English

FUJITSU Software BS2000

CRTE

C Library Functions for POSIX Applications

Reference Manual

Valid for:
CRTE V10.0B00/V11.0B00

Edition June 2018

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:
manuals@ts.fujitsu.com

Documentation creation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright and Trademarks

Contents

1 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) which, in turn, is a component of the Common Runtime Environment **CRTE**. The POSIX subsystem must be loaded in order to obtain the full functionality of the C library functions described in this manual.

The interfaces described in this manual are available in CRTE V10.0B or CRTE V11.0B and higher and in BS2000/OSD-BC V10.0 and higher.

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) (see also the "manual "C Library Functions" [6]) as well as numerous other extensions which are supported on many UNIX systems. 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 (DMS) as well as the XPG4 Version 2 conformant POSIX file system can be accessed (see the manual "POSIX Basics" [1]).

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 49 and the manuals "C Compiler" [3] and "C/C++ Compiler" [4]).

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

1.1 Objectives and target groups of this manual

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

– port C programs from UNIX platforms to the POSIX subsystem

– write C programs for XPG4 Version 2 conformant environments (POSIX subsystem) under BS2000

– 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 operating systems are prerequisite to working with this manual.

1.2 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.

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 (manuals „C Compiler“ [3] und „C/C++ Compiler“ [4]) 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" User Guide [7].

Readme files

The functional changes to the current product version and revisions to this manual are described in the product-specific Readme file.

Readme files are available to you online in addition to the product manuals under the various products at *http://manuals.ts.fujitsu.com*. You will also find the Readme files on the Softbook DVD.

Information under BS2000

When a Readme file exists for a product version, you will find the following file on the BS2000 system:

`SYSRME.<product>.<version>.<lang>`

This file contains brief information on the Readme file in English or German (<lang>=E/D). You can view this information on screen using the `/SHOW-FILE` command or an editor. The `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` command shows the user ID under which the product's files are stored.

Additional product information

Current information, version and hardware dependencies, and instructions for installing and using a product version are contained in the associated Release Notice. These Release Notices are available online at *http://manuals.ts.fujitsu.com*.

1.3 Organization of the POSIX documentation

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

- An introduction to working with the POSIX subsystem is presented in the manual "POSIX - Basics for Users and System Administrators" [1]. 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 [5] 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 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, for example, can also process files of the POSIX file system.

Due to the CRTE (Common Runtime Environment) extensions based on the XPG4 standard, it is possible to write portable C programs using the C library functions 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.

1.4 Changes since the last edition of the manual

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

- New functions of the so-called epoll group. These are a scalable I/O event notification mechanism and thus an alternative to the present POSIX functions `select()` and `poll()`.

  `epoll_create`
  : Create an `epoll` instance

  `epoll_ctl`
  : Create an `epoll` instance

  `epoll_wait`
  : Wait for events (`epoll` instance)

- Further new functions

  `dirfd`
  : Extract a file descriptor from a `DIR` object.

  `clock_gettime`, `clock_gettime64`
  : Functions to determine the time of a specfied clock. Only the system-wide real time clock CLOCK_REALTIME is supported.

- Changed functions

  `strptime`
  : The restriction, