0             |
| BS1    | Backspace-delay type 1             |
| VTDLY  | Select delays for vertical tabs:   |
| VT0    | Vertical-tab delay type 0          |
| VT1    | Vertical-tab delay type1           |
| FFDLY  | Select delays for form-feed:       |
| FF0    | Form-feed delay type 0             |
| FF1    | Form-feed delay type 1             |

### Baud rate selection

The input and output baud rates are stored in the `termios` structure. These are the valid values for objects of type `speed_t`. The following values are defined, but all baud rates need not be supported by the underlying hardware:

|      |          |
|------|----------|
| B0   | Hang up  |
| B50  | 50 baud  |
| B75  | 75 baud  |
| B110 | 110 baud |

|        |            |
|--------|------------|
| B134   | 134.5 baud |
| B150   | 150 baud   |
| B200   | 200 baud   |
| B300   | 300 baud   |
| B600   | 600 baud   |
| B1200  | 1200 baud  |
| B1800  | 1800 baud  |
| B2400  | 2400 baud  |
| B4800  | 4800 baud  |
| B9600  | 9600 baud  |
| B19200 | 19200 baud |
| B38400 | 38400 baud |

### Hardware attributes

The `c_cflag` field describes the hardware control of the terminal; not all values specified are required to be supported by the underlying hardware:

|        |                                           |
|--------|-------------------------------------------|
| CSIZE  | Character size                            |
| CS5    | 5 bits                                    |
| CS6    | 6 bits                                    |
| CS7    | 7 bits                                    |
| CS8    | 8 bits                                    |
| CSTOPB | Send two stop bits, else one              |
| CREAD  | Enable receiver                           |
| PARENB | Parity enable                             |
| PARODD | Odd parity, else even                     |
| HUPCL  | Hang up on last close                     |
| CLOCAL | Local connection, else dial-up connection |

### Local processing modes

The `c_lflag` field of the argument structure is used to control various terminal functions:

|        |                                                         |
|--------|---------------------------------------------------------|
| ECHO   | Enable echo                                             |
| ECHOE  | Echo erase character as backspace                       |
| ECHOK  | Echo KILL                                               |
| ECHONL | Echo NL                                                 |
| ICANON | Standard input processing                               |
| IEXTEN | Extended input character processing                     |
| ISIG   | Enable signal processing                                |
| NOFLSH | Disable flush after interrupt or quit                   |
| TOSTOP | Send SIGTTOU for background output                      |
| XCASE  | Canonical representation of uppercase/lowercase letters |

### Attribute selection

The following symbolic constants are defined for `tcsetattr()`:

|           |                                                                     |
|-----------|---------------------------------------------------------------------|
| TCSANOW   | Change attributes immediately                                       |
| TCSADRAIN | Change attributes when output has drained                           |
| TCSAFLUSH | Change attributes when output has drained; also flush pending input |

### Line control

The following symbolic constants for use with `tcflush()` are defined:

|           |                                                   |
|-----------|---------------------------------------------------|
| TCIFLUSH  | Flush pending input                               |
| TCOFLUSH  | Flush untransmitted output                        |
| TCIOFLUSH | Flush both pending input and untransmitted output |

The following symbolic constants for use with `tcflow()` are defined:

|                     |                                                            |
|---------------------|------------------------------------------------------------|
| <code>TCIOFF</code> | Transmit a STOP character, intended to suspend input data  |
| <code>TCION</code>  | Transmit a START character, intended to restart input data |
| <code>TCOOFF</code> | Suspend output                                             |
| <code>TCOON</code>  | Restart output                                             |

The following names are declared as functions:

```
cfgetispeed()  tcdrain()      tcgetattr()
cfgetospeed() tcflow()      tcsendbreak()
cfsetispeed()  tcflush()     tcsetattr()
cfsetospeed()
```

**See also** `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`, `tcdrain()`, `tcflow()`, `tcflush()`, `tcgetattr()`, `tcsendbreak()`, `tcsetattr()`.

## time.h - time types

Syntax `#include <time.h>`

Description The `time.h` header declares the structure `tm`, which includes at least the following members:

|                           |                                                                                                                                                                  |
|---------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int tm_sec</code>   | Seconds [0, 61]. The range [0,61] allows for the leap second.                                                                                                    |
| <code>int tm_min</code>   | Minutes [0, 59]                                                                                                                                                  |
| <code>int tm_hour</code>  | Hours [0, 23]                                                                                                                                                    |
| <code>int tm_mday</code>  | Day of month [1, 31]                                                                                                                                             |
| <code>int tm_mon</code>   | Month of year [0, 11]                                                                                                                                            |
| <code>int tm_year</code>  | Years since 1900                                                                                                                                                 |
| <code>int tm_wday</code>  | Day of week [0, 6] (Sonntag = 0)                                                                                                                                 |
| <code>int tm_yday</code>  | Day of year [0, 365]                                                                                                                                             |
| <code>int tm_isdst</code> | Variable for daylight saving time; the value is positive if daylight saving time is in effect, 0 if it is not, and negative if the information is not available. |

The following symbolic names are defined:

|                             |                                                                                                                                                                                                                                                                                                                |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>NULL</code>           | Null pointer                                                                                                                                                                                                                                                                                                   |
| <code>CLK_TCK</code>        | Number of clock ticks per second returned by <code>time()</code> ; the value is currently the same as the return value of <code>sysconf(_SC_CLK_TCK)</code> ; however, new applications should call <code>sysconf()</code> (Will no longer be supported by the X/Open-Standard in future.).                    |
| <code>CLOCKS_PER_SEC</code> | A number used to convert the value returned by the <code>clock()</code> function into seconds. The value is required to be 1 million on all X/Open-conformant systems, but may be variable on other systems. It should, however, never be assumed that <code>CLOCKS_PER_SEC</code> is a compile-time constant. |

The data types `clock_t`, `size_t` and `time_t` are defined (see `sys/types.h`).

The following names are declared as functions:

```
char *asctime(const struct tm *timeptr);
clock_t clock(void);
char *ctime(const time_t *clock);
double difftime(time_t *time1, time_t *time0)
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
time_t mktime(struct tm *timeptr);
size_t strftime(char *s, size_t maxsize, const char *format);
char *strptime(const char *buf, const char *format);
time_t time(time_t *tloc);
void tzset(void);
```

The following are declared as external variables:

```
extern int daylight;
extern long int timezone;
extern char *tzname[]
```

**See also** `asctime()`, `clock()`, `ctime()`, `daylight`, `difftime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `strptime()`, `sysconf()`, `time()`, `timezone`, `tzname()`, `tzset()`, `utime()`.



## ucontext.h - user context

**Syntax**      `#include <ucontext.h>`

**Description** Defines the `mcontext_t` type:

```
typedef struct {
    gregset_t    _gregs;
    fpregset_t  _fpregs;
    /* BS2000 */
    int          _uc_cc;
    int          _uc_pc;
    char        _uc_fpvalid;
    char        _uc_type;
    short       _uc_level;
    int         _uc_priv1;
    int         _uc_priv2;
    int         _uc_priv3[256];
    /* BS2000 */
} mcontext;
```

This header also defines the `ucontext` structure, which in turn defines the control context within an executing process:

```
typedef struct ucontext {
    ulong_t     uc_flags;
    struct ucontext *uc_link;
    sigset_t    uc_sigmask;
    stack_t     uc_stack;
    mcontext_t  uc_mcontext;
} ucontext_t;
```

`uc_link` is a pointer to the context, which is to be taken up again when the current context is terminated. If `uc_link` is zero, the current context is the main context; the process is terminated when this context is terminated.

`uc_sigmask` defines the signal sets which are blocked when this context is active (see `sigprocmask(2)`).

`uc_stack` defines the stack for this context (see `sigaltstack(2)`).

`uc_mcontext` contains the saved machine register and implementation-specific context data. Portable applications should not modify `uc_mcontext`.

The following functions are defined:

```
int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);
void makecontext(ucontext_t *ucp, (void *func)(), int argc, ...);
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

See also `getcontext(2)`, `sigaction(2)`, `sigprocmask(2)`, `sigaltstack(2)`, `makecontext(3C)`.

## ulimit.h - commands for ulimit

Syntax `#include <ulimit.h>`

Description The `ulimit.h` header defines the symbolic constants that can be used in the `ulimit()` function.

Symbolic constants:

`UL_GETFSIZE`           Get maximum file size

`UL_SETFSIZE`           Set maximum file size

The following name is defined as a function or macro:

```
long int ulimit (int cmd, ...);
```

See also `ulimit()`.

## unistd.h - standard symbolic constants and structures

Syntax `#include <unistd.h>`

Description The `unistd.h` header defines miscellaneous symbolic constants and types, and declares various functions.

### Version test macros

The following symbolic constants are defined with fixed values:

|                                |                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>_POSIX_VERSION</code>    | Integer value indicating version of the ISO POSIX-1 standard (C language binding). The value changes with each new version of the standard.                                                                                                                                                                                                                                                    |
| <code>_POSIX2_VERSION</code>   | Integer value indicating version of the ISO POSIX-2 standard (Commands). The value changes with each new version of the standard.                                                                                                                                                                                                                                                              |
| <code>_POSIX2_C_VERSION</code> | Integer value indicating version of the ISO POSIX-2 standard (C language binding) and whether the X/Open POSIX2 C-language binding functions are supported. The value changes with each new version of the standard. If the C-language binding option of the ISO POSIX-2 standard, and thus the X/Open POSIX2 C-language binding feature group, is not supported, the value will be set to -1. |
| <code>_XOPEN_VERSION</code>    | Integer value indicating the XPG version to which the system conforms. It has a value of 4 in this implementation.                                                                                                                                                                                                                                                                             |

### Mandatory symbolic constants

The following symbolic constants are either undefined or defined with a value  $\neq -1$ . If a constant is undefined, an application should use `sysconf()`, `pathconf()` or `fpathconf()` to determine which features are currently present on the system.

|                                      |                                                                                                                                     |
|--------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <code>_POSIX_CHOWN_RESTRICTED</code> | The use of <code>chown()</code> is restricted to processes with appropriate privileges. The value is $\neq -1$ .                    |
| <code>_POSIX_NO_TRUNC</code>         | Pathname components longer than <code>{NAME_MAX}</code> generate an error. The value is $\neq -1$ .                                 |
| <code>_POSIX_VDISABLE</code>         | Special characters for terminal output can be disabled using this character (see <code>termios.h</code> ). The value is $\neq -1$ . |

`_POSIX_SAVED_IDS` Each process has a saved set-user-ID and a saved set-group-ID.

`_POSIX_JOB_CONTROL`  
The implementation supports job control.

### Constants for options and feature groups

The following symbolic constants are defined to have the value -1 if the implementation does not support the corresponding feature, and a value  $\neq$  -1 if the implementation supports the feature. If these constants are undefined, the `sysconf` function can be used to check whether the feature is supported for a particular use of the application.

`_POSIX2_C_DEV` The implementation supports the C language development utilities option.

`_POSIX2_CHAR_TERM` The implementation supports at least one terminal type.

`_POSIX2_FORT_DEV` The implementation supports the FORTRAN development utilities option.

`_POSIX2_FORT_RUN` The implementation supports the FORTRAN runtime utilities option.

`_POSIX2_LOCALEDEF` The implementation supports the creation of locales by the `localedef` command.

`_POSIX2_SW_DEV` The implementation supports the software development utilities option.

`_POSIX2_UP` The implementation supports the user portability utilities option.

`_XOPEN_CRYPT` The implementation supports the die X/Open encryption feature group.

`_XOPEN_SHM` The implementation supports the X/Open shared memory feature group.

The following macros provide information on the version used or the implementation:

`_LFS_LARGEFILE`

`_LFS64_LARGEFILE`

`_LFS_ASYNCHRONOUS_IO`

`_LFS64_ASYNCHRONOUS_IO`

`_LFS_STDIO`

`_LFS64_STDIO`

## Constants for functions

The following symbolic constants are defined independently of any function:

NULL                      Null pointer

The following symbolic constants are defined for the `access()` function:

R\_OK                      Test for read permission

W\_OK                      Test for write permission

X\_OK                      Test for execute/search permission

F\_OK                      Test for existence of file

The constants `F_OK`, `R_OK`, `W_OK` and `X_OK` and the expressions `R_OK|W_OK`, `R_OK|X_OK` and `R_OK|W_OK|X_OK` all have distinct values.

The following symbolic constant is defined for the `confstr()` function:

\_CS\_PATH                 Value for the `PATH` environment variable that finds all standard commands.

\_CS\_LFS\_CFLAGS

\_CS\_LFS\_LDFLAGS

\_LFS\_LIBS

\_CS\_LFS\_LINTFLAGS

\_CS\_LFS64\_CFLAGS

\_CS\_LFS64\_LDFLAGS

\_LFS\_LIBS64

\_CS\_LFS64\_LINTFLAGS

The following symbolic constants are defined for the `lseek()` and `fcntl()` functions (they have distinct values):

SEEK\_SET                 Set file offset to *offset*.

SEEK\_CUR                 Set file offset to current position plus *offset*.

SEEK\_END                 Set file offset to EOF plus *offset*.

The following symbolic constants are defined for `sysconf()`:

```
_SC_2_C_BIND  
_SC_2_C_DEV  
_SC_2_C_VERSION  
_SC_2_FORT_DEV  
_SC_2_FORT_RUN  
_SC_2_LOCALEDEV  
_SC_2_SW_DEV  
_SC_2_UPE  
_SC_2_VERSION  
_SC_ARG_MAX  
_SC_BC_BASE_MAX  
_SC_BC_DIM_MAX  
_SC_BC_SCALE_MAX  
_SC_BC_STRING_MAX  
_SC_CHILD_MAX  
_SC_CLK_TCK  
_SC_COLL_WEIGHTS_MAX  
_SC_EXPR_NEST_MAX  
_SC_JOB_CONTROL  
_SC_LINE_MAX  
_SC_NGROUPS_MAX  
_SC_OPEN_MAX  
_SC_PASS_MAX (Will no longer be supported by the X/Open-Standard in future.)  
_SC_RE_DUP_MAX  
_SC_SAVED_IDS  
_SC_STREAM_MAX  
_SC_TZNAME_MAX  
_SC_VERSION  
_SC_XOPEN_CRYPT  
_SC_XOPEN_ENH_I18N  
_SC_XOPEN_SHM  
_SC_XOPEN_VERSION
```

The following symbolic constants are defined for `pathconf()`:

```
_PC_CHOWN_RESTRICTED
_PC_LINK_MAX
_PC_MAX_CANON
_PC_MAX_INPUT
_PC_NAME_MAX
_PC_NO_TRUNC
_PC_PATH_MAX
_PC_PIPE_BUF
_PC_VDISABLE
```

The following symbolic constants are defined for file streams:

```
STDIN_FILENO      File number of stdin. The value is 0.
STDOUT_FILENO     File number of stdout. The value is 1.
STDERR_FILENO     File number of stderr. The value is 2.
```

### Data type definitions

The `size_t`, `ssize_t`, `uid_t`, `gid_t`, `off_t`, and `pid_t` types are defined as described in `sys/types.h`.

### Declarations

The following names are declared as functions:

```
int access(const char *path, int amode);
unsigned int alarm(unsigned int seconds);
int chdir(const char *path);
int chown(const char *path, uid_t owner, gid_t group);
int chroot(const char *path); (Will no longer be supported by the X/Open-Standard
in future.)
int close(int fildes);
size_t confstr(int name, char *buf, size_t len);
char *crypt(const char *key, const char *salt);
char *ctermid(char *s);
char *cuserid(char *s); (Will no longer be supported by the X/Open-Standard in
future.)
int dup(int fildes);
int dup2(int fildes, int fildes2);
void encrypt(char block[64], int edflag);
int execl(const char *path, const char *arg0, ...);
int execlp(const char *file, const char *arg0, ...);
```

```
int execlp(const char *file, const char *arg0, ...);
int execv(const char *path, char * const argv[]);
int execve(const char *path, char * const argv[], char *const envp[]);
int execvp(const char *file, char * const argv[]);
void _exit(int status);
pid_t fork(void);
long int fpathconf(int fildes, int name);
int fsync(int fildes);
int ftruncate64 (int fildes, off64_t length);
char *getcwd(char *buf, size_t size);
gid_t getegid(void);
uid_t geteuid(void);
gid_t getgid(void);
int getgroups(int gidsetsize, gid_t grouplist[]);
char *getlogin(void);
int getlogin_r(char *name, size_t namesize);
int getopt(int argc, char * const argv[], const char *optstring);
char *getpass(const char *prompt); (Will no longer be supported by the X/Open-
Standard in future.)
pid_t getpgrp(void);
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
int isatty(int fildes);
int link(const char *path1, const char *path2);
int lockf64(int fildes, int function, off64_t size);
off_t lseek(int fildes, off_t offset, int whence);
off64_t lseek64 (int fildes, off64_t offset, int whence);
int nice(int incr);
long int pathconf(const char *path, int name);
int pause(void);
int pipe(int fildes[2]);
ssize_t read(int fildes, void *buf, size_t nbyte);
int rmdir(const char *path);
int setgid(gid_t gid);
int setpgid(pid_t pid, pid_t pgid);
pid_t setsid(void);
int setuid(uid_t uid);
unsigned int sleep(unsigned int seconds);
void swab(const void *src, void *dest, ssize_t nbytes);
long int sysconf(int name);
pid_t tcgetpgrp(int fildes);
int tcsetpgrp(int fildes, pid_t pgrp_id);
int truncate64 (const char *path, off64_t length);
```



```
char *ttyname(int fildes);  
int unlink(const char *path);  
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

The following external variables are declared:

```
extern char *optarg;  
extern int optind, opterr, optopt;
```

**See also**

access(), alarm(), chdir(), chown(), chroot(), close(), crypt(),  
ctermid(), cuserid(), dup(), encrypt(), environ, exec, exit(), fcntl(),  
fork(), fpathconf(), fsync(), getcwd(), getegid(), geteuid(), getgid(),  
getgroups(), getlogin(), getlogin\_r(), getpass(), getpgrp(), getpid(),  
getppid(), getuid(), isatty(), link(), lseek(), nice(), pathconf(),  
pause(), pipe(), read(), rmdir(), setgid(), setpgid(), setsid(), setuid(),  
sleep(), swab(), sysconf(), tcgetpgrp(), tcsetpgrp(), ttyname(),  
unlink(), write(), limits.h, sys/types.h, termios.h.

## utime.h - manipulate time structures

Syntax `#include <utime.h>`

Description The `utime.h` header declares the structure `utimbuf`, which includes the following members:

|                     |                      |                   |
|---------------------|----------------------|-------------------|
| <code>time_t</code> | <code>actime</code>  | Access time       |
| <code>time_t</code> | <code>modtime</code> | Modification time |

The times are measured in seconds since the Epoch.

The type `time_t` is declared in `sys/types.h`.

The following name is declared as a function:

```
int utime(const char *path, const struct utimbuf *times);
```

See also `utime()`, `sys/types.h`.

## utmpx.h - entry format

**Syntax**     **#include <utmpx.h>**

**Description** The `utmpx` structure is defined:

```
struct utmpx {
    char ut_user[32];           /* User ID */
    char ut_id[4];             /* ID of the Inittab */
    char ut_line[32];         /* Device name (console, lnxx) */
    pid_t ut_pid;              /* Process ID */
    short ut_type;             /* Type of the entry */
    struct exit_status ut_exit; /* Exit status */
    struct timeval ut_tv;      /* Timestamp of the entry */
    long ut_session;           /* Session ID with windowing */
    long pad[5];               /* Reserved for future versions */
    short ut_syslen;           /* Significant length ut_host (incl. \0)
*/
    char ut_host[257];         /* Name of the remote host */
};
```

The `/var/adm/utmpx` file contains user and accounting information for commands such as `who()`, `write()` and `login()`.

The following symbolic constants are defined as possible values for the data member `ut_type` of the `utmpx` structure:

|                            |                                                                            |
|----------------------------|----------------------------------------------------------------------------|
| <code>EMPTY</code>         | No valid accounting information exists for the user.                       |
| <code>BOOT_TIME</code>     | Time when the system was booted.                                           |
| <code>OLD_TIME</code>      | Time when the system time was changed.                                     |
| <code>NEW_TIME</code>      | Time after the system time has been changed.                               |
| <code>USER_PROCESS</code>  | Identifies a process.                                                      |
| <code>INIT_PROCESS</code>  | Identifies a process that was generated by the <code>INIT_</code> process. |
| <code>LOGIN_PROCESS</code> | Identifies the session leader of a logged-in user.                         |
| <code>DEAD_PROCESS</code>  | Identifies a session leader which has terminated.                          |

The following functions are defined:

```
void endutxent(void);
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);
void setutxent(void);
```

**See also**     `endutxent()`.

## varargs.h - handle variable argument list

Syntax `#include <varargs.h>`

`va_alist`

`va_dcl`

`void va_start(pvar)`

`va_list pvar;`

`type va_arg(pvar, type)`

`va_list pvar;`

`void va_end(pvar)`

`va_list pvar;`

Description The `varargs.h` header contains a set of macros which allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists (such as `printf()`) but do not use `varargs.h` are non-portable, since different systems use different argument-passing conventions.

`va_alist` Used as the parameter list in a function header.

`va_dcl` A declaration for `va_alist`. No semicolon should follow `va_dcl`.

`va_list` A type defined for the variable used to traverse the list.

`va_start()` Called to initialize `pvar` to the beginning of the list.

`va_arg()` Will return the next argument in the list pointed to by `pvar`. `type` is the type assumed for the argument. Different types can be mixed, but it is up to the routine to know what type of argument is expected next, as this cannot be determined at runtime.

`va_end()` Used to "clean up".

Multiple traversals, each bracketed by `va_start() ... va_end()`, are possible.

Notes It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, `exec1()` is passed a zero pointer to signal the end of the list. The `printf()` function can tell how many arguments are there by the format (i.e. from the first argument).

It is non-portable to specify a second argument of type `char`, `short` or `float` to `va_arg()`, since arguments seen by the called function are not of type `char`, `short` or `float`. The C language converts type `char` and `short` arguments to `int` and converts type `float` arguments to `double` before passing them to a function.

See also `iconvexec`, `printf()`.

## wchar.h - types for wide character values

Syntax `#include <wchar.h>`

Description The following data types are defined by typedef:

|                       |                                                                                                                   |
|-----------------------|-------------------------------------------------------------------------------------------------------------------|
| <code>wchar_t</code>  | See <code>stddef.h</code>                                                                                         |
| <code>wint_t</code>   | An integral type capable of storing any valid values of <code>wchar_t</code> or <code>WEOF</code> .               |
| <code>wctype_t</code> | A scalar type of a data object that can accept values which represent a locale-specific character classification. |
| <code>FILE</code>     | See <code>stdio.h</code>                                                                                          |
| <code>size_t</code>   | See <code>stddef.h</code>                                                                                         |

The following names are declared as functions:

```
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t *s, int n; FILE *stream);
wint_t fputwc(wint_t c, FILE *stream);
int fputws(const wchar_t *s, FILE *stream);
wint_t getwc(FILE *stream); (also defined as a macro)
wint_t getwchar(void); (also defined as a macro)
wchar_t *getws(wchar_t *s);

int iswalnum(wint_t wc); (also defined as a macro)
int iswalph(wint_t wc); (also defined as a macro)
int iswcntrl(wint_t wc); (also defined as a macro)
int iswdigit(wint_t wc); (also defined as a macro)
int iswgraph(wint_t wc); (also defined as a macro)
int iswlower(wint_t wc); (also defined as a macro)
int iswprint(wint_t wc); (also defined as a macro)
int iswpunct(wint_t wc); (also defined as a macro)
int iswspace(wint_t wc); (also defined as a macro)
int iswupper(wint_t wc); (also defined as a macro)
int iswxdigit(wint_t wc); (also defined as a macro)
int iswctype(wint_t wc, wctype_t prop); (also defined as a macro)
wint_t putwc(wint_t c, FILE *stream); (also defined as a macro)
wint_t putwchar(wint_t c); (also defined as a macro)
int putws(const wchar_t *s);
wint_t tolower(wint_t wc); (also defined as a macro)
wint_t towupper(wint_t wc); (also defined as a macro)
```

```

wint_t ungetwc(wint_t c, FILE *stream);
wctype_t wctype(const char *property);
wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcschr(const wchar_t *ws, wchar_t wc);
int wcsncmp(const wchar_t *ws1, const wchar_t *ws2);
int wscoll(const wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);
size_t wcsncpy(const wchar_t *ws1, const wchar_t *ws2);
size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *fmt,
                const struct tm *timptr);
size_t wcslen(const wchar_t *ws);
wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
wchar_t *wcpbrk(const wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc)
size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
double wcstod(const wchar_t *nptr, wchar_t **endptr);
wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2);
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
unsigned long int wcstoul (const wchar_t *nptr, wchar_t **endptr,
                          int base);
wchar_t *wcsvcs(const wchar_t *ws1, const wchar_t *ws2);
int wcswidth(const wchar_t *pwcs, size_t n);
size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
int wcwidth(const wchar_t wc);

```

The following macro names are defined:

|      |                                                                                                                   |
|------|-------------------------------------------------------------------------------------------------------------------|
| WEOF | Constant expression of type <code>wint_t</code> that is returned by multi-byte functions to indicate end-of-file. |
| NULL | See <code>stddef.h</code> .                                                                                       |

**Notes** Inclusion of the `wchar.h` header may make all symbols from the headers `ctype.h`, `stdio.h`, `stdarg.h`, `stdlib.h`, `string.h`, `stddef.h` and `time.h` visible.

**See also** `iswalnum()`, `iswalpha`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `mblen()`, `mbstowcs()`, `mbtowc()`, `setlocale()`, `tolower()`, `toupper()`, `wcstombs()`, `wctomb()`, `locale.h`, `stddef.h`.

## wctype.h – wide character classification and mapping utilities

Syntax `#include <wctype.h>`

Description The following data types are defined by typedef:

|                        |                                                                                                                   |
|------------------------|-------------------------------------------------------------------------------------------------------------------|
| <code>wint_t</code>    | See <code>wchar.h</code> .                                                                                        |
| <code>wctrans_t</code> | A scalar type of a data object that can accept values which represent a locale-specific character classification. |
| <code>wctype_t</code>  | See <code>wchar.h</code> .                                                                                        |

The following names are declared as functions:

```
int iswalnum(wint_t);
int iswalpha(wint_t);
int iswcntrl(wint_t);
int iswdigit(wint_t);
int iswgraph(wint_t);
int iswlower(wint_t);
int iswprint(wint_t);
int iswpunct(wint_t);
int iswspace(wint_t);
int iswupper(wint_t);
int iswxdigit(wint_t);
int iswctype(wint_t, wctype_t);
wint_t towctrans(wint_t, wctrans_t);
wint_t tolower(wint_t);
wint_t toupper(wint_t);
wctrans_t wctrans(const char *);
wctype_t wctype(const char *);
```

The following macro names are defined:

`WEOF` Constant expression of type `wint_t` that is returned by multi-byte functions to indicate end-of-file.

For all functions described in this include file that accept an argument of type `wint_t`, the value is represented as `wchar_t` or is equal to the value of `WEOF`. The response is undefined for all other values of this argument.

The response of this function is affected by the `LC_CTYPE` category of the current locale.

**Notes** Inclusion of the `wctype.h` header may also make all the symbols defined in `ctype.h`, `stdio.h`, `stdarg.h`, `stdlib.h`, `string.h`, `stddef.h`, `time.h` and `wchar.h` visible.

**See also** `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `towctrans()`, `towlower()`, `towupper()`, `wcttrans()`, `wctype()`, `locale.h`, `wchar.h`.



---

## Appendix: KR or ANSI functionality

All details presented in this section apply to the functions marked with xx in the table on page 19ff (Scope of the supported C library).

When the C library functions were first introduced with C V1.0, the ANSI-defined C library scope did not exist. The implementation was therefore based on the "provisional" definition by Kernighan & Ritchie ("KR") and on the commercially available UNIX/SINIX implementations.

The alignment of the original C library functions to the ANSI standard (C V2.0) has led to a few deviations in the execution of some I/O functions as compared with the predecessor version. In order to meet the requirements of the ANSI standard in full on one hand, while preserving the runtime behavior of "old-style" programs on the other, the I/O functions affected by these deviations in C/C++ versions V2.xx are now offered in two variants: with the new ANSI functionality and with the original "KR" functionality compatible with C V1.0.

The desired functionality is selected at compile time with the following compiler option:

```
SOURCE-PROPERTIES=PAR(LIBRARY-SEMANTICS=STD|V1-COMPATIBLE)
```

KR functionality (V1-COMPATIBLE) can only be selected in the KR and ANSI compilation modes. In the STRICT-ANSI and CPLUSPLUS compilation modes, the V1-COMPATIBLE specification is ignored, and STD is automatically assumed.

KR or ANSI functionality applies to the calls of all the library functions of a compilation unit.

### Important

If the same file is processed in a number of separately compiled source programs, these source programs must be compiled with the same LIBRARY-SEMANTICS parameter!

KR functionality cannot be enabled when programs are developed in the POSIX shell. In other words, all the I/O functions are always executed with ANSI functionality.

As of C/C++ V3.0 the KR functionality is no longer available.

The differences between KR and ANSI functionality are listed below.

### KR functionality

1. Default attributes of text files

When a new text file is created, it is generated as a SAM file with variable record length.

2. Location of the file position indicator in append mode

If the file position indicator of a file opened in append mode was explicitly moved from the end of the file (with `rewind()`, `fsetpos()`, `fseek()`, or `lseek()`), it will be automatically reset to the end of the file only when writing with the elementary function `write()`.

When a file is opened in append mode and for reading, the file position indicator will be set to the end of the file when the file is opened. The original contents of existing files are preserved.

3. ISAM files (flushing of buffers)

If the data of an ISAM file in the buffer does not end with a newline character, writing to the external file causes a change of record. Subsequent data is written to a new record.

4. `ungetc()`

When the contents of the buffer are written to the external file, the original data will be changed if a character other than the last character read was pushed back in the buffer.

5. Interpretation of the tab character (`\t`)

For output to text files of FCB type SAM or ISAM, the tab character is converted by default into the appropriate number of blanks.

6. `fprintf()`, `printf()`, `sprintf()`, `fscanf()`, `scanf()`, `sscanf()`

The ANSI extensions of the formatting and conversion characters are not available. The syntax and semantics of the predecessor version apply.

7. `vfprintf()`, `vprintf()`, `vsprintf()`

The conversion character `L` cannot be used, since the type `long double` is not supported in KR mode.

## ANSI functionality

1. Default attributes of text files  
When a new text file is created, it is generated as a ISAM file with variable record length.
2. Location of the file position indicator in append mode  
If the file position indicator of a file opened in append mode was explicitly moved from the end of the file (with `rewind()`, `fsetpos()`, `fseek()`, or `lseek()`), the current position will be ignored for all write functions, and the file position indicator will be automatically set to the end of the file.

When a file is opened in append mode and for reading, the file position indicator will be set to the end of the file when the file is opened. The original contents of existing files are preserved.

3. ISAM files (flushing of buffers)  
If the data of an ISAM file in the buffer does not end with a newline character, writing to the external file does not cause a change of record. Subsequent data extends the record in the file. In other words, when reading an ISAM file, only the newline characters explicitly written by the program are read.

If reading from any text file requires a data transfer from the external file to the internal C buffer, all ISAM file data that still in the buffer will be automatically written out to the files.

4. `ungetc()`  
When the contents of the buffer are written to the external file, the original data will not be changed if a character other than the last character read was pushed back in the buffer. The original data before the `ungetc` call is always written to the external file.
5. Interpretation of the tab character (`\t`)  
For output to text files of FCB type SAM or ISAM, the tab character is not converted by default into the appropriate number of blanks, but is written to the file as a text character (EBCDIC value).



---

# Glossary

The most important terms used in the manual are listed and explained here in alphabetical order.

## **8-bit transparency**

The ability of a software component to process 8-bit characters without modifying or utilizing any part of the character in a way that is inconsistent with the rules of the current coded character set.

## **absolute pathname**

A pathname beginning with the root directory of the POSIX file system and leading to a specific file or directory. Every file and every directory has a unique absolute pathname (see `pathname resolution`).

## **access mode**

The method used to access the records of a file.

## **account number**

*BS2000:*

Designates an account for the associated user ID. Multiple user IDs may be assigned the same account number. Each user ID can be provided with a maximum of 60 account numbers. The account number is evaluated at LOGON and at the time of an ENTER-JOB.

## **address**

In general, a number used to specify a memory location.

## **address space**

The memory area that can be accessed by a process.

## **alert**

An audible or visual indication at the user's terminal that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output.

### **alert character**

A character that in the output stream should cause a terminal to alert its user via a visual or audible signal. The alert character is the character designated by `\a` in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function.

### **alias name**

A word consisting solely of underscores ( `_` ), digits, alphabetic characters from the portable character set, and the characters `!`, `%`, and `@`. Other implementations may allow other characters within alias names as an extension.

### **appropriate privileges**

Special privileges needed by some of the function calls and function call options defined in the this manual. In accordance with the POSIX standard, this term supersedes the older concept of system administrator privileges.

### **argument**

In the shell, an argument is a parameter that is passed to a utility. This parameter is the equivalent of a single string in the `argv` array created by one of the `exec` functions. An argument can be one of the options, option-arguments or operands following the command name.

In the C language, an argument is a string that passes data to a function. The arguments of a function are specified within parentheses, which follow the function name. The number of arguments may also be zero. If two or more arguments are specified, they must be delimited by commas. The definition of a function includes a description of the number and types of arguments.

### **authentication**

A verification of user entries when logging on at the system. The user attributes "user ID" and "password" are checked against the entries in the join file (also called a user catalog).

### **background**

A method of executing a program in which no dialog between the user and computer occurs during program execution. The shell displays its prompt while the program is executing, so further commands may be invoked at the terminal (see `foreground`).

**background process**

A process which is a member of a background process group and which does not fully utilize system resources, but allows the simultaneous execution of other (generally more important) processes. A background process normally utilizes time gaps in which the processor would be otherwise unoccupied.

**background process group**

Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal.

**backslash**

The character `\`, also known as a reverse solidus.

**backspace character**

A character that, in the output stream, should cause printing (or displaying) to occur one column position previous to the position about to be printed. If the position about to be printed or displayed is at the first column of the current line, the behavior is unspecified. The backspace is the character designated by `\b` in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the backspace function.

**binary file**

An ordered sequence of bytes. The data written by C output functions is transferred to a binary file on a 1:1 basis. In contrast to text files, control characters for line feeds and tabs are non converted (see `text file`), but are mapped as corresponding EBCDIC values. Data that is read from a binary file thus corresponds precisely to the data that was originally written to the file.

The following files are binary files with stream-oriented I/O: cataloged PAM files, temporary PAM files (INCORE), and cataloged SAM files that were opened with `fopen()` or `freopen()` in binary mode.

The following files are binary files with record-oriented I/O: cataloged ISAM files, cataloged SAM files, and cataloged PAM files that were opened with the function `fopen()` or `freopen()` in binary mode and with the option `"type=record"`.

Binary mode can only be specified with the `fopen()` and `freopen()` functions. The elementary functions `open()` and `creat()` always open SAM and ISAM files as text files.

**block special file**

A special file for block-oriented I/O devices. A block special file is normally distinguished from a character special file by the fact that it provides access to the device in a manner such that the hardware characteristics of the device are not visible.

**block-mode terminal**

A terminal that does not support character-based input and output operations.

**buffer**

A memory area in which data is temporarily stored.

**buffering**

For all output functions that write data to text files and binary files with stream-oriented I/O (`printf()`, `putc()`, `fwrite()` etc.), data is initially stored in a buffer and is not written to the external file until a specific event occurs. This differs for text and binary files.

**carriage-return character**

A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred. The carriage-return is the character designated by `\r` in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line.

**character**

A sequence of one or more bytes representing a single graphic symbol or control code. This term also applies to multi-byte characters and single-byte characters, where a single-byte character is a special case of a multi-byte character.

**character class**

A named set of characters sharing an attribute associated with the name of the class. The classes and the characters contained in the set are dependent on the value of the `LC_CTYPE` category in the current locale.



**character set**

In the international "C" locale, characters are encoded according to the rules of the 7-bit US ASCII coded character set. Each character of the character set is assigned various attributes, such as a graphic symbol, possible conversions into corresponding uppercase or lowercase letters, the character class to which it belongs, and a position within the codeset collating sequence. Different native language character sets could be used in internationalized programs.

**character special file**

A special file for character-oriented I/O devices. One example of a character special file is a terminal device file.

**character string**

A contiguous sequence of characters that contains a null byte as the last element.

**child directory**

A directory that is under another directory at the next-higher level of the file system.

**child process**

See `process`.

**clock tick**

The (machine-specific) number of intervals per second is defined by `{CLK_TCK}`. It is used to express the value in type `clock_t` as returned by `time.h`.

**collating element**

The smallest entity used to determine the logical ordering of character or wide-character strings (see `collation sequence`). A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the `LC_COLLATE` category in the current locale determines the current set of collating elements.

### **collating sequence**

The relative order of collating elements, as determined by the setting of the `LC_COLLATE` category in the current locale.

The character order, as defined for the `LC_COLLATE` category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

Multi-level sorting is accomplished by assigning the collating elements one or more collation weights, up to the limit `{COLL_WEIGHTS_MAX}` (see the header file `limits.h`).

On each level, elements may be given the same weight (at the primary level, called an equivalence class; see `equivalence class`) or be omitted from the sequence. Strings that collate equal using the first assigned weight (primary ordering) are then compared using the next assigned weight (secondary ordering), and so on.

### **collation order**

The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements.

### **column position**

The distance of a character from the start of a line. It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The XPG4 standard utilities, when used as described in this manual, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes).

The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line.

### **command**

A directive to the shell to perform a particular task (see the manual "POSIX Commands").

**command interpreter**

An interface that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. It is possible for applications to invoke utilities through a number of interfaces, which are collectively considered to act as command interpreters. The most obvious of these are the `sh` utility and the `system()` function, although `popen()` and the various forms of `exec` may also be considered to behave as interpreters.

**control character**

A character, other than a graphic character, that affects the recording, processing, transmission or interpretation of text.

**controlling process**

The session leader that established the connection to the controlling terminal. If the terminal ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process.

**controlling terminal**

A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal.

**core dump**

An image of the memory area occupied by a specific process. If the process is aborted, the core dump is written to the file `core`.

**current (or working) directory**

A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash (`/`).

**daemon**

A background process that performs its activities silently once started and terminates only when the system is shut off. The best known UNIX example is the printer daemon, which handles the printing of a file in the background while the user continues working.

### **data set pointer (file pointer)**

A data set pointer (also called a file pointer) is a pointer to a structure of type `FILE`. It is used to process a file with the standard access functions (see `stdio.h`). When a file is opened with `fopen()`, `fdopen()`, or `freopen()`, it is assigned a file pointer, which serves as a file argument when the file is subsequently accessed using `fprintf()`, `fscanf()`, `fclose()`, etc. At program startup, the standard I/O files are automatically opened with the following file pointers: `stdin` (standard input), `stdout` (standard output), `stderr` (standard error).

### **default**

Normal method by which a program is executed when no additional specifications are made.

### **device**

A computer peripheral or an object that appears to the application as such.

### **device ID**

A non-negative integer used to identify a device.

### **directory**

A file that contains directory entries with unique names (see `file name`). Directories are used to organize files and other directories into a hierarchical system.

### **directory entry (or link)**

An object that associates a file name with a file. Several directory entries can associate names with the same file.

### **directory stream**

A per-process unique value used to reference an open directory.

### **display (on-screen)**

Output to the terminal device file. The output appears on the screen of the monitor. If the output is not directed to a terminal, the results are undefined. The terms "display" and "write" are clearly differentiated in the XPG4 standard. When the term "display" is used, the method of outputting to the terminal is unspecified; `termcap` or `terminfo` is frequently used for this purpose, but this is not a requirement. The term "write" is reserved for cases when a file descriptor is used and the output can be redirected. However, when the writing is directly to the terminal (i.e. has not been redirected elsewhere), there is no practical way for a user or test suite to determine whether a file descriptor is being used or not. Therefore, the use of a file descriptor is mandated only for the redirection case.

**dot**

A file name consisting of a single dot character (.); it represents the current working directory (see `pathname resolution`).

**dot-dot**

A file name consisting solely of two dot characters (..); it represents the parent directory (see `pathname resolution`).

**downshifting**

The conversion of uppercase characters to their corresponding lowercase representations.

**effective group ID**

An attribute of a process that is used in determining various permissions, including file access permissions (see `group ID`). This value is subject to change during the process lifetime, as described under `setgid()` and the `exec` family of functions.

**effective user ID**

An attribute of a process that is used in determining various permissions, including file access permissions (see `user ID`). This value is subject to change during the process lifetime, as described under `setuid()` and `exec`.

**elementary functions**

*BS2000:*

Functions that process a file on the basis of file descriptors are referred to as "elementary". This is in contrast to the standard I/O functions, all of which operate on the basis of file pointers. In addition, the elementary functions allow SAM files to be processed only as text files, whereas with the standard functions they can also be processed as binary files.

In UNIX/POSIX, elementary functions are implemented in the form of system calls, which differ from standard functions by virtue of improved performance and greater operating system support. No such distinction is made between a system call and a function in BS2000.

**empty directory**

A directory that contains, at most, directory entries for `.` and `..` (see `dot` and `dot-dot`).

**empty string**

A string whose first byte is a null byte.

**empty wide character string**

A wide character string whose first element is a null wide-character code.

### **epoch**

The time zero hours, zero minutes, zero seconds, on January 1, 1970 (Coordinated Universal Time).

*BS2000:*

The time zero hours, zero minutes, zero seconds, on January 1, 1950.

### **equivalence class**

A set of collating elements with the same primary collation weight. The following letters, for example, constitute an equivalence class, since they are all based on the same base letter and differ only in terms of their accents: á, à, â, ä, ã, å. The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight.

### **executable file**

A regular file which is accepted as a new process image by the `exec` family of functions, which has execute permission, and can thus be called as a command or utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files can also be provided. The internal format of an executable file is unspecified, but a conforming application can detect that an executable file is not a text file.

### **expression**

A mathematical or logical symbol or a meaningful combination of such symbols.

### **extended security controls**

The access control (see `file access permissions`) and privilege (see `appropriate privileges`) mechanisms have been defined to allow implementation-dependent extended security controls. These permit an implementation to provide security mechanisms that differ from those from those described in the XPG4 standard. These mechanisms do not alter or override the defined semantics of any of the functions described in this manual.

### **feature test macro**

A macro used to determine whether a particular set of features will be included from a header.

### **FIFO special file**

A type of file from which data is read on a first-in-first-out basis. Other properties of FIFO special files are described under `lseek()`, `open()`, `read()`, and `write()`.

**file**

An object that can be written to, or read from, or both. A file is identified in UNIX by means of an inode has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file and directory. A regular file contains text, data, programs or other information. A special file refers to a device or a part of a physical device such as a drive or hard disk partition. A directory contains other files.

*BS2000:*

Records that are related to one another are combined into a named unit (i.e. a file). Typical files include conventional I/O data of programs, load modules, and plaintext information that can be created and edited with an editor.

**file access permissions**

Part of the open file description. The file permission bits are used for file access control. These bits are set at the time of file creation by functions such as `open()`, `creat()`, `mkdir()` and `mkfifo()` and are changed by `chmod()`. The bits are read by `stat()` or `fstat()`.

Applications may provide additional or alternate file access control mechanisms, or both. An alternate file access control mechanism must have the following features:

- It must specify file permission bits for the file owner class, file group class, and file other class of the file.
- It must be enabled only by explicit user action, on a per-file basis by the file owner or a user with the appropriate privileges.
- It may be disabled for a file after the file permission bits are changed for that file with `chmod()`. The disabling of the alternate mechanism need not disable any additional mechanisms defined by the implementation.

Whenever a process requests file access permission for a read, write, or execute/search operation, access is determined as described below (provided no additional mechanism denies access):

If a process has appropriate privileges:

- If read, write or directory search permission is requested, access is granted.
- If execute permission is requested, access is granted if execute permission is granted to at least one user by the file permission bits or by an alternate access control mechanism; otherwise, access is denied.

If a process does not have appropriate privileges:

- The file permission bits of a file contain read, write and execute/search permissions for the file owner class, file group class and file other class.
- Access is granted if an alternate access control mechanism is not enabled and the requested access permission bit is set for the class (file owner class, file group class, or file other class) to which the process belongs, or if an alternate access control mechanism is enabled and it allows the requested access; otherwise, access is denied.

### **file description**

An object that contains information on how a process or group of processes are accessing a file. Each file descriptor refers to exactly one open file description, but an open file description can be referred to by more than one file descriptor. A file offset, file status and file access modes are attributes of an open file description.

### **file descriptor**

A per-process unique, positive integer used to establish a unique association between a process and an open file for the purpose of file access. The value of a file descriptor is from zero to `{OPEN_MAX}`. A process can have no more than `{OPEN_MAX}` file descriptors open simultaneously. File descriptors may also be used to implement message catalog descriptors and directory streams. See *open file description* in this Glossary and `{OPEN_MAX}` in the `limits.h` header.

### **file group class**

A process is in the file group class of a file if the process is not in the file owner class and if the effective group ID or one of the supplementary group IDs of the process matches the group ID associated with the file. Other conformant implementations may specify different members for this class.

### **file hierarchy**

Files in the system are organized in a hierarchical tree structure in which all of the non-terminal nodes (branches) are directories and all of the terminal nodes (leaves) are any type of file. Multiple directory entries may refer to the same file.

### **file mode**

A combination of attributes that specify the file type and the access permissions of a file (see the header file `sys/stat.h`).



**file name**

A name consisting of 1 to `{NAME_MAX}` bytes used to name a file. The characters composing the name may be selected from the set of all character values excluding the slash character (`/`) and the null byte (`\0`). The file names `.` (dot) and `..` (dot-dot) have special meaning; see `pathname resolution`. File names are constructed from the portable file name character set, since the use of other characters can be confusing or ambiguous in certain contexts. For example, the use of a colon (`:`) in a pathname could cause ambiguity if that pathname were included in a `PATH` definition (see `portable file name character set`).

**file offset**

The file offset specifies the byte position in the file, i.e. the number of bytes from the start of the file (byte 1 = 1), where the next I/O operation begins. Each open file description associated with a regular file, block special file or directory has a file offset. A character special file that does not refer to a terminal device may have a file offset. There is no file offset specified for a pipe or FIFO.

**file other class**

The property of a file indicating access permissions for a process related to the user and group identification of a process. A process is in the file other class of a file if the process is not in the file owner class or file group class.

**file owner class**

The property of a file indicating access permissions for a process related to the user identification of a process.

A process is in the file owner class of a file if the effective user ID of the process matches the user ID of the file. Other conformant implementations may specify different members for this class.

**file permission bits**

Information about a file that is used, along with other information, to determine if a process has read, write or execute/search permission to a file. The bits are divided into three parts: owner, group and other. Each part is used with the corresponding file class of processes. These bits are contained in the file mode, as described under `sys/stat.h`. The detailed usage of the file permission bits in access decisions is described under `file access permissions`.

**file position indicator**

The file position indicator contains information on the current file position. Data is read from or written to the file from this current position onwards. The structure of the information contained in the file position indicator varies in accordance with the type of file.

For text files, it contains information on the current record and the position within that record.

*BS2000:*

For binary files with stream I/O, it contains the byte offset, i.e. the number of bytes calculated from the beginning of the file. The structure differs for SAM and ISAM files. This information is used internally by the runtime system.

For binary files with record I/O, it contains information on the position after the last record to be read, written or deleted, or the position reached by an directly preceding seek operation.

For ISAM files with duplicate keys, it contains the position after the last record of a group having identical keys if one of these records was read, written or deleted earlier.

### **file serial number**

A per-file-system unique identifier for a file.

### **file status**

The current status of a file.

### **file structure**

As soon as a file is opened with `fopen()`, `fdopen()` or `freopen()`, it is automatically assigned a specific structure of type `FILE`. This structure is defined in `stdio.h` and includes, among other things, the following information on the file: pointer to the I/O buffer, buffer size, location of the file position indicator, and the size of the file.

### **file system**

A collection of files and certain of their attributes. A UNIX file system is organized in a hierarchical structure (see *file hierarchy*). A file system provides a name space for file serial numbers referring to the files in it.

### **file times update**

Each file has three associated time values that are updated when file data has been accessed, file data has been modified, or file status has been changed, respectively. These values are returned in the file characteristics structure `stat` (see `sys/stat.h`).

For each function in this manual that reads or writes file data or changes the file status, the appropriate time-related fields are noted as "marked for update". At the time of an update, all marked fields are set to the current time, and the update marks are cleared. Two such update times are when the file is no longer open by any process and when `stat()` or `fstat()` is performed on the file. Additional update times are unspecified. Updates are not done for files on read-only file systems.

**filter**

A command with which data is read from standard input or a list of input files and written to standard output. Typically, its function is to perform some transformation on the data stream.

**foreground**

Normal method of executing a command in a shell. When a command is executed in the foreground, the shell waits for that command to complete before prompting the user for further input.

**foreground process**

A process that is a member of a foreground process group.

**foreground process group**

A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal.

**foreground process group ID**

The process group ID of the foreground process group.

**form-feed character**

A character that in the output stream indicates that printing should start on the next page of an output device. The form-feed is the character designated by `\f` in the C language. If the form-feed is not the first character of an output line, the result is unspecified. It is likewise unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page.

**group database**

A system database of implementation-dependent format that contains at least the following information for each group ID: group name, numerical group ID, and a list of users allowed in the group. The list of users allowed in the group is used by the `newgrp` utility.

**group ID**

A non-negative integer that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, a supplementary group ID or a saved set-group-ID.

**group name**

A string that is used to identify a group, as described in group database. To be portable across XSI-conformant systems, the value must be composed of characters from the portable file name character set. The hyphen should not be used as the first character of a portable group name.

**header file (include file)**

The file containing data definitions which are copied by the compiler into source files (see `library`). Header file names end with the suffix `.h`. Header files are included in source files by means of an `#include` statement. Consequently, they are also referred to as include files.

**home directory**

The directory in which a user is automatically placed when connected with POSIX.

**host**

A central computer in a network. A host is the system on which programs are executed, files are stored, and I/O is controlled. Large powerful networks may often have several hosts.

**internationalization**

The provision, within a computer program, of the capability of making itself adaptable to the requirements of different native languages, local customs and coded character sets.

**interrupt**

An interruption in the normal processing of a program. Interrupts are caused by signals which are triggered by a hardware state of a peripheral device to indicate a particular status. If the interrupt is detected by the hardware, an interrupt service routine is executed. An interrupt character is usually an ASCII character that generates an interrupt when it is entered from the keyboard.

**job control**

A facility that allows users selectively to stop (or suspend) the execution of processes and continue (or resume) their execution at a later point. The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter.

**job control ID**

A handle that is used to refer to a job. The job control job ID can have any of the following forms:

| Job control job ID | Meaning                                     |
|--------------------|---------------------------------------------|
| %%                 | Current job                                 |
| %+                 | Current job                                 |
| %-                 | Previous job                                |
| % <i>n</i>         | Job number <i>n</i>                         |
| % <i>string</i>    | Job whose command begins with <i>string</i> |
| %? <i>string</i>   | Job whose command contains <i>string</i>    |

**job variable**

*BS2000:*

Job variables are named memory areas that are used for mutual data exchanges among jobs and for the exchange of information between jobs and the operating system. Each job variable has a name and a value (its content). The content can be used to control jobs and programs. Job variables can be created, modified, queried and deleted by the user. In addition, users can instruct the operating system to set a monitoring job variable to reflect changes in the status of a job or a program.

**join file (user catalog)**

A file containing the user attributes of all user IDs of a subset or system.

**kernel**

The code of the POSIX/UNIX operating system.

**library**

A collection of statically linked object files or source files that can be linked in dynamically (shareable library). The individual files of a library contain the program text for one or more related functions. When a relevant function is called in the source code, the corresponding object file must be linked into the program (see `header file`). The name of the library containing it must be specified at linkage. The file containing the library function used is then copied into the source code of the application.

**link**

See `directory entry`.

**link count**

The number of directory entries that refer to a file is called the link count of the file.

**local machine**

As far as the user is concerned, the local machine is always the one on which he or she is working. All other computers on the network are remote computers for that user.

**locale**

The conventions of a geographic area or territory for date, time, and currency formats.

**locale (country-specific)**

The definition of the subset of a user's environment that depends on language and cultural conventions.

**localization**

The process of establishing information within a computer system specific to the operation of particular native languages, local customs and coded character sets.

**login name**

*BS2000:*

A user name of up to 8 characters that is entered in the join file. The login name is the basis on which the user is identified on gaining access to the system. All files and job variables are created under a login name. The names of files and job variables are stored together with a login name in the file catalog.

**mathematical range**

The notation  $[n, m]$  and  $(n, m)$  denotes a mathematical range. The square brackets [ and ] include the respective limits; the parentheses ( and ) exclude them. Thus, if  $x$  is in the range  $[0, 1]$ , it can be from 0 to 1 inclusive, but if  $x$  is in  $(0, 1)$ , it can be from 0 up to but not including 1.

**memory area**

A restricted (and defined) area of working memory that can be assigned to specific programs and arbitrarily subdivided in accordance with program requirements.

**message catalog**

A file or storage area containing program messages, command prompts and responses to prompts for a particular native language, territory and codeset.

**message catalog descriptor**

A per-process unique value used to identify an open message catalog.

**mode**

A collection of attributes that specifies a file's type and its access permissions (see `file access permissions`).

**mount point**

Either the system root directory or a directory for which the `st_dev` field of structure `stat` (see `sys/stat.h`) differs from that of its parent directory.

**multi-byte character**

Characters that consist of multiple bytes, regardless of whether a normal character or wide character code is involved.

**NaN (not a number)**

A value that can be stored in a floating type but that is not a valid floating point number. One example of such a bit pattern is a floating-point number whose exponent bits are all set to 1.

**newline character**

A character that in the output stream indicates that printing should start at the beginning of the next line. The newline is the character designated by `\n` in the C language. If the newline is not the first character of an output line, the result is unspecified. It is likewise unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line.

**null byte**

A byte with all bits set to zero.

**null pointer**

The value that is obtained by converting the number 0 into a pointer; for example, `(void *) 0`. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers use it to indicate an error.

**object file**

A file that contains the source code of a program in binary representation. A relocatable object file contains references that have not been resolved by associations with the corresponding definitions; an executable object file is a linked program.

**open file**

A file that is currently associated with a file descriptor.

**option**

An argument to a command that affects the execution of that command. An option is a type of argument that follows the command name and usually precedes the other arguments on the command line. Options normally begin with a minus sign. The number and types of arguments allowed vary for different commands. If options also take arguments, the arguments are separated by spaces.

**option-argument**

A parameter that follows certain options. In some cases, an option-argument is included within the same argument string as the option; in most cases, it is the next argument.

**orphaned process group**

A process group in which the parent of every member is either itself a member of the group or is not a member of the group's session.

**parent directory**

The directory containing a directory entry for the file in question. This concept does not apply to `.` and `..` (dot and dot-dot).

**parent process**

See `process`.

**parent process ID**

A new process is created by a currently active process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of the `init` process.

**parser**

A parser performs a syntactic and lexical analysis of a text.

**password**

A sequence of characters that must be entered by the user to gain access to a user ID, a file, a job variable, a network node, or an application.



**pathname**

A character string that is used to identify a file. A pathname consists of, at most, `{PATH_MAX}` bytes, including the terminating null byte. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the pathname refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A pathname that begins with two successive slashes may be subject to special interpretation by some compatible implementations, although more than two leading slashes are treated as a single slash (see `pathname resolution`).

*BS2000:*

Every cataloged file in BS2000 can also be uniquely identified by a pathname. The pathname is composed of the catalog ID (`catid`), the user ID (`userid`), and a fully-qualified file name assigned by the user (e.g.: `catid:$userid.filename`).

**pathname prefix**

A pathname that begins with an optional slash and points to a directory.

**pathname resolution**

Pathname resolution is performed for a process to resolve a pathname to a particular file in a file hierarchy. There may be multiple pathnames that resolve to the same file.

Each file name in the pathname is located in the directory specified by its predecessor (for example, in the pathname fragment `a/b`, file `b` is located in directory `a`). Pathname resolution fails if this cannot be accomplished.

If the pathname begins with a slash, the predecessor of the first file name in the pathname is taken to be the root directory of the process. Such pathnames are referred to as absolute pathnames.

If the pathname does not begin with a slash, the predecessor of the first file name of the pathname is taken to be the current working directory of the process. Such pathnames are referred to as relative pathnames.

The interpretation of a pathname component is dependent on the values of `{NAME_MAX}` and `{_POSIX_NO_TRUNC}` associated with the path prefix of that component. If any pathname component is longer than `{NAME_MAX}`, and if `{_POSIX_NO_TRUNC}` is in effect for the path prefix of that component (see `pathconf()`), this is considered an error condition. Otherwise, only the first `{NAME_MAX}` bytes of the pathname component are taken into account. The special file name `.` (dot) refers to the directory specified by its predecessor. The special file name `..` (dot-dot) refers to the parent directory of its predecessor. As a special case, in the root directory, dot-dot may refer to the root directory itself. A pathname consisting of a single slash resolves to the root directory of the process. A null pathname is invalid.

## **pattern**

A sequence of characters used either with regular expression notation or for pathname expansion as a means of selecting various character strings or pathnames, respectively. The syntaxes of the two patterns are similar, but not identical. This manual always indicates the type of pattern being referred to in the immediate context of the use of the term.

## **pipe**

An object accessed by one of the pair of file descriptors created by the `pipe()` function. Once created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy.

## **portability**

The capability of a program to run on different operating systems without changes. This is achieved by using standardized open programming interfaces that are offered on a variety of platforms.

## **portable character set**

The collection of characters that are required to be present in all locales supported by XSI-conformant systems:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z  
a b c d e f g h i j k l m n o p q r s t u v w x y z  
0 1 2 3 4 5 6 7 8 9 ! # % ^ & * ( ) _ + - = { } [ ]  
: " ~ ; ' , < > ? , . | \ / @ $
```

## **portable file name character set**

For a file name to be portable across implementations conforming to the ISO POSIX-1 standard, it must consist only of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z  
a b c d e f g h i j k l m n o p q r s t u v w x y z  
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore and hyphen characters, respectively.

The hyphen must not be used as the first character of a portable file name. Uppercase and lowercase letters are differentiated by all conforming implementations.

In the case of a portable pathname, the slash character may also be used.

## **portable pathname**

For a pathname to be portable across compatible systems, it should consist of at most `{PATH_MAX}` bytes, including the terminating null byte. It should be a pathname consisting of an optional leading slash, followed by zero or more portable file names separated by slashes.

**POSIX file system**

A file system in BS2000 with the structure of a UNIX file system (UFS). The POSIX file system comprises a set of directories and files (POSIX files) that are organized in a hierarchical tree structure. The root directory (/) is at the root of the tree, and all other directories are the branches from the root directory. Each file in the file system can be reached via precisely one absolute path and several conceivable relative paths.

The difference between a POSIX file system and a UNIX file system is the storage location: a UNIX file system is stored on a physical device, whereas a POSIX file system is stored in a PAM container file.

**POSIX shell**

A ported SINIX system program that handles communication between the user and the system. The POSIX shell is a command interpreter. It translates the entered POSIX commands into a language that can be processed by the system.

If the POSIX shell was entered as the "program" attribute of the user, the POSIX shell will be started automatically when the user logs on at a remote computer (`rlogin`).

**process**

An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process by a call to the `fork()` function. The process that calls `fork()` is known as the parent process, and the new process created by the `fork()` is known as the child process.

**process group**

A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group that is identified by a process group ID. This grouping permits signals to be sent to related groups of processes. A newly created process joins the process group of its creator.

**process group ID**

The unique identifier representing a process group during its lifetime. A process group ID is a positive integer and cannot be reused by the system until the process group lifetime ends.

**process group leader**

A process whose process ID is the same as its process group ID.

### **process group lifetime**

A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, due either to the end of the last process lifetime or to the last remaining process calling the `setsid()` or `setpgid()` functions.

### **process ID**

A unique identifier of a process. A process ID is a positive integer that cannot be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID cannot be reused by the system until the process group lifetime ends. Only a system process can have a process ID of 1.

### **process lifetime**

The period of time that begins when a process is created and ends when its process ID is returned to the system.

After a process is created with a `fork()` function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a `wait()`, or `waitpid()` function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

### **protocol**

A set of rules for the exchange of data between two systems. The protocol defines the type of electrical connection, the data format, and the sequence of data.

### **pthread**

A thread is a part of a program which runs concurrently with other parts. Several threads can run concurrently within a single process. A process must, however, comprise at least one thread. Unlike processes, all the threads of a program share a common address space.

In the case of the Pthreads in BS2000, the threads of a single process can be distributed over several tasks, unlike DCE threads, for instance.

### **radix character**

The character that separates the integer part of a number from the fractional part.

**read-only file system**

A file system that has implementation-dependent characteristics restricting modifications.

**real group ID**

The attribute of a process that, at the time of process creation, identifies the group of the user who created the process (see `group ID`). This value is subject to change during the process lifetime, as described under `setgid()`.

**real user ID**

The attribute of a process that, at the time of process creation, identifies the user who created the process (see `user ID`). This value is subject to change during the process lifetime, as described under `setuid()`.

**record-oriented I/O**

*BS2000:*

Record-oriented I/O means that the file position indicator of the file can only be positioned at the start of a record or block. Record-oriented I/O enables efficient file processing, adapted to the structure of the BS2000 system. The unit for an I/O function call is always a record or block. Record-oriented processing can be used for cataloged SAM, ISAM and PAM files. Additional functions are available for actions such as deleting or inserting records or accessing keys in ISAM files.

**regular expression**

A pattern constructed according to specific rules (see section "Regular expressions" in the manual "POSIX Commands").

**regular file**

A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system.

**relative pathname**

An access path for a file or directory, starting from the position of the current directory within the file system. Relative pathnames do not begin with a slash (/) (see `pathname resolution`).

**remote machine**

In a local network, a distinction is made between the local computer and the remote machines. As far as the user is concerned, all computers in the network other than the one at which he or she is directly working are remote machines. The user can communicate with all remote machines on the network.

### **root directory**

A directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash.

### **saved set-group-ID**

An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described under `setgid()` and `exec`.

### **saved set-user-ID**

An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described under `setuid()` and `exec`.

### **security attributes**

*BS2000:*

The attributes of an object (file, job variable, etc.) which define and control access to that object and are thus relevant to security.

For example, the following security attributes exist for files: ACCESS/USER-ACCESS, SERVICE bit, AUDIT attribute, RDPASS, WRPASS, EXPASS, RETPD, BACL, ACL and GUARD.

### **session**

A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership (see `setsid()`). Implementations that support `setpgid()` can have multiple process groups in the same session.

### **session leader**

A process that has created a session (see `setsid()`).

### **session lifetime**

The period between when a session is created and the end of the lifetime of all the process groups that remain as members of the session.

### **shell**

A system program in UNIX that handles communication between the user and the system. The shell is a command interpreter. It translates the entered commands into a language that can be processed by the system. A shell is started for each user as soon as he or she has logged on to the system.

**signal**

A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term signal is also used to refer to the event itself.

**signal mask**

The currently defined set of signals for a process that are to be blocked before the signal is delivered to that process. The signal mask of a process is initialized by its parent process. The signal mask can be controlled and manipulated with the `sigaction()`, `sigprocmask()` and `sigsuspend()` functions.

**slash**

The term slash is used to represent the literal character `/`, also known as a `solidus`.

**special character**

Characters that are assigned special functions on I/O (see section "General terminal interface" on page 94).

**special file**

A file, also called a device driver, that serves as the interface to an I/O device such as a terminal, disk drive, or line printer.

**standard error**

An output stream used for diagnostic messages.

**standard input**

A stream associated with a primary input device.

**standard output**

A stream associated with a primary output device.

**standard utilities**

The commands described in the manual "POSIX Commands (BS2000/OSD)".

**stream**

A file access object that allows access to an ordered sequence of characters. Such objects can be created by the `fdopen()`, `fopen()` or `popen()` functions and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output.

### **stream-oriented I/O**

Stream-oriented I/O means that the file position indicator can be positioned on each individual byte in the file. Stream I/O is the conventional processing mode and is set by default, i.e. without any special qualifiers specified for the open functions. Text files can be processed exclusively in this I/O mode. In contrast to record-oriented I/O, the data for output to files with stream I/O is first stored in an internal buffer and is written to the external file later (see [buffering](#)).

### **supplementary group ID**

An attribute of a process used in determining file access permissions. A process has up to `{NGROUPS_MAX}` supplementary group IDs in addition to the effective group ID. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created. Whether a process' effective group ID is included in or omitted from its list of supplementary group IDs is unspecified.

### **suspended job**

A background job that has received a `SIGSTOP`, `SIGTSTP`, `SIGTTIN` or `SIGTTOU` signal.

### **system**

The term system is used in this manual to designate an implementation of the system interface.

### **system call**

Request, from within a program, for a service that is executed by the operating system kernel.

### **system process**

An object, other than a process executing an application, that is defined by the system and has a process ID.

### **system scheduling priority**

A number used as advice to the system to alter process scheduling priorities. Raising the value gives the process additional preference when it is scheduled to run; lowering the value reduces the preference.

### **terminal**

A character special file (i.e. a special file for a character-oriented device) that meets the specifications of the general terminal interface (see section "General terminal interface" on page 94).



**text file**

*BS2000:*

Text files are only possible for stream I/O. The following file types are treated as text files:

- cataloged SAM files (no binary mode on open),
- cataloged ISAM files,
- system files (SYSDTA, SYSOUT, SYSLST, SYSTEM)

A text file is an ordered sequence of bytes that are combined to form lines (or records). In contrast to binary files, the control characters for white space are converted to their appropriate effect, depending on the type of text file (see `white space`). This means that data read from a text file does not correspond precisely to the data that was originally written to it. Each written tab (`\t`) that is read is expanded to an appropriate number of spaces. The following points also apply to text files:

- Newline characters not originally written to the file may be read in (see `fflush()`, `fseek()`, `fsetpos()`, `lseek()`, `rewind()`).
- Output to `SYSOUT` and `SYSTEM` (for writing)  
Each line is started with a blank as a print control character. This produces a line feed.
- Output to `SYSLST`  
The line starts with a blank as the print control character only if none of the control characters `\f`, `\v`, `\r` or `\b` are specified in a line.

**UNIX system**

An operating system that works in interactive mode. UNIX was developed in 1969 by Bell Laboratories. Since only a central system kernel of this operating system is hardware-dependent, UNIX is installed on several different systems by various computer manufacturers. The Siemens Nixdorf version of UNIX is called SINIX. UNIX applications are portable to a large extent.

**upshifting**

The conversion of lowercase characters to their corresponding uppercase representations.

**user**

A representative of a user ID. The term user is used generically for people, applications, procedures, etc., that can obtain access to the operating system via a user ID.

**user administration**

*BS2000:*

All privileges that can be assigned with the command `/SET-PRIVILEGE` as well as the privileges of the security administrator and the system ID TSOS.

**user attributes**

All characteristics of a user ID that are stored in the join file (also called a user catalog).

**user catalog**

See `join file`.

**user database**

A system database of implementation-dependent format that contains at least the following information for each user ID:

user name, numerical user ID, initial numerical group ID, initial working directory, and initial user program.

The initial numerical group ID is used by the `newgrp` utility. Any other circumstances under which the initial values are made effective are implementation-dependent.

**user group**

A collection of individual users under a single name (group ID).

**user ID**

A non-negative integer that is used to identify a system user. When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or a saved set-user-ID.

**user name**

A string that is used to identify a user, as described in user database. To be portable across XSI-conformant systems, the value must be composed of characters from the portable file name character set. The hyphen should not be used as the first character of a portable user name.

**user privileges**

*BS2000:*

All attributes assigned to a user ID (login name), which are stored in the join file and which define the rights of the user.

**variable**

An object with a value that may change during program execution.

**white space**

A sequence of one or more characters that belong to the `space` character class as defined via the `LC_CTYPE` category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tabs), newline characters, carriage-return characters, form-feed characters and horizontal or vertical tab characters.

**wide character code**

An integer value corresponding to a single graphic symbol or control code. All wide character codes of a process consist of the same number of bits. A wide character code for which all bits are set to zero is called a null wide character code.

**wide character string**

A contiguous sequence of wide character codes terminated by and including the first null wide character code.

**zombie process**

An inactive process that will be deleted at some later time when its parent process executes a `wait()` or `waitpid()` function.



---

# Related publications

## Manuals of Fujitsu-Siemens Computers

### Ordering manuals

Please apply to your local office for ordering the manuals.

- [1] POSIX in BS2000/OSD  
**Gateway to the UNIX World**  
Summary Description

- [2] **POSIX** (BS2000/OSD)  
POSIX Basics for Users and System Administrators  
User Guide

*Target group*

BS2000 system administrators, POSIX administrators, BS2000 users,  
users of UNIX/SINIX workstations

*Contents*

- Introduction to and working with POSIX
- BS2000 software products in a POSIX environment
- Installing POSIX
- Controlling POSIX and administering file systems
- Administering POSIX users
- BS2000 commands for POSIX

- [3] **POSIX V1.1A** (BS2000/OSD)  
Commands  
User Guide

*Target group*

This manual addresses all users of the POSIX shell.

*Contents*

This manual is designed as a work of reference. It describes working with the POSIX shell and the commands of the POSIX shell in alphabetical order.

- [4] **C (BS2000)**  
**C Compiler**  
User Guide

*Target group*

C users in a BS2000 environment.

*Contents*

- Description of all activities concerned with the creation of an executable C program: compilation, linking, loading, debugging
- Programming notes and additional information on: program flow control, linking functions and languages, language scope of the C compiler

- [5] **C/C++ V3.0B (BS2000/OSD)**  
C/C++ Compiler  
User Guide

*Target group*

C and C++ users in a BS2000 environment.

*Contents*

- Description of all activities in the creation of executable C and C++ programs: compilation, linking, loading, debugging
- Programming notes and detailed information on: optimization, program flow control, linking of functions and languages, C and C++ language scope of the compiler.

- [6] **C/C++ V3.0B (BS2000/OSD)**  
POSIX Commands of the C/C++ Compiler  
User Guide

*Target group*

C and C++ users in a BS2000 environment.

*Contents*

- Introduction to C/C++ program development in the POSIX shell environment.
- Compiling and linking of C and C++ programs with the POSIX commands `cc`, `c89` and `CC`.
- Controlling the global C and C++ list generator with the POSIX command `cclistgen`.

- [7] **C Library Functions (BS2000/OSD)**

*Target group*

The manual is intended for C and C++ users in a BS2000/OSD environment.

*Contents*

- Descriptions of all C functions and macros which can be used in the BS2000 system without POSIX.
- Basic information, programming notes and examples on: BS2000 file processing, STXIT and contingency routines, locality.

- [8] **CRTE (BS2000/OSD)**  
Common RunTime Environment  
User Guide
- Target group*  
This manual addresses all programmers and system administrators in a BS2000 environment.
- Contents*  
It describes the common runtime environment for COBOL85, COBOL2000, C and C++ objects and for "language mixes":
- CRTE components
  - ILCS program communication interface
  - linkage examples
- [9] **DCE (BS2000)**  
POSIX Program Interface  
User Guide
- Target group*  
Programmers of DCE applications
- Contents*  
Description of the C library functions with POSIX functionality which are required for programming DCE applications in BS2000/OSD V1.0.
- [10] **SDF-P (BS2000/OSD)**  
**Programming in the Command Language**  
User Guide
- Target group*  
The manual addresses BS2000/OSD users and systems support.
- Contents*  
SDF-P is a structured procedure language in BS2000. The introduction is followed by a detailed description of commands, functions and macros.
- [11] **BS2000/OSD**  
Executive Macros  
User Guide
- Target group*  
The manual addresses all BS2000/OSD assembly language programmers.
- Contents*  
The manual contains a summary of all Executive macros, detailed descriptions of each macro with notes and examples, including job variable macros, and a comprehensive general training section.

- [12] **BS2000/OSD-BC**  
Introductory Guide to DMS  
User Guide

*Target group*

This manual is addressed to nonprivileged users and systems support staff.

*Contents*

It describes file management and processing in BS2000.

Attention is focused on the following topics:

- volumes and files
- file and catalog management
- file and data protection
- OPEN, CLOSE and EOVS processing
- DMS access methods (SAM, ISAM,...)

The most important new features in OSD-BC V3.0 are the SMS concept and the XCS network.

- [13] **JV (BS2000/OSD)**  
Job Variables  
User Guide

*Target group*

The manual addresses both nonprivileged users and systems support.

*Contents*

The manual describes management and possible uses of job variables. The command descriptions are divided according to function areas. The macro calls are described in a separate chapter.



## Other References

**X/Open CAE Specification**

System Interfaces and Headers, Issue 4 , Version 2

ISBN: 1-85912-037-7

X/Open Document Number: C435

**X/Open CAE Specification**

System Interface Definitions, Issue 4

ISBN: 1-872630-46-4

X/Open Document Number: C204

**X/Open CAE Specification**

Commands and Utilities, Issue 4

ISBN: 1-872630-48-0

X/Open Document Number: C203



---

# Index

#define statement 14  
#include directive 12  
\*key\_t 1019  
. 1061  
.. 1061  
/var/adm/utmpx 253  
\_\_DATE\_\_ 235  
\_\_FILE\_\_ 300  
\_\_LINE\_\_ 518  
\_\_STDC\_VERSION\_\_ 760  
\_\_STDC\_\_ 760  
\_\_TIME\_\_ 829  
\_CS\_LFS\_CFLAGS 1037  
\_CS\_LFS\_LDFLAGS 1037  
\_CS\_LFS\_LINTFLAGS 1037  
\_CS\_LFS64\_CFLAGS 1037  
\_CS\_LFS64\_LDFLAGS 1037  
\_CS\_LFS64\_LINTFLAGS 1037  
\_CS\_PATH 220, 1037  
\_edt 248  
\_exit 264, 1040  
\_IOFBF 985  
\_IOLBF 985  
\_IONBF 985  
\_LFS\_LIBS 1037  
\_LFS\_LIBS64 1037  
\_LFS64\_ASYNCHRONOUS-IO 1036  
\_LFS64\_LARGEFILE 1036  
\_LFS64\_STDIO 1036  
\_longjmp 538  
\_PC\_CHOWN\_RESTRICTED 617, 1039  
\_PC\_LINK\_MAX 617, 1039  
\_PC\_MAX\_CANON 617, 1039  
\_PC\_MAX\_INPUT 617, 1039  
\_PC\_NAME\_MAX 617, 1039  
\_PC\_NO\_TRUNC 617, 1039  
\_PC\_PATH\_MAX 617, 1039  
\_PC\_PIPE\_BUF 617, 1039  
\_PC\_VDISABLE 617, 1039  
\_POSIX\_ARG\_MAX 961  
\_POSIX\_CHILD\_MAX 961  
\_POSIX\_CHOWN\_RESTRICTED 617, 1035  
\_POSIX\_FSYNC 805  
\_POSIX\_JOB\_CONTROL 806, 1036  
\_POSIX\_LINK\_MAX 961  
\_POSIX\_MAPPED\_FILES 805  
\_POSIX\_MAX\_CANON 961  
\_POSIX\_MAX\_INPUT 961  
\_POSIX\_MEMLOCK 805  
\_POSIX\_MEMLOCK\_RANGE 805  
\_POSIX\_MEMORY\_PROTECTION 806  
\_POSIX\_MESSAGE\_PASSING 806  
\_POSIX\_NAME\_MAX 961  
\_POSIX\_NGROUPS\_MAX 961  
\_POSIX\_NO\_TRUNC 1035  
\_POSIX\_OPEN\_MAX 961  
\_POSIX\_PATH\_MAX 961  
\_POSIX\_PIPE\_BUF 961  
\_POSIX\_PRIORITIZED\_IO 806  
\_POSIX\_PRIORITY\_SCHEDULING 806  
\_POSIX\_REALTIME\_SIGNALS 806  
\_POSIX\_SAVED\_IDS 806, 1036  
\_POSIX\_SEMAPHORES 806  
\_POSIX\_SHARED\_MEMORY\_OBJECTS 806  
\_POSIX\_SSIZE\_MAX 961  
\_POSIX\_STREAM\_MAX 961  
\_POSIX\_SYNCHRONIZED\_IO 806  
\_POSIX\_THREAD\_ATTR\_STACKADDR 806  
\_POSIX\_THREAD\_ATTR\_STACKSIZE 806

`_POSIX_THREAD_PRIO_INHERIT` 806  
`_POSIX_THREAD_PRIO_PROTECT` 806  
`_POSIX_THREAD_PRIORITY_SCHEDULING`  
806  
`_POSIX_THREAD_PROCESS_SHARED` 806  
`_POSIX_THREAD_SAFE_FUNCTIONS` 806  
`_POSIX_THREADS` 806  
`_POSIX_TIMERS` 806  
`_POSIX_TZNAME_BUF` 961  
`_POSIX_VDISABLE` 617, 1035  
`_POSIX_VERSION` 806, 1035  
`_POSIX2_BC_BASE_MAX` 961  
`_POSIX2_BC_DIM_MAX` 961  
`_POSIX2_BC_SCALE_MAX` 961  
`_POSIX2_BC_STRING_MAX` 961  
`_POSIX2_C_BIND` 806  
`_POSIX2_C_DEV` 806, 1036  
`_POSIX2_C_VERSION` 806, 1035  
`_POSIX2_CHAR_TERM` 806, 1036  
`_POSIX2_COLL_WEIGHTS_MAX` 962  
`_POSIX2_EXPR_NEST_MAX` 962  
`_POSIX2_FORT_DEV` 806, 1036  
`_POSIX2_FORT_RUN` 806, 1036  
`_POSIX2_LINE_MAX` 962  
`_POSIX2_LOCALEDEF` 806, 1036  
`_POSIX2_RE_DUP_MAX` 962  
`_POSIX2_SW_DEV` 806, 1036  
`_POSIX2_UP` 1036  
`_POSIX2_UPE` 806  
`_POSIX2_VERSION` 806, 1035  
`_SC_2_C_BIND` 1038  
`_SC_2_C_DEV` 1038  
`_SC_2_C_VERSION` 1038  
`_SC_2_FORT_DEV` 1038  
`_SC_2_FORT_RUN` 1038  
`_SC_2_LOCALEDEV` 1038  
`_SC_2_SW_DEV` 1038  
`_SC_2_UPE` 1038  
`_SC_2_VERSION` 1038  
`_SC_ARG_MAX` 1038  
`_SC_BC_BASE_MAX` 1038  
`_SC_BC_DIM_MAX` 1038  
`_SC_BC_SCALE_MAX` 1038  
`_SC_BC_STRING_MAX` 1038  
`_SC_CHILD_MAX` 1038  
`_SC_CLK_TCK` 1038  
`_SC_COLL_WEIGHTS_MAX` 1038  
`_SC_EXPR_NEST_MAX` 1038  
`_SC_JOB_CONTROL` 1038  
`_SC_LINE_MAX` 1038  
`_SC_NGROUPS_MAX` 1038  
`_SC_OPEN_MAX` 1038  
`_SC_PASS_MAX` 1038  
`_SC_RE_DUP_MAX` 1038  
`_SC_SAVED_IDS` 1038  
`_SC_STREAM_MAX` 1038  
`_SC_TZNAME_MAX` 1038  
`_SC_VERSION` 1038  
`_SC_XOPEN_VERSION` 1038  
`_setjmp` 538  
`_tolower` 836, 932  
`_toupper` 837, 932  
`_XOPEN_CRYPT` 1036  
`_XOPEN_SHM` 1036  
`_XOPEN_VERSION` 1035

64-bit functions 14  
    for NFS V3.0 132

7-bit ASCII character  
    test 485

8-bit transparency 1053

**A**

a64l 163  
ABDAY\_1 954  
ABDAY\_2 954  
ABDAY\_3 954  
ABDAY\_4 954  
ABDAY\_5 954  
ABDAY\_6 954  
ABDAY\_7 954  
ABMON\_1 955  
ABMON\_10 955  
ABMON\_11 955  
ABMON\_12 955  
ABMON\_2 955  
ABMON\_3 955

- ABMON\_4 955
- ABMON\_5 955
- ABMON\_6 955
- ABMON\_7 955
- ABMON\_8 955
- ABMON\_9 955
- abnormal process abort 165
- abort 165, 991
- abs 166, 991
- absolute pathname 1053
- absolute value
  - compute 513
  - of an integer (long long int) 521
- access 167, 1039
- access mode 1053
- access permissions
  - check 167
- access protection for memory mapping 588
- access time
  - set for file 860
- access to slave pseudoterminal 458
- account number 1053
- acos 169
- acosh 170
- active handle 79
- ADD-FILE-LINK command 85
- address 1053
- address space 1053
- advance 171, 664, 975
  - regex 665
- AID 127
- AIO\_LISTIO\_MAX 805
- AIO\_MAX 805
- AIO\_PRIO\_DELTA\_MAX 805
- alarm 172, 1039
- alert 1053
- alert character 1054
- algorithms
  - for encoding strings 226
- alias name 1054
- allocate
  - code conversion descriptor 463
- allocated memory
  - free 341
- allocation
  - memory 560
- alphabetic character
  - test 484
- alphabetic wide character
  - test 497
- alphanumeric character
  - test 483
- alphanumeric wide character
  - test 496
- ALT\_DIGITS 955
- alternative ways of writing operators 953
- altzone 173
- AM\_STR 954
- Amendment 1 conformity
  - macro 760
- ANSI conformance
  - macro 760
- ANSI functionality 1049
- appropriate privileges 1054
- arc cosine
  - compute 169
- arc sine
  - compute 176
- arc tangent
  - compute 177
- arc tangent of x/y
  - compute 178
- ARG\_MAX 805, 958
- argument 1054
- argument list
  - formatted output 869
- array
  - binary search 192
  - I/O operations
    - definitions 1021
  - sort 192
- array variable
  - for timezone strings 846
- ASCII to EBCDIC string
  - convert 173
- ascii\_to\_ebcdic 173
- asctime 174, 1032

asctime\_r 175  
asin 176  
asinh 170, 176  
assert 177  
assert, macro 930  
assert.h 930  
atan 177  
atan2 178  
atanh 170, 178  
atexit 179, 991  
atof 180, 991  
atoi 181, 991  
atol 182, 991  
atoll 183  
authentication 1054

## B

B0 106, 1027  
B110 106, 1027  
B1200 106, 1028  
B134 106, 1027  
B150 106, 1028  
B1800 106, 1028  
B19200 106, 1028  
B200 106, 1028  
B2400 106, 1028  
B300 106, 1028  
B38400 106, 1028  
B4800 106, 1028  
B50 106, 1027  
B600 106, 1028  
B75 106, 1027  
B9600 106, 1028  
background 1054  
background process 1055  
background process group 1055  
backslash 1055  
backspace character 1055  
base name  
    for temporary file 834  
base-independent exponent 684  
basename 184  
basic data  
    operating system 852

baud rate  
    get for input 204  
    get for output 204  
    hang up 1027  
    set for input 205  
    set for output 206  
baud rates 1027  
BC\_BASE\_MAX 805, 960  
BC\_DIM\_MAX 805, 960  
BC\_SCALE\_MAX 805, 960  
BC\_STRING\_MAX 805, 960  
Bessel functions  
    of the first kind 509  
    of the second kind 927  
binary data  
    read 340  
binary data output 392  
binary file 1055  
binary search  
    sorted array 192  
binary search tree 841  
    build 841  
    find node 829  
    nomenclature 842  
    process 841  
    traverse 846  
binary tree  
    delete node 824  
blkcnt\_t 1019  
blkcnt64\_t 1019  
block special file 1056  
BLOCK-CONTROL-INFO 89  
blocked signal 725  
    change 744  
    examine 744  
block-mode terminal 1056  
blockwise encoding  
    strings 248  
brk 186  
BRKINT 102  
BS0 1027  
BS1 1027

- BS2000
  - console 110
  - execute command 190
  - functionality 42
  - get file names 189
- BS2000 command
  - execute 811
- bs2exit 188
- bs2fstat 189
- bs2system 190
- BSDLY 104, 1027
- bsearch 192, 991
- btowc 193
- buffer 1056
  - fflush 290
  - flushing 77
- buffering 1056
  - assign to a stream 698, 714
- BUFSIZ 985
- build
  - binary search tree 841
- byte
  - find in memory 562
  - get from standard input stream 399
  - get from stream 293, 396
  - push back onto input stream 853
  - put on a stream 334
  - put on standard output stream 631
  - put on stream 630
- bytes
  - copy in memory 561
  - read from file 647
  - swap 800
  - write to a file 920
- bytes in memory
  - compare 563
  - copy 564
  - with overlapping areas 566
- C**
- C locale 56, 704
- C++ source program
  - external "C" declarations 12
- c\_cc vector 109
- cabs 194
- calculate absolute value
  - of a complex number 194
- calloc 195, 991
- carriage-return character 1056
- cataloged disk file 85
- catclose 196, 972
- category
  - macros 965
- catgets 197, 972
- catopen 198, 972
- cbrt 199
- cc\_t 102, 1025
- cdisco 200
- ceil 201
- ceilf 201
- ceill 201
- cenaco 202
- cfgetispeed 204
- cfgetospeed 204
- cfsetispeed 205
- cfsetospeed 206
- change
  - current directory 270
  - file group 210, 273
  - file owner 210, 273
  - file permission bits 208
  - mode of file 271
  - priority of a process 606
  - root directory 212
  - working directory 207
- CHAR\_BIT 962
- CHAR\_MAX 962
- CHAR\_MIN 963
- character class 1056
- character in string
  - get first occurrence 781
  - get first occurrence of 464
  - get last occurrence 679, 785
- character set 1057
- character special file 1057
- character string 1057
  - regular expressions 665

- characters 1056
  - classification 932
  - convert to lowercase 837
  - convert to uppercase 838
  - in a line 97
  - white space 84
- CHARCLASS\_NAME\_MAX 964
- chdir 207, 1039
- check
  - access permissions 167
- child directory 1057
- child process 318, 1057
  - wait for process to stop or terminate 873
  - wait for status change 878
- CHILD\_MAX 805, 958
- chmod 208
- chown 210, 1039
- chroot 212, 1039
- classification
  - character 932
- clear
  - end-of-file indicator 213
  - error indicator 213
- clearerr 986
- CLK\_TCK 805
- CLK\_TCK, definition 1031
- CLOCAL 106, 1028
  - opening a terminal device file 94
- clock 214, 1032
- clock tick 1057
- clock\_t 1019
- CLOCKS\_PER\_SEC 1031
- close 215, 1039
  - a file 215
  - destroy handles 79
  - directory 216
  - stream 274
- closedir 216, 933
- code conversion 461
- code conversion descriptor
  - allocate 463
- CODESET 954
- codeset
  - conversion 953
- COLL\_WEIGHTS\_MAX 805, 960
- collating element 1057
- collating sequence 1058
- collation 1058
- column position 1058
- column positions
  - of a wide character code 915
  - of a wide character string 910
- command 1058
  - execution in BS2000 190, 811
  - execution in POSIX subsystem 811
- command interpreter 1059
- command option parsing 428
- command options
  - syntax analysis 428
  - variables for 428, 616
- command-line parser 428
- communication element 113
- compare
  - regular expressions 762
  - strings 762, 778
  - wide character strings 884, 917
  - with regular expression 171
- comparisons
  - regular expressions 975
- compilation date
  - macro 235
- compilation time
  - macro 829
- compile 219, 664, 975
  - regexp 665
- compiled regular expression 665
- complementary error function
  - use 257
- complementary wide character substring
  - get length 886
- compute
  - absolute value of floating-point number 268
  - arc cosine 169
  - arc sine 176
  - arc tangent 177
  - arc tangent of x/y 178
  - base 10 logarithm 536
  - cosine 221



- compute (continued)
  - difference between two calendar time values 239
  - Euclidian distance 460
  - exponential function 267
  - hyperbolic cosine 221
  - hyperbolic sine 751
  - hyperbolic tangent 814
  - remainder value of floating-point number 306
  - sine function 751
  - square root 754
  - tangent 814
- concatenate
  - strings 763
  - substrings 778
- confstr 220, 1039
- constants
  - for hardware control 1028
  - for tcflow 1030
  - for tcflush 1029
  - for tcsetattr 1029
  - implementation-dependent 957
  - language information 954
  - standard definition 1035
- contingency routine 117
  - define 202
  - disconnect 200
  - free programming 119
  - implementation using library functions 118
  - in Assembler 120
  - in C 119
- control 276
- control character 1059
  - test 487
- control devices 468
- control operations
  - for messages 590
- control STREAMS 468
- control wide character
  - test 498
- controlling process 95, 1059
- controlling terminal 95, 1059
  - of session leader 95
- conversion
  - codeset 953
- convert
  - ASCII string to EBCDIC string 173
  - date and time to string 175, 233, 531
  - date and time to user format 405
  - date and time to UTC 456, 457
  - date and time to wide character string 887
  - directory entries 410
  - EBCDIC string to ASCII 245
  - local time to time since epoch 579
  - multi-byte character to wide character 193
  - multi-byte string to wide-character 555, 557
  - string to long integer 790
  - string to long long integer 792
  - string to unsigned long long integer 796
  - wide character string to multi-byte string 894
  - wide character string to unsigned long long 907
  - wide character to 1-byte multi-byte character 912
  - wide character to character 912
  - wide characters to multi-byte characters 880
- convert to string
  - date and time 163, 782
  - monetary value 770
- copy
  - wide character string 917, 918
- core dump 1059
- cos 221
- cosh 221
- cosine
  - compute 221
- cpio archive mode values 931
- cpio.h 931
- CPU time used
  - by process 214
  - by task 222
- cputime 222
- CR 101
- CR0 1027
- CR1 1027
- CR2 1027
- CR3 1027

CRDLY 104, 1027  
CREAD 106, 1028  
creat 223, 946  
creat64 223, 946  
create  
    base name for temporary file 834  
    file 223  
    pathname for temporary file 827  
create FIFO file 571  
CRNCYSTR 956  
CRTE 11  
crypt 226, 1039  
CS5 1028  
CS6 1028  
CS7 1028  
CS8 1028  
CSIZE 106, 1028  
CSTOPB 106, 1028  
cstxlt 227  
ctermid 231, 986, 1039  
ctime 232, 1032  
ctime\_r 233  
ctype.h 932  
cube root 199  
currency\_symbol 965  
current directory 1059  
    change 270  
current time of day 449  
cuserid 234, 986, 1039

**D**  
D\_FMT 954  
D\_T\_FMT 954  
daemon 1059  
data  
    binary output 392  
    binary read 340  
    discard non-transmitted data 817  
data for file status 1011  
data segment  
    modify size 684  
data set pointer 1060  
data structures (IPC) 113  
data transmission  
    restart 816  
    serial 821  
    suspend 816  
data type  
    cc\_t 1025  
    speed\_t 1025  
    struct termios 1025  
    tflag\_t 1025  
data types 1019  
    definition 978  
    for time 1031  
    for wide characters 1045  
    header file 1019  
    NLS 972  
    standard 984  
date and time  
    additional definitions 1018  
    convert to local time 530  
    convert to string 174, 232, 233, 774  
    convert to user format 405  
    convert to UTC 456, 457  
    convert to wide character string 887  
    get 372, 448  
DAY\_1 954  
DAY\_2 954  
DAY\_3 954  
DAY\_4 954  
DAY\_5 954  
DAY\_6 954  
DAY\_7 954  
daylight 235, 1032  
daylight savings time  
    variable 235  
DBL\_DIG 962  
DBL\_MAX 962  
debugging options 127  
decimal digit, test 488  
decimal digit wide character, test 500  
decimal\_point 965  
declarations  
    for stack environment 977  
    for waiting for processes 1022  
    regular expressions 975

- default 1060
    - signal handling 978
  - default signal action 726
  - define values for termios 1025
  - define wide character class 914
  - define wide character mappings 913
  - definitions
    - for logging system error messages 1001
    - for memory management 1003
    - for the poll() function 973
    - of operations for ndbm database 970
    - of operations for XSI resources 1005
    - shared memory 1010
    - standard data types 984
  - DELAYTIMER\_MAX 805
  - delete
    - node from binary search tree 824
    - node from binary tree 824
    - record in ISAM file 283
  - delivered signal 725
  - descriptor
    - for code conversion 462, 463
  - descriptor table, size 412
  - destroy
    - search table 459
  - dev\_t 1019
  - device 1060
  - device ID 1060
  - device number
    - get formatted 552
    - get major component 550
    - get minor component 568
  - diagnostic messages
    - output 177
  - difference between two calendar time values 239
    - compute 239
  - difftime 239, 1032
  - DIR 933
  - directory 1060
    - change 207
    - close 216
    - get pathname of 403
    - make 569, 573
    - open 615
      - read 650
      - read (thread-safe) 652
      - remove 681
  - directory entry 1060
  - directory stream 1060
    - get current location 826
    - reset file position indicator 678
    - set position of 686
  - dirent.h 933
  - disk file 85
    - file attributes 85
    - record-oriented I/O 92
    - stream-oriented I/O 91
  - display 1059
  - display (on-screen) 1060
  - div 241, 991
  - DIV (DATA IN VIRTUAL) 91
  - div\_t 990
  - division of integers (long long int) 522
  - division remainder 671
  - DMS 162, 929
  - dot 1061
  - dot-dot 1061
  - downshifting 1061
  - drand48 242, 991
  - dup 244, 1039
    - create handles 79
  - dup2 244, 1039
- E**
- E2BIG 934
  - EACCES 934
  - EADDRINUSE 934
  - EADDRNOTAVAIL 934
  - EADV 934
  - EAFNOSUPPORT 934
  - EAGAIN 934
  - EALREADY 935
  - EBADF 935
  - EBADFD 935
  - EBADMSG 935
  - EBADR 935
  - EBADRQC 935

EBADSLT 935  
EBCDIC character  
    test 489  
EBCDIC to ASCII string conversion 245  
ebcdic\_to\_ascii 245  
EBFONT 935  
EBUSY 935  
ECHILD 935  
ECHNL 1029  
ECHO 1029  
ECHOE 1029  
ECHOK 1029  
ECHRNG 935  
ECOMM 935  
ECONNABORTED 936  
ECONNREFUSED 936  
ECONNRESET 936  
ecvt 246  
EDEADLK 936  
EDEADLOCK 936  
EDESTADDRREQ 936  
EDMS 936  
EDOM 936  
EDT  
    call 248  
edt 248  
EEOF 936  
EEXIST 936  
EFAULT 936  
EFBIG 936  
effective group ID 1061  
    get 412  
effective user ID 1061  
    get 414  
EHOSTDOWN 936  
EHOSTUNREACH 936  
EIDRM 936  
EIKEYOFLW 936  
EILSEQ 937  
EINPROGRESS 937  
EINTR 937  
EINVAL 937  
EIO 937  
EIORESID 937  
EISCONN 937  
EISDIR 937  
EISNAM 937  
EL2HLT 937  
EL2NSYNC 937  
EL3HLT 937  
EL3RST 937  
element in queue 467  
elementary functions 1061  
ELIBACC 938  
ELIBBAD 938  
ELIBEXEC 938  
ELIBMAX 938  
ELIBSCN 938  
ELNRNG 938  
ELOOP 938  
EMACRO 938  
EMFILE 938  
EMLINK 938  
EMODE 938  
empty directory 1061  
empty signal set  
    initialize 735  
empty string 1061  
empty wide character string 1061  
EMSGSIZE 938  
EMULTIHOP 938  
ENAME 938  
ENAMETOOLONG 939  
ENAVAIL 939  
encoding key  
    set 703  
encrypt 248, 1039  
encryption algorithms 226  
endgrent 249  
end-of-file indicator  
    clear 213  
    test 288  
endpwent 251  
endutxent 253  
ENETDOWN 939  
ENETRESET 939  
ENETUNREACH 939  
ENFILE 939

ENOANO 939  
ENOBUFS 939  
ENOCSS 939  
ENODATA 939  
ENODEV 939  
ENOENT 939  
ENOEXEC 939  
ENOLCK 939  
ENOLINK 939  
ENOMEM 940  
ENOMSG 940  
ENONET 940  
ENOPKG 940  
ENOPROTOPT 940  
ENOSPC 940  
ENOSR 940  
ENOSTR 940  
ENOSUBSYS 940  
ENOSYS 940  
ENOTBLK 940  
ENOTCONN 940  
ENOTDIR 940  
ENOTEMPTY 940  
ENOTNAM 941  
ENOTSOCK 941  
ENOTTY 941  
ENOTUNIQ 941  
entry  
    in linear search table 517  
entry format 1043  
entry in group file  
    for group name 417, 418  
    get for group ID 415, 416  
environ 256  
environment 256  
    external variable 256  
environment variable  
    add 632  
    change 632  
    get value of 413  
    LANG 72  
ENXIO 941  
EOF 100, 288, 985  
EOL 100  
EOPNOTSUPP 941  
EOPR 941  
EOVERFLOW 941  
EPERM 941  
EPFNOSUPPORT 941  
EPIPE 941  
epoch 16, 1062  
    get time 830  
    local time conversion 579  
EPROTO 941  
EPROTONOSUPPORT 941  
EPROTOTYPE 941  
equivalence class 1062  
ERA 955  
ERA\_D\_FMT 955  
ERA\_D\_T\_FMT 955  
ERA\_T\_FMT 955  
erand48 242, 256, 991  
ERANGE 941  
ERASE 99  
    effect 97  
EREMCHG 941  
EREMOTE 942  
EREMOTEIO 942  
EREPL 942  
ERESTART 942  
erf 257  
erfc 257  
EROFS 942  
errno 126, 258, 934  
errno.h 126, 934  
ERROR  
    regexp 665  
error code 126  
error codes  
    system 934  
error function  
    use 257  
error indicator  
    clear 213  
    test on stream 289  
error messages 126  
error return values  
    XSI 258

ESHUTDOWN 942  
ESOCKTNOSUPPORT 942  
ESPIPE 942  
ESRCH 942  
ESRMNT 942  
ESSNOTAVAIL 942  
ESTALE 942  
ESTRPIPE 943  
ETIME 943  
ETIMEDOUT 943  
ETOOMANYREFS 943  
ETXTBSY 943  
EUCLEAN 943  
Euclidian distance, compute 460  
EUNATCH 943  
EUSERS 943  
eventing 118  
events  
    generating signals 725  
EWOULDBLOCK 943  
EX 1011, 1012  
EXDEV 943  
exec 259  
exec functions  
    destroy handles 79  
execl 259, 1039  
execle 259, 1039  
execlp 259, 1040  
executable file 1062  
execute  
    BS2000 command 190, 811  
    POSIX command 811  
    system command 811  
execv 259, 1040  
execve 259, 1040  
execvp 259, 1040  
EXFULL 943  
exit 264, 991, 1040  
EXIT\_FAILURE 990  
EXIT\_SUCCESS 990  
exp 267  
expiration date for time functions 16  
exponent of a floating-point number  
    load 516  
    exponential function  
        compute 267  
        use 267  
EXPR\_NEST\_MAX 805, 960  
expression 1062  
extended security controls 1062  
extended tar definitions 1023  
external "C" declarations 12  
external variable  
    environment 256

**F**  
F\_DUPFD 944  
F\_GETFD 944  
F\_GETFL 944  
F\_GETLK 944  
F\_GETLK64 944  
F\_OK 1037  
F\_RDLCK 944  
F\_SETFD 944  
F\_SETFL 944  
F\_SETLK 944  
F\_SETLK64 944  
F\_SETLKW 944  
F\_SETLKW64 944  
F\_UNLCK 944  
F\_WRLCK 944  
fabs 268  
FCBTYPE  
    record-oriented I/O 93  
fchdir 270  
fchmod 271  
fchown 273  
fclose 274, 986, 988  
    destroy handles 79  
fcntl 276, 946  
    create handles 79  
fcntl.h 944  
fcvt 282  
FD\_CLOEXEC 944  
FD\_CLR 282  
FD\_ISSET 282  
FD\_SET 282  
FD\_ZERO 282

fdelrec 283  
fdopen 286, 986  
    create handles 79  
feature test macro 1062  
feof 288, 986  
ferror 289, 986  
FF0 1027  
FF1 1027  
FFDLY 105, 1027  
fflush 290, 986  
ffs 292  
fgetc 293, 986  
fgetpos 295, 986  
fgetpos64 295, 986  
fgets 296, 986  
fgetwc 297, 1045  
fgetws 299, 1045  
FIFO special file 1062  
FILE 986  
file 1063  
    association with a stream 77  
    change group 210, 273  
    change owner 210, 273  
    close 215  
    closing 77  
    control 276  
    convert EBCDIC to ASCII 245  
    create 223  
    create base name 834  
    creating 77  
    disassociating from a stream 77  
    execute 259  
    get file-position indicator 825  
    open 609  
    opening 77  
    overwrite 223  
    read bytes from 647  
    remove 672  
    rename 674  
    set access time 860  
    set modification time 860  
    set to specified length 375, 840  
    symbolic link 802  
    synchronize changes 369  
    utmpx 714  
    write bytes to 920  
file access and modification time  
    set 862  
file access permission 1063  
    change 208  
file control options 944  
file description 1064  
    handles for 79  
    open file 79  
file descriptor 1064  
    creating 79  
    duplicate 244  
    get 300  
file editor  
    call 248  
file group  
    change 210  
file group class 1064  
file hierarchy 1064  
file link  
    create 519  
file mode 1064  
    change 208, 271  
file mode bits 271  
    change 271  
file mode creation mask  
    get and set 850  
file name 1065  
    temporary 577  
file name/pathname  
    output 657  
file offset 1065  
    lseek 544  
file other class 1065  
file owner  
    change 210, 273  
file owner class 1065  
file permission bits 1065  
    change 208  
file position indicator 77, 1065  
    fseek 356  
    fsetpos 361  
    ftell 370

- file position indicator (continued)
  - reset to start of directory 678
  - rewind 677
  - set in directory stream 686
  - set in ISAM file 301
- file processing
  - disk files 85
  - INCORE files 94
- file serial number 1066
- file status 1066
  - get 363, 756
  - query 548
- file structure 1066
- file system 1066
  - get type 809
  - mount 586
  - unmount 851
- file system information 760
  - read 366
- file times update 1066
- file tree
  - traverse 377
- file tree traversal 951
- FILENAME\_MAX 985
- fileno 300, 986
  - create handles 79
- file-position indicator
  - tell 825
- filter 1067
- find
  - entry in linear search table 517
  - node in binary search tree 829
- flag bits
  - definition 981
- float.h 947
- floating-point number
  - convert to string 246, 282, 395
  - extract mantissa and exponent 344
  - load exponent 684
  - next displayable 602
  - round off 305
  - round up 201
  - split into integer and fractional parts 585
- flocate 301
- flock 945
- flock64 946
- flockfile 303, 986
- floor 305
- floorf 305
- floorl 305
- FLT\_DIG 962
- FLT\_MAX 963
- flush
  - /reopen stream 132
  - non-transmitted data 817
- fmod 306
- fmtmsg 307
- fdopen 312, 986
  - create stream 79
- FOPEN\_MAX 985
- fdopen64 312, 986
- foreground 1067
- foreground process 1067
- foreground process group 1067
- foreground process group ID 1067
  - get 819
  - set 824
- foreground process groups
  - definition 94
- fork 318, 1040
  - create handles 79
- format
  - output wide characters 380
- format of directory entries 933
- formatted device number
  - get 552
- formatted input
  - read from standard input stream 685
  - read from string 755
- formatted output
  - of variable argument list 869
  - of wide characters 919
  - to a string 871
  - to standard output 870
  - to string 754
  - write on output stream 321
- formatted output of wide characters 870



- 
- formatted read 345, 386, 801, 927
    - from standard input stream 345
    - from stream 345
    - from string 345
  - formatted wide character output 870
  - formatted write on standard output stream 629
  - form-feed character 1067
  - fpathconf 320, 617, 1040
  - fpos\_t 986
  - fpos64\_t 986
  - fprintf 321, 986
  - fputc 334, 986, 987
  - fputs 336, 986
  - fputwc 337, 1045
  - fputws 339, 1045
  - frac\_digits 965
  - fread 340, 986
  - free 341, 991
  - freopen 342, 987
  - freopen64 342, 987
  - fsblkcnt\_t 1019
  - fsblkcnt64\_t 1019
  - fscanf 345, 987
  - fseek 356, 987
  - fseek64 356, 987
  - fseeko 356
  - fseeko64 356
  - fsetpos 361, 987
  - fsetpos64 361, 987
  - fsfilcnt\_t 1019
  - fsfilcnt64\_t 1019
  - fstat 363
  - fstat64 363
  - fstatvfs 366
  - fstatvfs64 366
  - fsync 369, 1040
  - ftell 370, 987
  - ftell64 370
  - ftello 370
  - ftello64 370, 987
  - ftime 372
  - ftruncate 375
  - ftrylockfile 303, 376, 987
  - ftw 377
  - ftw.h 951
  - FTW\_D 951
  - FTW\_DNR 377, 951
  - FTW\_F 951
  - FTW\_NS 377, 951
  - full-duplex mode
    - operation 96
  - fully buffered stream 77
  - function
    - general information 13
    - reentrant 727, 729
    - safe 727
    - unsafe 728
  - function and macro, differences 13
  - function definitions for pattern matching 956
  - funlockfile 303, 378, 987
  - fwide 379
  - fwprintf 380
  - fwrite 392, 987
  - fwscanf 386
- G**
- gamma 394
  - garbcoll 395
  - gcvt 395
  - general terminal interface 94
  - generate
    - process in virtual memory 868
    - pseudo-random number 645
  - get
    - BS2000 file names 189
    - byte from standard input stream 399
    - byte from stream 293, 396
    - character in string 464, 679, 785
    - column positions of a wide character
      - code 915
    - column positions of a wide character
      - string 910
    - entry in group file for group ID 415, 416
    - file status 756
    - group file entry for group name 417
    - login name 423, 424
    - number of bytes in a multi-byte character 554

get (continued)  
  number of bytes remaining in a multi-byte  
    character 554  
  string from standard input stream 446  
  string from stream 296  
  string length 777  
  substring length 786  
  user name 438  
  wide character from standard input  
    stream 454  
  wide character from stream 453  
  wide character in string 893  
  wide character string from stream 297, 299

get exponent part 464, 537

get file position indicator  
  fgetpos 295

get process group ID 447

get program name 432

get status (file) 363

GETALL 1008

GETC  
  regexp 664

getc 396, 987

getc\_unlocked 398, 987

getchar 399, 987

getchar\_unlocked 398, 400, 987

getcontext 401

getcwd 403, 1040

getdate 405

getdents 410

getdents64 410

getdtablesize 412

getegid 412, 1040

getenv 413, 991

geteuid 414, 1040

getgid 414, 1040

getgrent 249, 414

getgrgid 415

getgrgid\_r 416  
  max. data buffer size 807

getgrnam 417

getgrnam\_r 418  
  max. data buffer size 807

getgroups 419, 1040

gethostid 420

gethostname 420

getitimer 421

getlogin 423, 1040

getlogin\_r 424

getmsg 425

GETNCNT 1008

getopt 428, 987, 1040

getpass 431, 1040

getpgrp 432

getpgname 432

getpgrp 433, 1040

GETPID 1008

getpid 433, 1040

getpmsg 434

getppid 432, 434, 1040

getpwent 251, 436

getpwnam 437

getpwnam\_r 438  
  max. data buffer size 807

getpwuid 439

getpwuid\_r 440  
  max. data buffer size 807

getrlimit 441

getrlimit64 441

getrusage 445

gets 446, 987

getsid 447

getsubopt 448

gettimeofday 449

gettsn 450

getuid 450, 1040

getutx 253, 451

getutxent 253, 451

getutxid 253, 451

getutxline 253, 451

GETVAL 1008

getw 452, 987

getwc 453, 1045

getwchar 454, 1045

getws 1045

GETZCNT 1008

gid\_t 1019

global pattern matching 455

gmatch 455

- gmtime 456, 1032
  - gmtime\_r 457
  - grantpt 458
  - group database 1067
  - group file entry
    - for group ID 415, 416
    - for group name 417, 418
    - get 414
  - group ID 1067
    - effective 412
    - for job control 707
    - of a process 433
    - of process 700
    - real 414
    - set 709
  - group IDs
    - supplementary 419
  - group name 1068
  - group of file
    - change 273
  - group structure 952
  - grouping 965
  - grp.h 952
- H**
- handles 79
    - active 79
    - creating 79
    - destroy 79
    - for file description 79
  - hang up, baud rate 1027
  - hardware
    - name 852
  - hardware control of a terminal 1028
  - hash table
    - manage 459
  - hcreate 459
  - hdestroy 459
  - header file 1068
    - data types 1019
    - data types for time 1031
    - implementation-dependent constants 957
    - signals 978
    - standard constants 1035
    - standard I/O 985
    - standard library 990
    - standard structures 1035
    - structure for system names 1021
  - hexadecimal digit
    - test 508
  - hexadecimal digit wide character
    - test 507
  - home directory 1068
  - host 1068
  - host, ID of current 420
  - host, name of current 420
  - hsearch 459
  - HUGE\_VAL 967
  - HUPCL 106, 1028
  - hyperbolic cosine
    - compute 221
  - hyperbolic sine
    - compute 751
  - hyperbolic tangent
    - compute 814
  - hypot 460
- I**
- I/O on terminal 110
  - I/O, record-oriented 1077
  - IC@LOCAL 70
  - ICANON 1029
  - iconv 461, 953
  - iconv.h 953
  - iconv\_close 462, 953
  - iconv\_open 463, 953
  - iconv\_t 953
  - ICRNL 102, 1026
  - ID of current host 420
  - IEXTEN 1029
  - IGNBRK 102, 1026
  - IGNCR 102, 1026
  - IGNPAR 102, 1026
  - ilogb 464
  - implementation-dependent constants
    - header file 957
  - INCORE file 94
  - index 464

- index names for special characters 109
- information
  - on file system type 809
- INIT
  - regex 664
- INLCR 102, 1026
- inline generation 13
- ino\_t 1019
- INPCK 102, 1026
- input baud rate
  - get 204, 205
- input buffer 96
  - for terminal 96
- input line
  - maximum length 97
- input mode
  - non-canonical 96
  - standard 96
- input processing, types 96
- input stream
  - push back byte 853
- insque 467
- int\_curr\_symbol 965
- int\_frac\_digits 965
- INT\_MAX 963
- INT\_MIN 963
- integer
  - convert to legal value 835, 836
  - division 517
- integer absolute value
  - return 166
- integers
  - divide 241
- internationalization 1068
- interprocess communication 112
  - data structures 113
  - status information 115
  - structure 1000
  - system identifier 113
- interrupt 1068
- interrupt behavior, system calls 737
- interval timer
  - set 848
  - set or read 421
- INTR 99
- ioctl 468
- IPC\_CREAT 1000
- IPC\_EXCL 1000
- IPC\_NOWAIT 1000
- IPC\_PRIVATE 1000
- IPC\_RMID 1000
- IPC\_SET 1000
- IPC\_STAT 1000
- isalnum 483, 932
- isalpha 484, 932
- ISAM file 85
  - delete record 283
  - K/NK format 90
  - set file position indicator 301
- isascii 485, 932
- isatty 486, 1040
- iscntrl 487, 932
- isdigit 488, 932
- isebcdic 489
- isgraph 490, 932
- ISIG 1029
- islower 491, 932
- isnan 491
- iso646.h 953
- iso646.h, header file 13
- isprint 492, 932
- ispunct 493, 932
- isspace 494, 932
- ISTRIP 102, 1026
- isupper 495, 932
- iswalnum 496, 1045
- iswalph 497, 1045
- iswcntrl 498, 1045
- iswctype 499, 1045
- iswdigit 500, 1045
- iswgraph 501, 1045
- iswlower 502, 1045
- iswprint 503, 1045
- iswpunct 504, 1045
- iswspace 505, 1045
- iswupper 506, 1045
- iswxdigit 507, 1045
- isxdigit 508, 932

- IUCLC 102, 1026
- IXANY 102, 1026
- IXOFF 102, 1026
- IXON 102, 1026
- J**
- j0, j1, jn 509
- job control 1068
- job control ID 1069
- job variable 1069
- job-step termination 266
- join file 1069
- jrnd48 242, 509, 991
- jump
  - non-local 539
  - non-local jump using signal 739
  - set label for non-local jump 702
  - set non-local label 701
- K**
- K block format 89
- kernel 1069
- kex\_t 1019
- KILL 100
  - effect 97
- kill 510, 981
- K-ISAM file 90
- KR functionality 1049
- L**
- L\_ctermid 985
- L\_cuserid 985
- L\_tmpnam 985
- l64a 163
- label
  - for non-local jump 701
  - for non-local jump using a signal 747
  - set for non-local jump 702
- labs 513, 991
- LANG 72
- langinfo.h 954
- language information
  - constants 954
- LC\_ALL 72, 704, 965
- LC\_C\_GERMANY 59
- LC\_C\_V1CTYPE 58, 59
- LC\_COLLATE 53, 704, 884, 965
- LC\_CTYPE 53, 704, 965
- LC\_MESSAGES 704, 965
- LC\_MONETARY 54, 704, 965
- LC\_NUMERIC 54, 704, 965
- LC\_TIME 704, 965
- lcong48 242, 516, 991
- lconv 965
- ldexp 516
- ldiv 517, 991
- ldiv\_t 990
- length
  - of a string 777
  - of an input line 97
  - of complementary substring 766
- lfind 517, 543
- lgamma 518
- libraries for time functions 16
- library 1069
- limit for resource
  - get 441
  - set 441, 711
- limits.h 957
- line buffered stream 77
- line control 1029
- line feed 110
- line number
  - macro 518
- LINE\_MAX 805, 960
- linear search 543
- linear search table 517
- linear table
  - search 543
- linear update 543
- link 519, 1040, 1069
  - make link to file 802
  - remove 856
- link count 1070
- link name, IC@LOCAL 70
- link option for time functions 16
- link to file
  - create 519

- LINK\_MAX 617, 959
  - llabs 521
  - lldiv 522
  - llrint, llrintf, llrintl 523
  - llround, llroundf, llroundl 524
  - loc1 525, 664, 975
  - loc2 525, 664, 975
  - local machine 1070
  - local time
    - convert into time since the Epoch 579
  - locale 1070
    - alter 704
    - change components 526
    - get values 607
    - query 704
    - set 704
    - user-specific 70
  - locale.h 965
  - localeconv 526, 966
  - localization 1070
  - localtime 530, 1032
  - localtime\_r 531
  - location
    - in directory stream 826
  - lock clients 398, 400
  - lock file section 532
  - lock standard input/output 303, 376
  - lockf 532
  - lockf64 532, 1040
  - locs 535, 664, 975
  - log message 810
  - log priority mask 706
  - log10 536
  - logarithm
    - compute to base 10 536
    - of gamma function 394, 518
  - logb 537
  - logging system error messages
    - definitions 1001
  - login name 1070
    - get 234, 423, 424
  - LOGIN\_NAME\_MAX 805
  - long division
    - of integers 517
    - long integer
      - compute absolute value 513
  - LONG\_BIT 963
  - LONG\_MAX 963
  - LONG\_MIN 963
  - longjmp 539
  - lowercase letter
    - convert to uppercase 837
    - test 491
  - lowercase wide character
    - test 502
  - lrand48 242, 540, 991
  - lrint, lrintf, lrintl 541
  - lround, lroundf, lroundl 542
  - lsearch 543
  - lseek 544, 1040
  - lseek64 544, 1040
  - lstat 548
  - lstat64 548
- M**
- machine 1021
  - machine-dependent floating-point arithmetic 684
  - macro
    - Amendment 1 conformity 760
    - assert 930
    - for ANSI conformance 760
    - for compilation date 235
    - for compilation time 829
    - for line number 518
    - for source file names 300
    - for synchronous multiplexing 282
    - General information 14
  - macro and function, differences 13
  - macros
    - category 965
  - major 550
  - major component of device number
    - get 550
  - make
    - directory 569
    - directory, special file or text file 573
    - temporary file name 577
    - unique temporary file name 576

---

- make file name 576
- makecontext 551
- makedev 552
- malloc 553, 991
- manage
  - hash table 459
- manipulate time structures 1042
- math.h 967
- mathematical functions and constants 967
- mathematical range 1070
- MAX\_CANON 97, 617, 959
- MAX\_INPUT 617, 959
- MB\_CUR\_MAX 990
- MB\_LEN\_MAX 963
- mblen 554, 991
- mbrlen 554
- mbrtowc 555
- mbsinit 556
- mbsrtowcs 557
- mbstowcs 558, 991
- mbtowc 559, 991
- memalloc 560
- memccpy 561, 993
- memchr 562, 993
- memcmp 563, 993
- memcpy 564, 993
- memfree 565
- memmove 566, 993
- memory
  - allocate 195
  - allocator 553
  - free allocated memory 341
  - release to system 395
  - request 867
  - shared 716, 718
  - synchronize 598
- memory allocation 560
- memory area 1070
  - free 565
  - initialize 567
- memory mapping, access protection 588
- memory page mapping
  - set up 581
- memory pages
  - map 581
  - unmap 600
- memory reallocator 656
- memset 567, 993
- message 112
  - get text 768
  - output formatted 307
  - read 197
  - read from STREAMS file 425, 434
  - receive from queue 594
  - send to queue 596
  - send to STREAMS file 633
- message catalog 1070
  - close 196
  - open 198
- message catalog descriptor 1071
- message queue 112
  - get 592
- message queue identifier (msgid) 113
- messages 112
  - control operations 590
  - to standard error 622
- minor 568
- minor component of device number
  - get 568
- mkdir 569
- mkfifo 571
- mknod 573
- mkstemp 576
- mktemp 577
- mktime 579, 1032
- mmap 581
- mode 1071
- mode\_t 1019
- modf 585
- modification time
  - set for file 860
- modify
  - size of data segment 684
  - user context 699
- MON\_1 954
- MON\_10 955
- MON\_11 955

MON\_12 955  
MON\_2 954  
MON\_3 954  
MON\_4 954  
MON\_5 954  
MON\_6 955  
MON\_7 955  
MON\_8 955  
MON\_9 955  
mon\_decimal\_point 965  
mon\_grouping 965  
mon\_thousands\_sep 965  
monetary value  
    convert to string 770  
monetary.h 970  
mount 586  
mount point 1071  
mprotect 588  
MQ\_OPEN\_MAX 805  
MQ\_PRIO\_MAX 805  
mrand48 242, 589, 991  
MSG\_NOERROR 1004  
msgctl 590  
msgget 592  
msgrcv 594  
msgsnd 596  
msync 598  
multi-byte character 1071  
    convert to wide character 193  
multi-byte characters  
    convert to wide characters 559  
multi-byte characters, introduction 15  
multi-byte string  
    complete/convert to wide-character  
        string 555  
    convert to wide-character string 557, 558  
multiplex STREAMS I/O 624  
multiplexing 282  
munmap 600

**N**  
n\_cs\_precedes 965  
n\_sep\_by\_space 965  
n\_sign\_posn 965  
name 852  
    current host 420  
    hardware 852  
    operating system 852  
NAME\_MAX 617, 959  
NaN 491  
nanosleep 601  
natural logarithm 536  
NCCS 102, 1025  
negative\_sign 965  
network name  
    operating system 852  
new process  
    create 318  
new process image file 259  
newline 110  
newline character 1071  
next displayable floating-point number 602  
nextafter 602  
NFS V3.0 support 14  
nftw 603  
NGROUPS\_MAX 805, 960  
nice 606, 1040  
nice value 606  
NK block format 89  
NK-ISAM file 90  
NL 100  
NL\_ARGMAX 964  
NL\_CAT\_LOCALE 972  
nl\_catd 972  
nl\_item 972  
nl\_langinfo 607  
NL\_LANGMAX 964  
NL\_MSGMAX 964  
NL\_NMAX 964  
NL\_SETD 972  
NL\_SETMAX 964  
NL\_TEXTMAX 964  
nl\_types.h 972  
NLO 1026  
NL1 1027  
NLDLY 104, 1026  
nlink\_t 1019



NLS  
   data types 972  
 node  
   delete from binary search tree 824  
   find in binary search tree 829  
 nodename 1021  
 NOEXPR 955  
 NOFLSH 1029  
 nomenclature  
   binary search trees 842  
 non-canonical input mode 96  
 non-local jump  
   execute 539  
   execute using signal 739  
   set label 701, 702  
   set label using signal 747  
   without signal mask 538  
 NOSTR 955  
 nrand48 242, 607, 991  
 NULL 965, 984, 985, 990, 993, 1037  
 null byte 1071  
 null pointer 990, 1037, 1071  
 NULL, definition 1031  
 number  
   characters in a line 97  
 number of bytes in a multi-byte character  
   get 554  
 number of bytes remaining in a multi-byte character  
   get 554  
 NZERO 964  
  
**O**  
 O\_ACCMODE 945  
 O\_APPEND 945  
 O\_CREAT 945  
 O\_EXCL 945  
 O\_NOCTTY 945  
 O\_NONBLOCK 945  
   buffering of output 99  
   clear 97  
 O\_RDONLY 945  
 O\_RDWR 945  
 O\_SYNC 945  
  
 O\_TRUNC 945  
 O\_WRONLY 945  
 object file 1071  
 OCRNL 104, 1026  
 OFDEL 104, 1026  
 off\_t 1019  
 offset of structure component  
   from start of structure 608  
 offsetof 608, 984  
 OFILL 104, 1026  
 OLCUC 104, 1026  
 ONLCR 104, 1026  
 ONLRET 104, 1026  
 ONOCR 104, 1026  
 open 79, 609, 946  
   a file 609  
   directory 615  
   pipe stream 627  
   stream 312  
 open file 1072  
   control 276  
 OPEN\_MAX 805, 958  
 open64 609  
 opendir 615, 933  
 operating system  
   basic data 852  
   UNIX 1081  
 operations  
   for strings 993, 994  
 OPOST 104, 1026  
 optarg 428, 616, 988, 1041  
 opterr 428, 616, 988, 1041  
 optind 428, 616, 988, 1041  
 option 1072  
 option-argument 1072  
 optopt 428, 616, 988, 1041  
 orphaned process group 1072  
 output  
   diagnostics 78  
   wait for transmission of 815  
   writing 78  
 output baud rate  
   get 204  
   set 206

- output file name/pathname 657
- overflow block
  - NK-ISAM file 90
- overwrite
  - file 223
- P**
- p\_cs\_precedes 965
- p\_sep\_by\_space 965
- p\_sign\_posn 965
- P\_tmpdir 985
- PAM file 85
  - temporary 94
- PARENB 106, 1028
- parent directory 1072
- parent process 318, 1072
- parent process ID 1072
  - get 434
- PARMRK 102, 1026
- PARODD 106, 1028
- parser 1072
  - for command line 428
- parsing
  - command options 428
- PASS\_MAX 958
- password 1072
- password structure 974
- PATH\_MAX 617, 959
- pathconf 617, 1040
- pathname 1073
  - for temporary file 827
  - generate for terminal 231
  - of a terminal 843
  - of current working directory 403
- pathname prefix 1073
- pathname resolution 1073
- pathname variable
  - get value 617
  - get value of 320
  - value 959
- pattern 1074
- pattern matching 665
  - global 455
- pause 620, 1040
- pclose 621, 987
- PEEKC
  - regexp 664
- pending signal 725
  - examine 743
- perror 622, 987
- pid\_t 978, 1019
- pipe 79, 623, 1040, 1074
  - create 623
- pipe stream to or from a process, open 627
- pipe stream, close 621
- PIPE\_BUF 617, 959
- PM\_STR 954
- pointer
  - as result parameter 124
  - as return value 124
- poll 624
- popen 627, 987
  - create stream 79
- portability 1074
- portable character set 1074
- portable file name character set 1074
- portable pathname 1074
- position
  - of directory stream 686
  - read/write pointer in ISAM file 301
  - read/write pointer on directory stream 678
  - set file position indicator in ISAM file 301
  - set file position indicator in stream 361
- positive\_sign 965
- POSIX command
  - execute 811
- POSIX file system 1075
- POSIX functionality 40
- POSIX link option (for time functions) 16
- POSIX locale 56, 704
- POSIX shell 1075
- POSIX thread functions
  - for explicitly locking clients 141
  - for locking and unlocking objects 140
  - reentrant functions 140
  - that affect a process or thread 141
- POSIX thread support 14
  - alphabetical order 140

- POSIX\_ASYNCHRONOUS\_IO 805
- pow 628
- power function 628
- printf 321, 629, 987
- printing character
  - test 492
- printing wide character
  - test 503
- process 1075
  - binary search tree 841
  - change priority 606
  - controlling 95
  - create 318
  - effective group ID 412
  - effective user ID 414
  - generate in virtual memory 868
  - real group ID 414
  - real user ID 450
  - report CPU time used 214
  - suspend 620, 752, 859
  - terminate normally 264
  - wait for a signal 750
  - zombie 264
- process abort 165
  - abnormal 165
- process group 1075
- process group ID 700, 1075
  - for job control 707
  - get 412, 414, 433, 447
  - get for foreground process 819
  - set 708
  - set for foreground process group 824
- process group leader 1075
- process group lifetime 1076
- process ID 1076
  - get 433
  - of parent process 434
- process image file
  - new 259
- process lifetime 1076
- process limits
  - get 849
  - set 849
- process nice value 606
- process termination
  - job-step 266
  - normal 264
- process termination function
  - register 179
- process times
  - get 831
- processes
  - declarations for waiting 1022
- processor 852
- processor name 852
- program
  - terminate with MONJV 188
- program termination
  - with MONJV 188
- protocol 1076
- pseudo-random number
  - generate 242, 516, 540, 589, 645
  - generate with initialization value 256, 509, 607
  - generate with seed 754
- pseudoterminal pair
  - unlock 858
- pthread 1076
- PTHREAD\_DESTRUCTOR\_ITERATIONS 806, 807
- PTHREAD\_KEYS\_MAX 806, 807
- pthread\_kill 981
- pthread\_sigmask 981
- PTHREAD\_STACK\_MIN 806, 807
- PTHREAD\_THREADS\_MAX 807
- ptrdiff\_t 984
- punctuation character
  - test 493
- punctuation wide character
  - test 504
- push back
  - byte onto input stream 853
- put
  - byte on a stream 334
  - byte on standard output stream 631
  - byte on stream 630
  - string on a stream 336
  - string on standard output stream 637

put (continued)  
  wide character code on stream 337  
  wide character on standard output  
    stream 641  
  wide character on stream 640  
  wide character string on a stream 339  
putc 630, 986, 987  
putc\_unlocked 398, 630, 987  
putchar 631, 987  
putchar\_unlocked 398, 631, 987  
putenv 632, 991  
putmsg 633  
putpwent 636  
puts 637, 987  
pututxline 253, 638  
putw 639, 987  
putwc 640, 1045  
putwchar 641, 1045  
putws 1045  
pwd.h 974

**Q**

qsort 642, 992  
query  
  file status 548  
queue 467  
  get for messages 592  
  remove element 673  
quicksort algorithm 642  
QUIT 99

**R**

R\_OK 1037  
radix character 1076  
RADIXCHAR 955  
raise 643, 981  
rand 645, 992  
RAND\_MAX 990  
rand\_r 645  
random numbers  
  generate 589  
re\_comp 658  
RE\_DUP\_MAX 807, 960  
re\_exec 658

read 647, 1040  
  array from file 654  
  binary data 340  
  bytes from file 647  
  contents of a symbolic link 653  
  directory 650  
  file system information 366  
  formatted 345, 386, 801, 927  
  formatted from file 386  
  formatted from standard input 386  
  formatted input from file 380  
  formatted input from standard input  
    stream 685  
  formatted input from string 755  
  from standard input stream with format 345  
  from stream with format 345  
  stream with format 345  
  string with format 345  
  string without echo 431  
  wide character from stream 297  
  word from stream 452  
read (thread-safe)  
  directory 652  
read input 78  
readdir 650, 933  
readdir\_r 652, 933  
readdir64 650, 933  
readlink 653  
read-only file system 1077  
real group ID 1077  
  get 414  
real user ID 1077  
  get 450  
realloc 656, 992  
realpath 657  
record  
  delete in ISAM file 283  
record-oriented I/O 92, 1077  
reentrant functions 727, 729  
regcmp 661  
regex 661  
regexp 664  
regexp.h 975

- register process termination function 179
  - regular expression 171, 1077
  - regular expressions 664
    - comparisons 975
    - compile and execute 658, 661
    - declarations 975
  - regular file 1077
  - relative pathname 1077
  - release 1021
  - release date
    - operating system 852
  - release number
    - operating system 852
  - remainder 671
  - remainder from division 671
  - remainder value
    - compute for floating-point number 306
  - remote machine 1077
  - remove 672, 988
    - directory 681
    - files 672
    - link 856
  - remque 467, 673
  - rename 674, 988
    - file 674
  - reposition
    - file position indicator 356
  - reset file position indicator 701
  - resource
    - get limit 441
    - set limit 441
  - resource limit
    - set 711
  - restart behavior, system calls 737
  - restrictions, compared with XPG4 Version 2 2
  - result parameter, pointer 124
  - RETURN, regexp 664
  - return integer absolute value 166
  - return value
    - pointer 124
    - void \* 124
  - return values
    - variable for 258
  - rewind 677, 988
  - rewinddir 678, 933
  - rindex 679
  - rint, rintf, rintl 680
  - rmdir 681, 1040
  - root directory 1078
    - change 212
  - round off
    - floating-point number 305
  - round to nearest integer value 523, 541
  - round up
    - floating-point number 201
  - round up to next integer value 524, 542, 683
  - round, roundf, roundl 683
  - RTSIG\_MAX 807
- S**
- S\_IFBLK 1012
  - S\_IFCHR 1012
  - S\_IFDIR 1012
  - S\_IFIFO 1012
  - S\_IFMT 1012
  - S\_IFSOCK 1012
  - S\_IRGRP 1012
  - S\_IROTH 1012
  - S\_IRUSR 1012
  - S\_IRWXG 1012
  - S\_IRWXO 1012
  - S\_IRWXU 1012
  - S\_ISBLK() 1013
  - S\_ISCHR() 1013
  - S\_ISDIR() 1013
  - S\_ISFIFO() 1013
  - S\_ISUID 1012
  - S\_ISVTX 1012
  - S\_IWGRP 1012
  - S\_IWOTH 1012
  - S\_IWUSR 1012
  - S\_IXGRP 1012
  - S\_IXOTH 1012
  - S\_IXUSR 1012
  - sa\_flags 723, 980
  - sa\_handler 723, 980
  - sa\_mask 723, 980
  - SA\_NOCLDSTOP 723, 981

SA\_NOCLDWAIT 724  
safe functions 727  
SAM file 85  
saved set-group-ID 1078  
saved set-user-ID 1078  
sbrk 186, 684  
scalb 684  
scanf 345, 685, 988  
SCHAR\_MAX 963  
SCHAR\_MIN 963  
schedule alarm signal 172  
search  
    binary tree 829, 841  
    linear 543  
    linear table 543  
    sorted array 192  
    table of data 642  
search for first occurrence of a wide character  
    string 896  
search function  
    bsearch 192  
    hsearch 459  
    lfind 517  
    lsearch 543  
    qsort 642  
    tdelete 841  
    tfind 829, 841  
    tsearch 841  
    twalk 841, 846  
search table  
    destroy 459  
search tables 976  
search tree  
    binary 841  
search.h 976  
security attributes 1078  
security controls (extended) 1062  
seed for pseudo-random numbers  
    set 685  
seed48 242, 685, 992  
seek  
    first set bit 292  
SEEK\_CUR 985, 1037  
SEEK\_END 985, 1035, 1037  
SEEK\_SET 985, 1037  
seekdir 686, 933  
SEM\_NSEMS\_MAX 807  
SEM\_UNDO 1008  
SEM\_VALUE\_MAX 807  
semaphore  
    control operations 689  
    get ID 692  
    operations 694  
semaphore identifier (semid) 113  
semaphore structures 1008  
semaphores 112  
semctl 689  
semget 692  
semop 694  
send  
    message to STREAM 633  
    signal 643  
serial data transmission  
    interrupt 821  
session 1078  
session ID of terminal 820  
session leader 1078  
    controlling terminal 95  
session lifetime 1078  
set  
    alternative signal stack 732  
    file access and modification time 862  
    file access time 860  
    file mode creation mask 850  
    file modification time 860  
    group ID 709  
    interval timer 701, 848  
    process priority 708  
    seed for pseudo-random number  
        generator 755  
    user ID 710  
set up, user context 551  
set wide characters in wide character string 918  
SETALL 1008  
setbuf 698, 988  
setcontext 401, 699  
setgid 700, 1040  
setgrent 249, 701

setitimer 421, 701  
setjmp 702  
setjmp.h 977  
setkey 703, 992  
setlocale 704, 966  
setlogmask 706  
setpgid 707, 1040  
setpgrp 708  
setpwent 251, 708  
setregid 709  
setreuid 710  
setrlimit 441, 711  
setrlimit64 441, 711  
setsid 712, 1040  
setuid 713, 1040  
setutxent 253, 714  
SETVAL 1008  
setvbuf 714, 988  
shared memory 112, 116  
    attach 716  
    control operations 718  
    detach 720  
shared memory identifier (shmid) 113  
shared memory segment, create 721  
shell 1078  
SHM\_RDONLY 1010  
shmat 716  
shmctl 718  
shmdt 720  
shmget 721  
SHMLBA 1010  
SHRT\_MAX 963  
SHRT\_MIN 963  
sig\_atomic\_t 978  
SIG\_BLOCK 744, 981  
SIG\_DFL 726, 978  
SIG\_ERR 978  
SIG\_HOLD 978  
SIG\_IGN 726, 978  
SIG\_SETMASK 744, 981  
SIG\_UNBLOCK 744, 981  
SIGABRT 978  
sigaction 981  
    function 723  
    structure 723, 980  
sigaddset 731, 981  
SIGALRM 978  
sigaltstack 732  
SIGBUS 978  
SIGCHLD 978  
SIGCONT 979  
sigdelset 734, 981  
SIGDVZ 979  
sigemptyset 735, 981  
SIGEMT 979  
sigfillset 736, 981  
SIGFPE 979  
sighold 740  
SIGHUP 979  
sigignore 740  
SIGILL 979  
SIGINT 979  
siginterrupt 737  
SIGIO 979  
SIGIOT 979  
sigismember 738, 981  
SIGKILL 979  
siglongjmp 739  
signal 740, 981, 1079  
    add to a signal set 731  
    alternative stack 732  
    blocked 725  
    default signal handling 978  
    delete from signal set 734  
    delivered 724, 725  
    generated 725  
    pending 725, 743  
    send to calling process 643  
    send to process 510  
    send to process group 510  
    sent 725  
    set alternative stack 749  
    test for presence in signal set 738  
    wait for 750  
signal action  
    catch signal 726  
    default 726  
    ignore signal 726

- signal actions 726, 978
- signal handling 118
  - change 723
  - examine 723
  - examine and change 740
  - modify 746
- signal mask 725, 1079
  - change 744
  - examine 744
- signal numbers
  - symbolic name 978
- signal processing 111
- signal set
  - add signal to 731
  - delete signal from 734
  - initialize and empty 735
  - initialize and fill 736
  - test for a specific signal 738
- signal.h 111, 978
- signal-generating event 725
- signals
  - effects on functions 728
  - foreground process groups 94
  - header file 978
- siggam 394, 518, 743
- sigpause 740
- sigpending 743, 981
- SIGPIPE 979
- SIGPOLL 979
- sigprocmask 744, 981
- SIGPROF 979
- SIGPWR 979
- SIGQUEUE\_MAX 807
- SIGQUIT 979
- sigrelse 740
- SIGSEGV 979
- sigset 746
- sigset\_t 978
- sigsetjmp 747
- sigstack 749
- SIGSTOP 980
- sigsuspend 750, 981
- SIGSYS 980
- SIGTERM 980
- SIGTIM 980
- SIGTRAP 980
- SIGTSTP 980
- SIGTTIN 980
- SIGTTIN signals
  - conditions 95
- SIGTTOU 980
- SIGURG 980
- SIGUSR1 980
- SIGUSR2 978, 980
- SIGVTALRM 980
- SIGWINCH 980
- SIGXCPU 980
- SIGXFSZ 980
- sin 751
- sine function
  - compute 751
- sinh 170, 751
- size\_t 984, 986, 990, 993, 1019
- slash 1079
- sleep 752, 1040
- sort
  - array 192
  - table of data 642
- sorted array
  - binary search 192
- source file name
  - macro 300
- special character 1079
- special file 1079
  - make 573
- specify file orientation 379
- speed\_t 1025
- sprintf 321, 754, 988
- sqrt 754
- square root
  - compute 754
- srand 754, 992
- srand48 242, 755, 992
- srandom 755
- sscanf 345, 755, 988
- SSIZE\_MAX 963
- ssize\_t 1019



- stack
  - alternative signal stack 732
  - set signal stack 749
- standard constants
  - header file 1035
- standard data types, definitions 984
- standard error 78, 1079
  - write messages 622
- standard I/O
  - header file 985
  - lock by clients 398, 400
- standard I/O streams 78
  - variables 761
- standard input 78, 1079
- standard input mode 96
- standard input stream
  - read with format 345
- standard input/output
  - lock 303, 376
  - unlock 378
- standard library
  - header file 990
- standard output 78, 1079
- standard output stream
  - put a string on 637
- standard structures
  - header file 1035
- standard utilities 1079
- START 100
- stat 756
- stat64 756
- status, of open file 363
- status change, child process 878
- status information 115
- statvfs 366, 760
- statvfs64 366, 760, 1014
- stdarg.h 982
- stddef.h 984
- stderr 761, 985
- STDERR\_FILENO 761, 1039
- stdin 761, 985
- STDIN\_FILENO 761, 1039
- stdio.h 985
- stdlib.h 990
- stdout 761, 985
- STDOUT\_FILENO 761, 1039
- step 664, 762, 975
  - regexp 665
- STOP 100
- stop signal 725
- strcasecmp 762, 778
- strcat 763, 993
- strchr 763, 993
- strcmp 764, 993
- strcoll 765, 993
- strcpy 766, 993
- strcspn 766, 993
- strdup 767
- STREAM
  - multiplex I/O 624
  - send message 633
- stream 761, 1079
  - associate with file descriptor 286
  - close 274
  - creating 79
  - flush 274, 290
  - flush and reopen 342
  - fully buffered 77
  - get byte from 396
  - get file position indicator 295, 370
  - get wide character string 299
  - get wide character string from 297
  - line buffered 77
  - open 312
  - push back byte 853
  - put byte on 334, 630
  - put word on 639
  - read with format 345
  - reposition a file position indicator 356
  - reset file position indicator 677
  - set file position indicator 361
  - test end-of-file indicator 288
  - test error indicator 289
  - unbuffered 77
- STREAM\_MAX 807, 958
- stream-oriented I/O 91, 1080
- STREAMS file
  - read message 425, 434

- STREAMS interface 995
- strerror 768, 993
- strfill 769
- strfmon 770, 970
- strftime 774, 1032
- string 704
  - convert to double-precision floating-point number 787
  - convert to floating-point number 180
  - convert to integer 181
  - convert to long integer 182, 790
  - convert to long long integer 183, 792
  - convert to lowercase letters 777
  - convert to unsigned long 794
  - convert to unsigned long long integer 796
  - convert to uppercase letters 798
  - copy 766
  - duplicate 767
  - encode using algorithms 226
  - formatted output to 754
  - get from standard input stream 446
  - get from stream 296
  - get suboptions 448
  - put on a stream 336
  - put on standard output stream 637
  - read formatted input 755
  - read with format 345
  - read without echo 431
  - scan for characters 763
  - search function 763
  - split into tokens 788, 789
  - transform based on LC\_COLLATE 799
- string comparison 762
- string length
  - get 777
- string operations 993, 994
- string value
  - system variable 220
- string.h 993
- strings
  - compare 764, 778
  - compare using collating sequence 765
  - concatenate 763
  - convert ASCII to EBCDIC 173
  - convert to date and time 782
  - encode blockwise 248
  - strlen 777, 993
  - strlower 777
  - strncasecmp 762
  - strncat 778, 993
  - strncmp 779, 993
  - strncpy 780, 993
  - strpbrk 781, 993
  - strptime 782, 1032
  - strrchr 785, 993
  - strspn 786, 993
  - strstr 786, 993
  - strtod 787, 992
  - strtok 788, 993
  - strtok\_r 789, 993
  - strtol 790, 992
  - strtoll 792
  - strtoul 794, 992
  - strtoull 796
  - struct termios 1025
  - structure
    - file times 1018
    - interprocess communication 1000
    - of message display 950
    - of VFS file system information 1014
    - sigaction 723, 980
    - stxlt 227
  - structure for system names
    - header file 1021
  - structures
    - for message queues 1004
    - standard definition 1035
  - strupper 798
  - strxfrm 799, 993
  - stxlt
    - structure 227
  - STXIT event classes 118
  - STXIT routine 117
    - define 227
    - free programming 121
    - implementation using library functions 118
    - structure 121
  - stxlt.h 227

- suboptions from string 448
  - substring
    - copy 769
    - find in string 786
  - substring length
    - get 786
  - substrings
    - compare 779
    - concatenate 778
  - superblock, update 804
  - supplementary group ID 1080
    - get 419
  - SUSP 100
  - suspend (thread) 601
  - suspend process 859
  - suspend thread 601
  - suspended job 1080
  - swab 800, 1040
  - swapcontext 551, 800
  - swprintf 380, 801
  - swscanf 386, 801
  - symbolic link
    - to file 802
  - symlink 802
  - sync 804
  - synchronize, memory 598
  - sys/ipc.h 1000
  - sys/msg.h 1004
  - sys/sem.h 1008
  - sys/shm.h 1010
  - sys/stat.h 1011
  - sys/times.h 1018
  - sys/types.h 1019
  - sys/utsname.h 1021
  - sys/wait.h 1022
  - sysconf 805, 1040
  - SYSDTA 82
  - sysfs 809
  - SYSLST 84
  - sysname 1021
  - SYSOUT 83
  - system 811, 992, 1080
    - error codes 934
    - UNIX 1081
  - system call 1080
  - system calls
    - interrupt behavior 737
  - system command
    - execute 811
    - execution in BS2000 190
  - system identifier for inter-process communication 113
  - system process 1080
  - system scheduling priority 1080
  - system variable
    - string value 220
    - value of 805
- T**
- T\_FMT 954
  - T\_FMT\_AMPM 954
  - TAB0 1027
  - TAB1 1027
  - TAB2 1027
  - TAB3 1027
  - TABDLY 104, 1027
  - table of data
    - sort 642
  - tabulator 110
  - tan 814
  - tangent
    - compute 814
  - tanh 814
  - tar.h 1023
  - task
    - time used (by CPU) 222
  - tcdrain 815
  - tcflag\_t 102, 1025
  - tcflow 816
    - constants 1030
  - tcflush 817
    - constants 1029
  - tcgetattr 818
  - tcgetpgrp 819, 1040
  - tcgetsid 820
  - TCIFLUSH 1029
  - TCIOFF 1030
  - TCIOFLUSH 1029

TCION 1030  
TCOFLUSH 1029  
TCOOFF 1030  
TCOON 1030  
TCSADRAIN 1029  
TCSAFLUSH 1029  
TCSANOW 1029  
tcsendbreak 821  
tcsetattr 822  
    constants 1029  
tcsetpgrp 824, 1040  
tdelete 824, 841  
tell 825  
telldir 826, 933  
tempnam 827, 988  
temporary file  
    create 833  
    create base name 834  
    create pathname 827  
temporary file name  
    make 577  
temporary PAM file 94  
terminal 1080  
    controlling 95  
    find path name 844  
    find pathname 843  
    generate pathname 231  
    get session ID 820  
    opening a device file 94  
terminal device  
    test 486  
terminal interface (general) 94  
terminal parameters  
    get 818  
    set 822  
termios 102  
termios.h 1025  
test  
    control wide character 498  
    decimal digit 488  
    EBCDIC character 489  
    for „initial conversion“ state 556  
    for 7-bit ASCII character 485  
    for a hexadecimal digit wide character 507  
    for a printing wide character 503  
    for alphabetic character 484  
    for alphabetic wide character 497  
    for alphanumeric character 483  
    for alphanumeric wide character 496  
    for control character 487  
    for decimal digit wide character 500  
    for hexadecimal digit 508  
    for printing character 492  
    for punctuation character 493  
    for punctuation wide character 504  
    for terminal device 486  
    for uppercase wide character 506  
    for visible character 490  
    for visible wide character 501  
    for white-space character 494  
    for white-space wide character 505  
    lowercase letter 491  
    lowercase wide character 502  
    uppercase letter 495  
text file 1081  
    make 573  
tfind 829, 841  
TGEXEC 1024  
TGREAD 1024  
TGWRITE 1024  
thousands\_sep 965  
THOUSEP 955  
time 830, 1032  
time data types 1031  
time functions 16  
TIME link option 16  
time since Epoch, get 830  
time types 1016  
time used  
    by process 214  
    by task 222  
time.h 1031  
time\_t 1019  
TIMER\_MAX 807  
times 831  
timezone 832, 1032  
    set conversion information 847  
tm\_hour 1031

tm\_isdst 1031  
 tm\_mday 1031  
 tm\_min 1031  
 tm\_mon 1031  
 tm\_sec 1031  
 tm\_wday 1031  
 tm\_yday 1031  
 tm\_year 1031  
 TMP\_MAX 964, 985  
 tmpfile 833, 988  
 tmpfile64 988  
 tmpnam 834, 988  
 toascii 835, 932  
 toebcdic 836  
 TOEXEC 1024  
 tolower 837, 932  
 TOREAD 1024  
 TOSTOP 1029  
 toupper 838, 932  
 towctrans 838  
 tolower 839, 1045  
 TOWRITE 1024  
 toupper 839, 1045  
 transmission of data  
     restart 816  
     suspend 816  
     wait for output 815  
 traverse (walk) file tree 377  
 traverse binary search tree 846  
 traverse file tree 603  
 tree, nomenclature for search tree 842  
 truncate 375, 840  
 truncate64 840  
 tsearch 841  
 TSGID 1024  
 TSN, get 450  
 TSUID 1024  
 TSVTX 1024  
 TTY\_NAME\_MAX 807  
 ttyname 843, 1041  
 ttyname\_r 844  
 ttyslot 845  
 TUEXEC 1024  
 TUREAD 1024  
 TUWRITE 1024  
 twalk 841, 846  
 types for floating-point numbers 947  
 types for monetary values 970  
 tzname 846, 1032  
 TZNAME\_MAX 807, 958  
 tzset 847, 1032  
  
**U**  
 ualarm 848  
 UCHAR\_MAX 963  
 uid\_t 1019  
 UINT\_MAX 963  
 UL\_GETFSIZE 1034  
 UL\_SETFSIZE 1034  
 ulimit 849  
 ulimit.h 1034  
 ULONG\_MAX 963  
 umount 851  
 uname 852  
 unbuffered (stream) 77  
 UNGETC  
     regexp 664  
 ungetc 853, 988  
 ungetwc 855, 1046  
 unistd.h 1035  
 UNIX system 1081  
 unlink 856, 1041  
 unlock pseudoterminal pair 858  
 unlock standard input/output 378  
 unlockpt 858, 992  
 unmap memory pages 600  
 unsafe functions 728  
 update  
     linear 543  
 uppercase letter  
     convert to lowercase 836  
     test 495  
 uppercase wide character  
     test 506  
 upshifting 1081  
 user 1081, 1082  
     enter into user catalog 636  
     user accounting file 253

- user administration 1081
- user attributes 1082
- user catalog 1082
  - enter user 636
  - manage 251
  - read user data sequentially 436
- user context 1033
  - display or modify 401
  - modify 699
  - set up 551
  - swap 800
- user data
  - read from user catalog 436
- user database 1082
- user entry
  - utmp file 845
- user group 1082
- user ID 1082
  - effective 414
  - get 414, 439, 440
  - real 450
  - set 710, 713
- user ID and group ID 114
- user name 1082
  - get 437, 438
- user privileges 1082
- USER-ID 1070
- user-specific locale 70
- USHRT\_MAX 963
- usleep 859
- USLOCA 70
- USLOCC 70
- utime 860
- utime.h 1042
- utimes 862
- utmp file
  - find user entry 845
- utmpx
  - reset pointer 714
- utmpx entry
  - write 638
- utsname.h 1021

## V

- V1CTYPE 58
- va\_arg 864
- va\_end 865
- va\_list 986
- va\_start 866
- valloc 867
- value
  - of environment variable 413
  - of locale 607
  - of system variable 805
  - pathname variable 959
  - system variable 220
- varargs.h 1044
- variable 1082
  - for command options 428, 616
  - for difference between local time and UTC 832
  - for error return values 258
  - for standard I/O streams 761
  - for time zone 173
- variable argument list
  - end 865
  - formatted output 869
  - handling 1044
  - initialize 866
  - process 864
- vector
  - c\_cc 109
- verify program assertion 930
- version 1021
- vfork 868
- fprintf 869, 988
- vwprintf 380
- virtual memory
  - generate process in 868
- visible character
  - test 490
- visible wide character
  - test 501
- void \*, return value 124
- vprintf 869, 870, 988
- vsprintf 869, 871, 988
- vswprintf 380, 872

VT0 1027  
VT1 1027  
VTDLY 105, 1027  
vwprintf 380, 872

**W**

W\_OK 1037  
wait 873, 1022  
    for child process to stop or terminate 873  
waitid 878  
waitpid 873, 1022  
wchar.h 1045  
wchar\_t 984, 990  
WCOREDUMP 1022  
wctomb 880  
wscat 881, 1046  
wchr 882, 1046  
wscmp 883, 1046  
wscoll 884, 1046  
wscpy 885, 1046  
wscspn 886, 1046  
wcsftime 887, 1046  
wcslen 888, 1046  
wscncat 889, 1046  
wscncmp 890, 1046  
wscncpy 891, 1046  
wscpbrk 892, 1046  
wscrchr 893, 1046  
wscrtombs 894  
wcsspn 895, 1046  
wcssr 896  
wcstod 897, 1046  
wcstok 899, 1046  
wcstol 900, 1046  
wcstoll 902  
wcstombs 904, 992  
wcstoul 905, 1046  
wcstoull 907  
wswcs 909, 1046  
wswidth 910, 1046  
wscxfrm 911, 1046  
wctob 912  
wctomb 912, 992  
wctrans 913

wctype 914, 1046  
wctype.h 1047  
wctype\_t 1045  
wcwidth 915, 1046  
WEOF 15, 1046, 1047  
WEXITSTATUS 990, 1022  
white space 1082  
white-space character  
    test 494  
white-space characters 84  
white-space wide character  
    test 505  
wide character  
    convert to character 912  
    formatted output 919  
    from standard input stream 454  
    get from stream 297, 453  
    push back onto input stream 855  
    put on standard output stream 641  
    put on stream 640  
    test for specified class 499  
wide character class  
    define 914  
wide character code 1083  
    put on stream 337  
wide character in string  
    get first occurrence 892  
    get last occurrence 893  
wide character string 1083  
    convert to character string 904  
    convert to double-precision floating-point  
        number 897  
    convert to long integer 900  
    convert to long long integer 902  
    convert to multi-byte character string 894  
    convert to unsigned long 905  
    convert to unsigned long long 907  
    copy 885, 917  
    copy in overlapping area 918  
    get from stream 299  
    get length 888  
    put on a stream 339  
    scan for wide characters 882  
    search for first occurrence of 896

- wide character string (continued)
    - search for wide character in 916
    - split into tokens 899
    - transform 911
  - wide character strings
    - compare 883, 884, 917
    - concatenate 881
  - wide character substring
    - copy 891
    - find in wide character string 909
    - get length 895
  - wide character substrings
    - compare 890
    - concatenate 889
  - wide characters
    - convert to lowercase 839
    - convert to multi-byte characters 880
    - convert to uppercase 839
    - data types 1045
    - define mappings 913
    - formatted output 380, 801
    - map 838
  - WIFCONTINUED 1022
  - WIFEXITED 990, 1022
  - WIFSIGNALED 1022
  - WIFSIGNALED() 990
  - WIFSTOPPED 1022
  - WIFSTOPPED() 990
  - wint\_t 1045
  - wmemchr 916
  - wmemcmp 917
  - wmemcpy 917
  - wmemmove 918
  - wmemset 918
  - WNOHANG 990, 1022
  - word
    - read from stream 452
  - WORD\_BIT 963
  - working directory
    - change 207
  - wprintf 380, 919
  - write 920, 1041
    - bytes to a file 920
    - formatted output on output stream 321
    - formatted output on standard output stream 629
    - formatted output to string 754
  - write diagnostic output 78
  - write utmpx entry 638
  - writev 926
  - wscanf 386, 927
  - WSTOPSIG 1022
  - WSTOPSIG() 990
  - WTERMSIG 1022
  - WTERMSIG() 990
  - WUNTRACED 990, 1022
- ## X
- X/Open Portability Guide 1
  - X\_OK 1037
  - XCASE 1029
  - XPG4 Version 2 1
- ## Y
- y0, y1, yn 927
  - YESEXPR 955
  - YESSTR 955
- ## Z
- zombie process 264, 1083



---

# Contents

|                                                              |           |
|--------------------------------------------------------------|-----------|
| <b>Preface</b> .....                                         | <b>1</b>  |
| Target group .....                                           | 3         |
| Summary of contents .....                                    | 3         |
| Organization of the POSIX documentation .....                | 4         |
| Notational conventions .....                                 | 6         |
| Changes since the last manual .....                          | 7         |
| Changes since the C library manual V2.2A .....               | 9         |
| <b>The C programming interface</b> .....                     | <b>11</b> |
| System requirements .....                                    | 11        |
| Components of the C library .....                            | 12        |
| Header files .....                                           | 12        |
| Functions and macros .....                                   | 13        |
| 64-bit functions to support NFS V3.0 .....                   | 14        |
| POSIX thread support in the C runtime library .....          | 14        |
| Wide characters and multi-byte characters .....              | 15        |
| Time functions .....                                         | 16        |
| Setting the time zone for POSIX time functions .....         | 17        |
| Scope of the supported C library .....                       | 18        |
| Selecting functionality .....                                | 40        |
| Range of functions extended by the POSIX functionality ..... | 40        |
| BS2000 functionality .....                                   | 42        |
| Selecting the file system and the system environment .....   | 42        |
| – Associating the I/O streams .....                          | 42        |
| – Setting the PROGRAM_ENVIRONMENT variable .....             | 43        |
| – Syntax in the source program .....                         | 44        |
| Portability .....                                            | 45        |
| Name space .....                                             | 46        |
| Character sets .....                                         | 47        |
| Portable character set .....                                 | 47        |
| Character classes .....                                      | 52        |

|                                                                       |     |
|-----------------------------------------------------------------------|-----|
| Locale .....                                                          | 53  |
| Predefined locales .....                                              | 55  |
| – Locale files .....                                                  | 55  |
| – POSIX or C locale .....                                             | 56  |
| – V1CTYPE .....                                                       | 58  |
| – V2CTYPE .....                                                       | 59  |
| – GERMANY .....                                                       | 59  |
| – De.EDF04F and De.EDF04F@euro .....                                  | 60  |
| User-specific locales .....                                           | 70  |
| Environment variables .....                                           | 71  |
| File processing .....                                                 | 74  |
| Streams .....                                                         | 77  |
| – Buffering streams .....                                             | 77  |
| – Disassociating a file from a stream .....                           | 77  |
| – Standard I/O streams .....                                          | 78  |
| Interaction of file descriptors and streams .....                     | 79  |
| Support for file systems in ASCII .....                               | 81  |
| BS2000 file processing .....                                          | 81  |
| – BS2000 system files .....                                           | 82  |
| – White-space characters .....                                        | 84  |
| – Cataloged disk files (SAM, ISAM, PAM) .....                         | 85  |
| – Default values and possible modifications for file attributes ..... | 86  |
| – K and NK block formats .....                                        | 89  |
| – K and NK-ISAM files .....                                           | 90  |
| – Support for the DIV access method .....                             | 91  |
| – Notes on stream-oriented I/O .....                                  | 91  |
| – Notes on record-oriented I/O .....                                  | 92  |
| Temporary PAM files in virtual memory (INCORE files) .....            | 94  |
| General terminal interface .....                                      | 94  |
| Opening a terminal device file .....                                  | 94  |
| Process groups .....                                                  | 94  |
| – The controlling terminal .....                                      | 95  |
| – Terminal access control .....                                       | 95  |
| – Input processing and reading data .....                             | 96  |
| – Canonical mode input processing .....                               | 97  |
| – Non-canonical mode input processing .....                           | 98  |
| – Writing data and output processing .....                            | 99  |
| – Special characters .....                                            | 99  |
| – Modem disconnect .....                                              | 101 |
| – Closing a terminal device file .....                                | 101 |

---

|                                                           |            |
|-----------------------------------------------------------|------------|
| Settable parameters                                       | 102        |
| – The termios structure                                   | 102        |
| – Input modes                                             | 102        |
| – Output modes                                            | 104        |
| – Control modes                                           | 106        |
| – Local modes                                             | 107        |
| – Special control characters                              | 109        |
| Block terminal support                                    | 110        |
| Support for BS2000 consoles                               | 110        |
| Process control                                           | 111        |
| Signals                                                   | 111        |
| Interprocess communication                                | 112        |
| – General description                                     | 112        |
| – Shared memory                                           | 116        |
| Contingency and STXIT routines                            | 117        |
| – The C library functions alarm(), raise(), and signal()  | 118        |
| – STXIT contingency routines                              | 118        |
| – Event-driven routines                                   | 118        |
| – Free use of contingency routines                        | 119        |
| – Free use of STXIT contingency routines                  | 121        |
| Thread-safe C runtime library by supporting POSIX threads | 122        |
| Programming notes                                         | 124        |
| Return values and result parameters                       | 124        |
| Error handling                                            | 126        |
| Debugging options                                         | 127        |
| <b>Functions and variables arranged by theme</b>          | <b>129</b> |
| File processing                                           | 129        |
| I/O on terminal                                           | 133        |
| Processes                                                 | 135        |
| Functions to support POSIX threads                        | 140        |
| Memory management and memory operations                   | 146        |
| System environment                                        | 147        |
| Characters and strings                                    | 148        |
| Conversion of entities                                    | 152        |
| Regular expressions                                       | 153        |
| Time functions                                            | 153        |
| Math functions                                            | 154        |
| Search and sort procedures                                | 157        |

|                                                                                        |            |
|----------------------------------------------------------------------------------------|------------|
| Terminal interface and data transmissions . . . . .                                    | 158        |
| Database functions . . . . .                                                           | 159        |
| List processing . . . . .                                                              | 159        |
| POSIX-IO macros . . . . .                                                              | 160        |
| <b>Functions and variables in alphabetical order (a - m)</b> . . . . .                 | <b>161</b> |
| a64l, l64a - convert string to 32-bit integer . . . . .                                | 163        |
| abort - abort process . . . . .                                                        | 165        |
| abs - return integer absolute value . . . . .                                          | 166        |
| access - check access permissions for file . . . . .                                   | 167        |
| acos - arc cosine function . . . . .                                                   | 169        |
| acosh, asinh, atanh - inverse hyperbolic functions . . . . .                           | 170        |
| advance - pattern match given compiled regular expression . . . . .                    | 171        |
| alarm - schedule alarm signal . . . . .                                                | 172        |
| altzone - variable for time zone ( <i>extension</i> ) . . . . .                        | 173        |
| ascii_to_ebcdic - convert ASCII string to EBCDIC string ( <i>extension</i> ) . . . . . | 173        |
| asctime - convert date and time to string . . . . .                                    | 174        |
| asctime_r - convert date and time to string (thread-safe) . . . . .                    | 175        |
| asin - arc sine function . . . . .                                                     | 176        |
| asinh - inverse hyperbolic sine function . . . . .                                     | 176        |
| assert - output diagnostic messages . . . . .                                          | 177        |
| atan - arc tangent function . . . . .                                                  | 177        |
| atan2 - arc tangent of x/y . . . . .                                                   | 178        |
| atanh - inverse hyperbolic tangent function . . . . .                                  | 178        |
| atexit - register function to run at process termination . . . . .                     | 179        |
| atof - convert string to double-precision number . . . . .                             | 180        |
| atoi - convert string to integer . . . . .                                             | 181        |
| atol - convert string to long integer . . . . .                                        | 182        |
| atoll - convert string to long long integer (long long int) . . . . .                  | 183        |
| basename - return last element of pathname . . . . .                                   | 184        |
| bcmp - compare memory areas . . . . .                                                  | 185        |
| bcopy - copy memory area . . . . .                                                     | 185        |
| brk, sbrk - modify size of data segment . . . . .                                      | 186        |
| bs2exit - program termination with MONJV ( <i>BS2000</i> ) . . . . .                   | 188        |
| bs2fstat - get BS2000 file names from catalog ( <i>BS2000</i> ) . . . . .              | 189        |
| bs2system - execute BS2000 command ( <i>extension</i> ) . . . . .                      | 190        |
| bsd_signal - simplified signal handling . . . . .                                      | 191        |
| bsearch - conduct binary search of sorted array . . . . .                              | 192        |
| btowc - (one byte) convert multi-byte character to wide character . . . . .            | 193        |
| bzero - initialize memory with X'00' . . . . .                                         | 194        |
| cabs - calculate absolute value of complex number ( <i>BS2000</i> ) . . . . .          | 194        |

|                                                                                                                                                     |     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| calloc - allocate memory                                                                                                                            | 195 |
| catclose - close message catalog                                                                                                                    | 196 |
| catgets - read message                                                                                                                              | 197 |
| catopen - open message catalog                                                                                                                      | 198 |
| cbrt - cube root                                                                                                                                    | 199 |
| cdisco - disconnect contingency routine ( <i>BS2000</i> )                                                                                           | 200 |
| ceil, ceilf, ceill - round up floating-point number                                                                                                 | 201 |
| cenaco - define contingency routine ( <i>BS2000</i> )                                                                                               | 202 |
| cfgetspeed - get input baud rate                                                                                                                    | 204 |
| cfgetospeed - get output baud rate                                                                                                                  | 204 |
| cfsetspeed - set input baud rate                                                                                                                    | 205 |
| cfsetospeed - set output baud rate                                                                                                                  | 206 |
| chdir - change working directory                                                                                                                    | 207 |
| chmod - change mode of file                                                                                                                         | 208 |
| chown - change owner and group of file                                                                                                              | 210 |
| chroot - change root directory                                                                                                                      | 212 |
| clearerr - clear end-of-file and error indicators                                                                                                   | 213 |
| clock - report CPU time used by a process                                                                                                           | 214 |
| close - close file                                                                                                                                  | 215 |
| closedir - close directory                                                                                                                          | 216 |
| closelog, openlog, setlogmask, syslog - control system log                                                                                          | 217 |
| compile - produce compiled regular expression                                                                                                       | 219 |
| confstr - get string value of system variable                                                                                                       | 220 |
| cos - cosine function                                                                                                                               | 221 |
| cosh - hyperbolic cosine function                                                                                                                   | 221 |
| cputime - calculate CPU time used by current task ( <i>BS2000</i> )                                                                                 | 222 |
| creat - create new file or overwrite existing one                                                                                                   | 223 |
| crypt - encode strings using algorithms                                                                                                             | 226 |
| cstxlt - define STXIT routine ( <i>BS2000</i> )                                                                                                     | 227 |
| ctermid - generate pathname for controlling terminal                                                                                                | 231 |
| ctime - convert date and time to string                                                                                                             | 232 |
| ctime_r - thread-safe conversion of date and time to string                                                                                         | 233 |
| userid - get login name                                                                                                                             | 234 |
| __DATE__ - macro for compilation date                                                                                                               | 235 |
| daylight - daylight savings time variable                                                                                                           | 235 |
| dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey,<br>dbm_nextkey, dbm_open, dbm_store - functions for managing dbm databases | 236 |
| difftime - compute difference between two calendar time values                                                                                      | 239 |
| dirname - parent directory of pathname                                                                                                              | 240 |
| div - divide with integers                                                                                                                          | 241 |

|                                                                                             |     |
|---------------------------------------------------------------------------------------------|-----|
| drand48 - generate pseudo-random numbers between 0.0 and 1.0                                | 242 |
| dup, dup2 - duplicate file descriptor                                                       | 244 |
| ebcdic_to_ascii - convert EBCDIC string to ASCII string ( <i>extension</i> )                | 245 |
| ecvt, fcvt, gcvt - convert floating-point number to string                                  | 246 |
| _edt - call EDT ( <i>BS2000</i> )                                                           | 248 |
| encrypt - encode strings blockwise                                                          | 248 |
| endgrent, getgrent, setgrent - group management                                             | 249 |
| endpwent, getpwent, setpwent - manage user catalog                                          | 251 |
| endutxent, getutxent, getutxid, getutxline, pututxline, setutxent -<br>manage utmpx entries | 253 |
| environ - external variable for environment                                                 | 256 |
| erand48 - generate pseudo-random numbers between 0.0 and 1.0 with<br>initialization value   | 256 |
| erf, erfc - error and complementary error functions                                         | 257 |
| errno - variable for error return values                                                    | 258 |
| exec: execl, execlv, execlx, execve, execlp, execvp - execute file                          | 259 |
| exit, _exit - terminate process                                                             | 264 |
| exp - use exponential function                                                              | 267 |
| expm1 - compute exponential function                                                        | 267 |
| fabs - compute absolute value of floating-point number                                      | 268 |
| fattach - assign file descriptor under STREAMS to object in name space of<br>file system    | 268 |
| fchdir - change current directory                                                           | 270 |
| fchmod - change mode of file                                                                | 271 |
| fchown - change owner or group of file                                                      | 273 |
| fclose - close stream                                                                       | 274 |
| fcntl - control open file                                                                   | 276 |
| fcvt - convert floating-point number to string                                              | 282 |
| FD_CLR, FD_ISSET, FD_SET, FD_ZERO - macros for synchronous<br>I/O multiplexing              | 282 |
| fdelrec - delete record in ISAM file ( <i>BS2000</i> )                                      | 283 |
| fdetach - cancel assignment to STREAMS file                                                 | 284 |
| fdopen - associate stream with file descriptor                                              | 286 |
| feof - test end-of-file indicator on stream                                                 | 288 |
| ferror - test error indicator on stream                                                     | 289 |
| fflush - flush stream                                                                       | 290 |
| ffs - seek first set bit                                                                    | 292 |
| fgetc - get byte from stream                                                                | 293 |
| fgetpos - get current value of file position indicator in stream                            | 295 |
| fgets - get string from stream                                                              | 296 |
| fgetwc - get wide character string from stream                                              | 297 |

|                                                                                                |     |
|------------------------------------------------------------------------------------------------|-----|
| fgetws - get wide character string from stream                                                 | 299 |
| __FILE__ - macro for source file names                                                         | 300 |
| fileno - get file descriptor                                                                   | 300 |
| flocate - set file position indicator in ISAM file ( <i>BS2000</i> )                           | 301 |
| flockfile, ftrylockfile, funlockfile - functions for locking standard input/output             | 303 |
| floor, floorf, floorl- round off floating point number                                         | 305 |
| fmod - compute floating-point remainder value function                                         | 306 |
| fmsg - output message to stderr and/or system console                                          | 307 |
| fopen - open stream                                                                            | 312 |
| fork - create new process                                                                      | 318 |
| fpathconf - get value of pathname variable                                                     | 320 |
| fprintf, printf, sprintf - write formatted output on output stream                             | 321 |
| fputc - put byte on stream                                                                     | 334 |
| fputs - put string on stream                                                                   | 336 |
| fputwc - put wide-character code on stream                                                     | 337 |
| fputws - put wide character string on stream                                                   | 339 |
| fread - read binary data                                                                       | 340 |
| free - free allocated memory                                                                   | 341 |
| freopen - flush and reopen stream                                                              | 342 |
| frexp - extract mantissa and exponent from double precision number                             | 344 |
| fscanf, scanf, sscanf - read formatted input                                                   | 345 |
| fseek - reposition file position indicator in stream                                           | 356 |
| fsetpos - set file position indicator for stream to current value                              | 361 |
| fstat - get file status of open file                                                           | 363 |
| fstatvfs, statvfs - read file system information                                               | 366 |
| fsync - synchronize changes to file                                                            | 369 |
| ftell - get current value of file position indicator for stream                                | 370 |
| ftime - get date and time                                                                      | 372 |
| ftok - interprocess communication                                                              | 374 |
| ftruncate, truncate - set file to specified length                                             | 375 |
| ftrylockfile - lock standard input/output                                                      | 376 |
| ftw - traverse (walk) file tree                                                                | 377 |
| funlockfile - unlock standard input/output                                                     | 378 |
| fwide - specify file orientation                                                               | 379 |
| fwprintf, swprintf, vfwprintf, vswprintf, vwprintf, wprintf - output formatted wide characters | 380 |
| fwscanf, swscanf, wscanf - formatted read                                                      | 386 |
| fwrite - output binary data                                                                    | 392 |
| gamma - compute logarithm of gamma function                                                    | 394 |
| garbcoll - release memory space to system ( <i>BS2000</i> )                                    | 395 |

|                                                                                                                               |     |
|-------------------------------------------------------------------------------------------------------------------------------|-----|
| gcvt - convert floating-point number to string . . . . .                                                                      | 395 |
| getc - get byte from stream . . . . .                                                                                         | 396 |
| getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked -<br>standard I/O with explicit lock by the client . . . . . | 398 |
| getchar - get byte from standard input stream . . . . .                                                                       | 399 |
| getchar_unlocked - standard input with explicit lock by the client . . . . .                                                  | 400 |
| getcontext, setcontext - display or modify user context . . . . .                                                             | 401 |
| getcwd - get pathname of current working directory . . . . .                                                                  | 403 |
| getdate - convert time and date to user format . . . . .                                                                      | 405 |
| getdents - convert directory entries . . . . .                                                                                | 410 |
| getdtablesize - get size of descriptor table . . . . .                                                                        | 412 |
| getegid - get effective group ID of process . . . . .                                                                         | 412 |
| getenv - get value of environment variable . . . . .                                                                          | 413 |
| geteuid - get effective user ID of process . . . . .                                                                          | 414 |
| getgid - get real group ID of process . . . . .                                                                               | 414 |
| getgrent - get group file entry . . . . .                                                                                     | 414 |
| getgrgid - get group file entry for group ID . . . . .                                                                        | 415 |
| getgrgid_r - get group file entry for group ID (thread-safe) . . . . .                                                        | 416 |
| getgrnam - get group file entry for group name . . . . .                                                                      | 417 |
| getgrnam_r - get group file entry for group name (thread-safe) . . . . .                                                      | 418 |
| getgroups - get supplementary group IDs . . . . .                                                                             | 419 |
| gethostid - get ID of current host . . . . .                                                                                  | 420 |
| gethostname - get name of current host . . . . .                                                                              | 420 |
| getitimer, setitimer - read or set . . . . .                                                                                  | 421 |
| getlogin - get login name . . . . .                                                                                           | 423 |
| getlogin_r - get login name (thread-safe) . . . . .                                                                           | 424 |
| getmsg - get message from STREAMS file . . . . .                                                                              | 425 |
| getopt, optarg, optind, opterr, optopt - command option parsing . . . . .                                                     | 428 |
| getpagesize - get current page size . . . . .                                                                                 | 430 |
| getpass - read string of characters without echo . . . . .                                                                    | 431 |
| getpgid - get process group ID . . . . .                                                                                      | 432 |
| getpgmname - get program name ( <i>BS2000</i> ) . . . . .                                                                     | 432 |
| getpgrp - get process group ID . . . . .                                                                                      | 433 |
| getpid - get process ID . . . . .                                                                                             | 433 |
| getpmsg - get message from STREAMS file . . . . .                                                                             | 434 |
| getppid - get parent process ID . . . . .                                                                                     | 434 |
| getpriority, setpriority - get or set process priority . . . . .                                                              | 435 |
| getpwent - read user data from user catalog . . . . .                                                                         | 436 |
| getpwnam - get user name . . . . .                                                                                            | 437 |
| getpwnam_r - get user name (thread-safe) . . . . .                                                                            | 438 |



---

|                                                                       |     |
|-----------------------------------------------------------------------|-----|
| getpwuid - get user ID                                                | 439 |
| getpwuid_r - get user ID (thread-safe)                                | 440 |
| getrlimit, setrlimit - get or set limit for resource                  | 441 |
| getrusage - get information on usage of resources                     | 445 |
| gets - get string from standard input stream                          | 446 |
| getsid - get process group ID                                         | 447 |
| getsubopt - get suboptions from string                                | 448 |
| gettimeofday - read current time of day                               | 449 |
| gettsn - get TSN (task sequence number) ( <i>BS2000</i> )             | 450 |
| getuid - get real user ID                                             | 450 |
| getutxent, getutxid, getutxline - get utmpx entry                     | 451 |
| getw - read word from stream                                          | 452 |
| getwc - get wide character from stream                                | 453 |
| getwchar - get wide character from standard input stream              | 454 |
| getwd - get pathname of current working directory                     | 455 |
| gmatch - global pattern matching ( <i>extension</i> )                 | 455 |
| gmtime - convert date and time to UTC                                 | 456 |
| gmtime_r - convert date and time to UTC (thread-safe)                 | 457 |
| grantpt - grant access to the slave pseudoterminal                    | 458 |
| hsearch, hcreate, hdestroy - manage hash tables                       | 459 |
| hypot - Euclidean distance function                                   | 460 |
| iconv - code conversion function                                      | 461 |
| iconv_close - deallocate code conversion descriptor                   | 462 |
| iconv_open - allocate code conversion descriptor                      | 463 |
| ilogb - get exponent part of floating-point number                    | 464 |
| index - get first occurrence of character in string                   | 464 |
| initstate, random, setstate, srandom - generate pseudo-random numbers | 465 |
| insque, remque - Insert element in queue or remove element from queue | 467 |
| ioctl - control devices and STREAMS                                   | 468 |
| isalnum - test for alphanumeric character                             | 483 |
| isalpha - test for alphabetic character                               | 484 |
| isascii - test for 7-bit ASCII character                              | 485 |
| isastream - test file descriptor                                      | 486 |
| isatty - test for terminal device                                     | 486 |
| iscntrl - test for control character                                  | 487 |
| isdigit - test for decimal digit                                      | 488 |
| isebcdic - test for EBCDIC character ( <i>BS2000</i> )                | 489 |
| isgraph - test for visible character                                  | 490 |
| islower - test for lowercase letter                                   | 491 |
| isnan - test for NaN (not a number)                                   | 491 |

|                                                                                           |     |
|-------------------------------------------------------------------------------------------|-----|
| isprint - test for printing character                                                     | 492 |
| ispunct - test for punctuation character                                                  | 493 |
| isspace - test for white-space character                                                  | 494 |
| isupper - test for uppercase letter                                                       | 495 |
| iswalnum - test for alphanumeric wide character                                           | 496 |
| iswalpha - test for alphabetic wide character                                             | 497 |
| iswcntrl - test for control wide character                                                | 498 |
| iswctype - test wide character for class                                                  | 499 |
| iswdigit - test for decimal digit wide character                                          | 500 |
| iswgraph - test for visible wide character                                                | 501 |
| iswlower - test for lowercase wide character                                              | 502 |
| iswprint - test for printing wide character                                               | 503 |
| iswpunct - test for punctuation wide character                                            | 504 |
| iswspace - test for white-space wide character                                            | 505 |
| iswupper - test for uppercase wide character                                              | 506 |
| iswxdigit - test for hexadecimal digit wide character                                     | 507 |
| isxdigit - test for hexadecimal digit                                                     | 508 |
| j0, j1, jn - Bessel functions of first kind                                               | 509 |
| rand48 - generate pseudo-random numbers between -231 and 231 with<br>initialization value | 509 |
| kill - send signal to process or process group                                            | 510 |
| killpg - send signal to process group                                                     | 512 |
| l64a - convert 32-bit integer number to string                                            | 513 |
| labs - return long integer absolute value                                                 | 513 |
| lchown - change owner/group of file                                                       | 514 |
| lcong48 - pseudo-random number (signed long int) generator                                | 516 |
| ldexp - load exponent of floating-point number                                            | 516 |
| ldiv - long division of integers                                                          | 517 |
| lfind - find entry in linear search table                                                 | 517 |
| lgamma - compute logarithm of gamma function                                              | 518 |
| __LINE__ - macro for current source program line number                                   | 518 |
| link - create link to file                                                                | 519 |
| llabs - return absolute value of an integer (long long int)                               | 521 |
| lldiv - division of integers (long long int)                                              | 522 |
| llrint, llrintf, llrintl - round to nearest integer value (long long int)                 | 523 |
| llround, llroundf, llroundl - round up to next integer value (long long int)              | 524 |
| loc1, loc2 - pointers to characters matched by regular expressions                        | 525 |
| localeconv - change components of locale                                                  | 526 |
| localtime - convert date and time to local time                                           | 530 |
| localtime_r - convert date and time to string (thread-safe)                               | 531 |

---

|                                                                           |     |
|---------------------------------------------------------------------------|-----|
| lockf - lock file section                                                 | 532 |
| locs - stop regular expression matching in string                         | 535 |
| log - natural logarithm function                                          | 536 |
| log10 - base 10 logarithm function                                        | 536 |
| log1p - compute natural log                                               | 537 |
| logb - get exponent part of floating-point number                         | 537 |
| _longjmp, _setjmp - non-local jump (without signal mask)                  | 538 |
| longjmp - execute non-local jump                                          | 539 |
| lrand48 - generate pseudo-random numbers between 0 and 231                | 540 |
| lrint, lrintf, lrintl - round to nearest integer value (long int)         | 541 |
| lround, lroundf, lroundl - round up to next integer value (long int)      | 542 |
| lsearch, lfind - linear search and update                                 | 543 |
| lseek - move read/write file offset                                       | 544 |
| lstat - query file status                                                 | 548 |
| major - get major component of device number ( <i>extension</i> )         | 550 |
| makecontext, swapcontext - set up user context                            | 551 |
| makedev - get formatted device number ( <i>extension</i> )                | 552 |
| malloc - memory allocator                                                 | 553 |
| mblen - get number of bytes in multi-byte character                       | 554 |
| mbrlen - get number of bytes in multi-byte character                      | 554 |
| mbrtowc - complete and convert multi-byte string to wide-character string | 555 |
| mbsinit - test for “initial conversion” state                             | 556 |
| mbsrtowcs - convert multi-byte string to wide-character string            | 557 |
| mbstowcs - convert multi-byte string to wide-character string             | 558 |
| mbtowc - convert multi-byte character to wide character                   | 559 |
| memalloc - memory allocator ( <i>BS2000</i> )                             | 560 |
| memccpy - copy bytes in memory                                            | 561 |
| memchr - find byte in memory                                              | 562 |
| memcmp - compare bytes in memory                                          | 563 |
| memcpy - copy bytes in memory                                             | 564 |
| memfree - free memory area ( <i>BS2000</i> )                              | 565 |
| memmove - copy bytes in memory with overlapping areas                     | 566 |
| memset - initialize memory area                                           | 567 |
| minor - get minor component of device number ( <i>extension</i> )         | 568 |
| mkdir - make directory                                                    | 569 |
| mkfifo - create FIFO file                                                 | 571 |
| mknod - make directory, special file, or text file                        | 573 |
| mkstemp - make unique temporary file name                                 | 576 |
| mktemp - make unique temporary file name ( <i>extension</i> )             | 577 |
| mktime - convert local time into time since the Epoch                     | 579 |

|                                                                                        |            |
|----------------------------------------------------------------------------------------|------------|
| mmap - map memory pages                                                                | 581        |
| modf - split floating-point number into integral and fractional parts                  | 585        |
| mount - mount file system ( <i>extension</i> )                                         | 586        |
| mprotect - modify access protection for memory mapping                                 | 588        |
| rand48 - generate pseudo-random numbers between -231 and 231                           | 589        |
| msgctl - message control operations                                                    | 590        |
| msgget - get message queue                                                             | 592        |
| msgrcv - receive message from queue                                                    | 594        |
| msgsnd - send message to queue                                                         | 596        |
| msync - synchronize memory                                                             | 598        |
| munmap - unmap memory pages                                                            | 600        |
| <b>Functions and variables in alphabetical order (n - y)</b>                           | <b>601</b> |
| nanosleep - suspend current thread                                                     | 601        |
| nextafter - next displayable floating-point number                                     | 602        |
| nftw - traverse file tree                                                              | 603        |
| nice - change priority of process                                                      | 606        |
| nl_langinfo - get locale values                                                        | 607        |
| nrnd48 - generate pseudo-random numbers between 0 and 231 with initialization value    | 607        |
| offsetof - get offset of structure component from start of structure ( <i>BS2000</i> ) | 608        |
| open - open file                                                                       | 609        |
| opendir - open directory                                                               | 615        |
| openlog - system logging                                                               | 616        |
| optarg, opterr, optind, optopt - variables for command options                         | 616        |
| pathconf, fpathconf - get value of pathname variable                                   | 617        |
| pause - suspend process until signal is received                                       | 620        |
| pclose - close pipe stream                                                             | 621        |
| perror - write error messages to standard error                                        | 622        |
| pipe - create pipe                                                                     | 623        |
| poll - multiplex STREAMs I/O                                                           | 624        |
| popen - initiate pipe stream to or from process                                        | 627        |
| pow - power function                                                                   | 628        |
| printf - write formatted output on standard output stream                              | 629        |
| ptsname - name of pseudoterminal                                                       | 629        |
| putc, putc_unlocked - put byte on stream                                               | 630        |
| putchar - put byte on standard output stream (thread-safe)                             | 631        |
| putchar_unlocked - put byte on standard output stream (thread-safe)                    | 631        |
| putenv - change or add environment variables                                           | 632        |
| putmsg, putpmsg - send message to STREAMs file                                         | 633        |
| putpwent - enter user into user catalog ( <i>extension</i> )                           | 636        |

|                                                                                          |     |
|------------------------------------------------------------------------------------------|-----|
| puts - put string on standard output                                                     | 637 |
| pututxline - write utmpx entry                                                           | 638 |
| putw - put word on stream                                                                | 639 |
| putwc - put wide character on stream                                                     | 640 |
| putwchar - put wide character on standard output stream                                  | 641 |
| qsort - sort table of data                                                               | 642 |
| raise - send signal to calling process                                                   | 643 |
| rand - pseudo-random number generator (int)                                              | 645 |
| rand_r - pseudo-random number generator (int, thread-safe)                               | 645 |
| random - create pseudo-random numbers                                                    | 646 |
| read - read bytes from file                                                              | 647 |
| readdir - read directory                                                                 | 650 |
| readdir_r - read directory (thread-safe)                                                 | 652 |
| readlink - read contents of symbolic link                                                | 653 |
| readv - read array from file                                                             | 654 |
| realloc - memory reallocator                                                             | 656 |
| realpath - output real file name/pathname                                                | 657 |
| re_comp, re_exec - compile and execute regular expressions                               | 658 |
| regcmp, regex - compile and execute regular expression                                   | 661 |
| regexp: advance, compile, step, loc1, loc2, locs - compile and match regular expressions | 664 |
| remainder - remainder from division                                                      | 671 |
| remove - remove files                                                                    | 672 |
| remque - remove element from queue                                                       | 673 |
| rename - rename file                                                                     | 674 |
| rewind - reset file position indicator to start of stream                                | 677 |
| rewinddir - reset file position indicator to start of directory stream                   | 678 |
| rindex - get last occurrence of character in string                                      | 679 |
| rint, rintf, rintl - round to nearest integer value                                      | 680 |
| rmdir - remove directory                                                                 | 681 |
| round, roundf, roundl - round up to next integer value                                   | 683 |
| sbrk - modify size of data segment                                                       | 684 |
| scalb - load exponent of base-independent floating-point number                          | 684 |
| scanf - read formatted input from standard input stream                                  | 685 |
| seed48 - set seed (int) for pseudo-random numbers                                        | 685 |
| seekdir - set position of directory stream                                               | 686 |
| select - synchronous I/O multiplexing                                                    | 687 |
| semctl - semaphore control operations                                                    | 689 |
| semget - get semaphore ID                                                                | 692 |
| semop - semaphore operations                                                             | 694 |

|                                                                              |     |
|------------------------------------------------------------------------------|-----|
| setbuf - assign buffering to stream                                          | 698 |
| setcontext - modify user context                                             | 699 |
| setgid - set group ID of process                                             | 700 |
| setgrent - reset file position indicator to beginning of group file          | 701 |
| setitimer - set interval timer                                               | 701 |
| _setjmp - set label for non-local jump (without signal mask)                 | 701 |
| setjmp - set label for non-local jump                                        | 702 |
| setkey - set encoding key                                                    | 703 |
| setlocale - set or query locale                                              | 704 |
| setlogmask - set log priority mask                                           | 706 |
| setpgid - set process group ID for job control                               | 707 |
| setpgrp - set process group ID                                               | 708 |
| setpriority - set process priority                                           | 708 |
| setpwnent - delete pointer to search user catalog                            | 708 |
| setregid - set real and effective group IDs                                  | 709 |
| setreuid - set real and effective user IDs                                   | 710 |
| setrlimit - set resource limit                                               | 711 |
| setsid - create session and set process group ID                             | 712 |
| setstate - pseudo-random numbers                                             | 713 |
| setuid - set user ID                                                         | 713 |
| setutxent - reset pointer to utmpx file                                      | 714 |
| setvbuf - assign buffering to stream                                         | 714 |
| shmat - shared memory attach operation                                       | 716 |
| shmctl - shared memory control operations                                    | 718 |
| shmdt - shared memory detach operation                                       | 720 |
| shmget - create shared memory segment                                        | 721 |
| sigaction - examine and change signal handling                               | 723 |
| sigaddset - add signal to signal set                                         | 731 |
| sigaltstack - set/read alternative stack of signal                           | 732 |
| sigdelset - delete signal from signal set                                    | 734 |
| sigemptyset - initialize and empty signal set                                | 735 |
| sigfillset - initialize and fill signal set                                  | 736 |
| sighold, sigignore - add signal to signal mask / register SIG_IGN for signal | 736 |
| siginterrupt - change behavior of system calls in response to interrupts     | 737 |
| sigismember - test for member of signal set                                  | 738 |
| siglongjmp - execute non-local jump using signal                             | 739 |
| signal - examine or change signal handling                                   | 740 |
| signgam - variable for sign of lgamma                                        | 743 |
| sigpause - remove signal from signal mask and deactivate process             | 743 |
| sigpending - examine pending signals                                         | 743 |

|                                                                    |     |
|--------------------------------------------------------------------|-----|
| sigprocmask - examine or change blocked signals                    | 744 |
| sigrelse - remove signal from signal mask                          | 746 |
| sigset - modify signal handling                                    | 746 |
| sigsetjmp - set label for non-local jump using signal              | 747 |
| sigstack - set or query alternative stack for signal               | 749 |
| sigsuspend - wait for signal                                       | 750 |
| sin - sine function                                                | 751 |
| sinh - hyperbolic sine function                                    | 751 |
| sleep - suspend process for fixed interval of time                 | 752 |
| sprintf - write formatted output to string                         | 754 |
| sqrt - square root function                                        | 754 |
| srand - generate pseudo-random numbers with seed                   | 754 |
| srandom - pseudo-random numbers                                    | 755 |
| srand48 - seed (double-precision) pseudo-random number generator   | 755 |
| sscanf - read formatted input from string                          | 755 |
| stat - get file status                                             | 756 |
| statvfs - read file system information                             | 760 |
| __STDC__ - macro for ANSI conformance                              | 760 |
| __STDC_VERSION__ - Amendment 1 conformity?                         | 760 |
| stderr, stdin, stdout - variables for standard I/O streams         | 761 |
| step - compare regular expressions                                 | 762 |
| strcasemp, strncasemp - non-case-sensitive string comparison       | 762 |
| strcat - concatenate two strings                                   | 763 |
| strchr - scan string for characters                                | 763 |
| strcmp - compare two strings                                       | 764 |
| strcoll - compare strings using collating sequence                 | 765 |
| strcpy - copy string                                               | 766 |
| strcspn - get length of complementary substring                    | 766 |
| strdup - duplicate string                                          | 767 |
| strerror - get message string                                      | 768 |
| strfill - copy substring ( <i>BS2000</i> )                         | 769 |
| strfmon - convert monetary value to string                         | 770 |
| strftime - convert date and time to string                         | 774 |
| strlen - get length of string                                      | 777 |
| strlower - convert a string to lowercase letters ( <i>BS2000</i> ) | 777 |
| strncasemp - non-case-sensitive string comparisons                 | 778 |
| strncat - concatenate two substrings                               | 778 |
| strncmp - compare two substrings                                   | 779 |
| strncpy - copy substring                                           | 780 |
| strpbrk - get first occurrence of character in string              | 781 |

|                                                                       |     |
|-----------------------------------------------------------------------|-----|
| strptime - convert string to date and time                            | 782 |
| strrchr - get last occurrence of character in string                  | 785 |
| strspn - get length of substring                                      | 786 |
| strstr - find substring in string                                     | 786 |
| strtod - convert string to double-precision number                    | 787 |
| strtok - split string into tokens                                     | 788 |
| strtok_r - split string into tokens (thread-safe)                     | 789 |
| strtol - convert string to long integer                               | 790 |
| strtoll - convert string to long long integer                         | 792 |
| strtoul - convert string to unsigned long integer                     | 794 |
| strtoull - convert string to unsigned long long                       | 796 |
| strupper - convert string to uppercase letters ( <i>BS2000</i> )      | 798 |
| strxfrm - string transformation based on LC_COLLATE                   | 799 |
| swab - swap bytes                                                     | 800 |
| swapcontext - swap user context                                       | 800 |
| swprintf - output formatted wide characters                           | 801 |
| swscanf - formatted read                                              | 801 |
| symlink - make symbolic link to file                                  | 802 |
| sync - update superblock                                              | 804 |
| sysconf - get numeric value of configurable system variable           | 805 |
| sysfs - get information on file system type ( <i>extension</i> )      | 809 |
| syslog - log message                                                  | 810 |
| system - execute system command                                       | 811 |
| tan - compute tangent                                                 | 814 |
| tanh - compute hyperbolic tangent                                     | 814 |
| tcdrain - wait for transmission of output                             | 815 |
| tcflow - suspend or restart data transmission                         | 816 |
| tcflush - discard non-transmitted data                                | 817 |
| tcgetattr - get parameters associated with terminal                   | 818 |
| tcgetpgrp - get foreground process group ID                           | 819 |
| tcgetsid - get session ID of specified terminal                       | 820 |
| tcsendbreak - interrupt serial data transmission                      | 821 |
| tcsetattr - set parameters associated with terminal                   | 822 |
| tcsetpgrp - set foreground process group ID                           | 824 |
| tdelete - delete node from binary search tree                         | 824 |
| tell - get current value of file position indicator ( <i>BS2000</i> ) | 825 |
| telldir - get current location of named directory stream              | 826 |
| tempnam - create pathname for temporary file                          | 827 |
| tfind - find node in binary search tree                               | 829 |
| __TIME__ - macro for compilation time                                 | 829 |



---

|                                                                          |     |
|--------------------------------------------------------------------------|-----|
| time - get time since the Epoch                                          | 830 |
| times - get process times                                                | 831 |
| timezone - variable for difference between local time and UTC            | 832 |
| tmpfile - create temporary file                                          | 833 |
| tmpnam - create base name for temporary file                             | 834 |
| toascii - convert integer to legal value                                 | 835 |
| toebcdic - convert integer to legal value ( <i>BS2000</i> )              | 836 |
| _tolower - convert uppercase letters to lowercase                        | 836 |
| tolower - convert characters to lowercase                                | 837 |
| _toupper - convert lowercase letters to uppercase                        | 837 |
| toupper - convert characters to uppercase                                | 838 |
| towctrans - map wide characters                                          | 838 |
| tolower - convert wide characters to lowercase                           | 839 |
| towupper - convert wide characters to uppercase                          | 839 |
| truncate - set file to specified length                                  | 840 |
| tsearch, tfind, tdelete, twalk - process binary search trees             | 841 |
| ttyname - find pathname of terminal                                      | 843 |
| ttyname_r - find pathname of terminal (thread-safe)                      | 844 |
| ttyslot - find entry of current user in utmp file                        | 845 |
| twalk - traverse binary search tree                                      | 846 |
| tzname - array variable for timezone strings                             | 846 |
| tzset - set timezone conversion information                              | 847 |
| ualarm - set interval timer                                              | 848 |
| ulimit - get and set process limits                                      | 849 |
| umask - get and set file mode creation mask                              | 850 |
| umount - unmount file system ( <i>extension</i> )                        | 851 |
| uname - get basic data on current operating system                       | 852 |
| ungetc - push byte back onto input stream                                | 853 |
| ungetwc - push wide character back onto input stream                     | 855 |
| unlink - remove link                                                     | 856 |
| unlockpt - remove lock from master/slave pseudoterminal pair             | 858 |
| usleep - suspend process for defined interval                            | 859 |
| utime - set file access and modification times                           | 860 |
| utimes - set file access time and file modification time                 | 862 |
| va_arg - process variable argument list                                  | 864 |
| va_end - end variable argument list                                      | 865 |
| va_start - initialize variable argument list                             | 866 |
| valloc - request memory aligned with page boundary                       | 867 |
| vfork - generate new process in virtual memory                           | 868 |
| vfprintf, vprintf, vsprintf - formatted output of variable argument list | 869 |

|                                                                           |     |
|---------------------------------------------------------------------------|-----|
| vwprintf - formatted output of wide characters                            | 870 |
| vprintf - formatted output to standard out                                | 870 |
| vsprintf - formatted output to a string                                   | 871 |
| vswprintf - formatted output of wide characters                           | 872 |
| vwprintf - formatted output of wide characters                            | 872 |
| wait, waitpid - wait for child process to stop or terminate               | 873 |
| wait3 - wait for status change of child processes                         | 877 |
| waitid - wait for status change of child processes                        | 878 |
| wcrtomb - convert wide characters to multi-byte characters                | 880 |
| wscat - concatenate two wide character strings                            | 881 |
| wcschr - scan wide character string for wide characters                   | 882 |
| wscmp - compare two wide character strings                                | 883 |
| wscoll - compare two wide character strings according to LC_COLLATE       | 884 |
| wscopy - copy wide character string                                       | 885 |
| wcscspn - get length of complementary wide character substring            | 886 |
| wcsftime - convert date and time to wide character string                 | 887 |
| wcslen - get length of wide character string                              | 888 |
| wcsncat - concatenate two wide character strings                          | 889 |
| wcsncmp - compare two wide character substrings                           | 890 |
| wcsncpy - copy wide character substring                                   | 891 |
| wcspbrk - get first occurrence of wide character in wide character string | 892 |
| wcsrchr - get last occurrence of wide character in wide character string  | 893 |
| wcsrtombs - convert wide character string to multi-byte string            | 894 |
| wcssp - get length of wide character substring                            | 895 |
| wcsstr - search for first occurrence of a wide character string           | 896 |
| wctod - convert wide character string to double-precision number          | 897 |
| wctok - split wide character string into tokens                           | 899 |
| wctol - convert wide character string to long integer                     | 900 |
| wctoll - convert wide character string to long long integer               | 902 |
| wcstombs - convert wide character string to character string              | 904 |
| wcstoul - convert wide character string to unsigned long                  | 905 |
| wcstoull - convert wide character string to unsigned long long            | 907 |
| wcswcs - find wide character substring in wide character string           | 909 |
| wcswidth - get number of column positions of wide character string        | 910 |
| wcsxfrm - transform wide character string                                 | 911 |
| wctob - convert wide character to 1-byte multi-byte character             | 912 |
| wctomb - convert wide character code to character                         | 912 |
| wctrans - define wide character mappings                                  | 913 |
| wctype - define wide character class                                      | 914 |
| wcwidth - get number of column positions of wide character code           | 915 |

|                                                                       |            |
|-----------------------------------------------------------------------|------------|
| wmemchr - search for wide character in a wide character string        | 916        |
| wmemcmp - compare two wide character strings                          | 917        |
| wmemcpy - copy wide character string                                  | 917        |
| wmemmove - copy wide character string in overlapping area             | 918        |
| wmemset - set first <i>n</i> wide characters in wide character string | 918        |
| wprintf - formatted output of wide characters                         | 919        |
| write - write bytes to file                                           | 920        |
| writev - write to file                                                | 926        |
| wscanf - formatted read                                               | 927        |
| y0, y1, yn - Bessel functions of the second kind                      | 927        |
| <b>Header files</b>                                                   | <b>929</b> |
| assert.h - verify program assertion                                   | 930        |
| cpio.h - cpio archive mode values                                     | 931        |
| ctype.h - character classification                                    | 932        |
| dirent.h - format of directory entries                                | 933        |
| errno.h - system error codes                                          | 934        |
| fcntl.h - file control options                                        | 944        |
| float.h - types for floating-point numbers                            | 947        |
| fmtmsg.h - structure of message display                               | 950        |
| ftw.h - file tree traversal                                           | 951        |
| grp.h - group structure                                               | 952        |
| iconv.h - codeset conversion facility                                 | 953        |
| iso646.h - alternative ways of writing operators                      | 953        |
| langinfo.h - language information constants                           | 954        |
| libgen.h - function definitions for pattern matching                  | 956        |
| limits.h - implementation-dependent constants                         | 957        |
| locale.h - category macros                                            | 965        |
| math.h - mathematical functions and constants                         | 967        |
| monetary.h - types for monetary values                                | 970        |
| ndbm.h - definitions of operations for ndbm database                  | 970        |
| nl_types.h - native language data types                               | 972        |
| poll.h - definitions for poll() function                              | 973        |
| pwd.h - password structure                                            | 974        |
| re_comp.h - comparison of regular expressions with re_comp            | 975        |
| regex.h - declarations for regular expressions                        | 975        |
| search.h - search tables                                              | 976        |
| setjmp.h - stack environment declarations                             | 977        |
| signal.h - signals                                                    | 978        |
| stdarg.h - edit variable argument list                                | 982        |
| stddef.h - standard type definitions                                  | 984        |

|                                                                |             |
|----------------------------------------------------------------|-------------|
| stdio.h - standard buffered input/output                       | 985         |
| stdlib.h - standard library definitions                        | 990         |
| string.h - string operations                                   | 993         |
| strings.h - string operations                                  | 994         |
| stropts.h - STREAMS interface                                  | 995         |
| sys/ipc.h - interprocess communication access structure        | 1000        |
| syslog.h - definitions for logging system error messages       | 1001        |
| sys/mman.h - definitions for memory management                 | 1003        |
| sys/msg.h - message queue structures                           | 1004        |
| sys/resource.h - definitions of operations for XSI resources   | 1005        |
| sys/sem.h - semaphore structures                               | 1008        |
| sys/shm.h - shared memory definitions                          | 1010        |
| sys/stat.h - data for file status                              | 1011        |
| sys/statvfs.h - structure of VFS file system information       | 1014        |
| sys/time.h - time types                                        | 1016        |
| sys/timeb.h - additional definitions for date and time         | 1018        |
| sys/times.h - structure for file times                         | 1018        |
| sys/types.h - data types                                       | 1019        |
| sys/uio.h - definitions for array I/O operations               | 1021        |
| sys/utsname.h - system name structure                          | 1021        |
| sys/wait.h - declarations for waiting                          | 1022        |
| tar.h - extended tar definitions                               | 1023        |
| termios.h - define values for termios                          | 1025        |
| time.h - time types                                            | 1031        |
| ucontext.h - user context                                      | 1033        |
| ulimit.h - commands for ulimit                                 | 1034        |
| unistd.h - standard symbolic constants and structures          | 1035        |
| utime.h - manipulate time structures                           | 1042        |
| utmpx.h - entry format                                         | 1043        |
| varargs.h - handle variable argument list                      | 1044        |
| wchar.h - types for wide character values                      | 1045        |
| wctype.h - wide character classification and mapping utilities | 1047        |
| <b>Appendix: KR or ANSI functionality</b>                      | <b>1049</b> |
| <b>Glossary</b>                                                | <b>1053</b> |
| <b>Related publications</b>                                    | <b>1085</b> |
| <b>Index</b>                                                   | <b>1091</b> |

---

# **C Library Functions V2.5A for POSIX Applications (BS2000/OSD)**

## **Reference Manual**

### *Target group*

This manual addresses C and C++ programmers.

### *Contents*

The manual documents the XPG4-conformant C programming interface which is supported by the POSIX subsystem in BS2000. This programming interface permits access to both the POSIX file system and BS2000 files. The programming interface also incorporates extensions which ensure compatibility with the existing C library.

**Edition: August 2001**

**File: c\_plib.pdf**

Copyright © Fujitsu Siemens Computers GmbH, 2001.

All rights reserved.

Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers..

Fujitsu Siemens computers GmbH  
User Documentation  
81730 Munich  
Germany

Comments  
Suggestions  
Corrections

**Fax: (++49) 700 / 372 00000**

e-mail: [manuals@fujitsu-siemens.com](mailto:manuals@fujitsu-siemens.com)  
<http://manuals.fujitsu-siemens.com>

---

Submitted by

---

Comments on C Library Functions V2.5A  
for POSIX Applications



## Information on this document

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.

This document from the document archive refers to a product version which was released a considerable time ago or which is no longer marketed.

Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.

Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format ...@[ts.fujitsu.com](mailto:ts.fujitsu.com).

The Internet pages of Fujitsu Technology Solutions are available at [http://ts.fujitsu.com/...](http://ts.fujitsu.com/)

and the user documentation at <http://manuals.ts.fujitsu.com>.

Copyright Fujitsu Technology Solutions, 2009

## Hinweise zum vorliegenden Dokument

Zum 1. April 2009 ist Fujitsu Siemens Computers in den alleinigen Besitz von Fujitsu übergegangen. Diese neue Tochtergesellschaft von Fujitsu trägt seitdem den Namen Fujitsu Technology Solutions.

Das vorliegende Dokument aus dem Dokumentenarchiv bezieht sich auf eine bereits vor längerer Zeit freigegebene oder nicht mehr im Vertrieb befindliche Produktversion.

Bitte beachten Sie, dass alle Firmenbezüge und Copyrights im vorliegenden Dokument rechtlich auf Fujitsu Technology Solutions übergegangen sind.

Kontakt- und Supportadressen werden nun von Fujitsu Technology Solutions angeboten und haben die Form ...@[ts.fujitsu.com](mailto:ts.fujitsu.com).

Die Internetseiten von Fujitsu Technology Solutions finden Sie unter [http://de.ts.fujitsu.com/...](http://de.ts.fujitsu.com/), und unter <http://manuals.ts.fujitsu.com> finden Sie die Benutzerdokumentation.

Copyright Fujitsu Technology Solutions, 2009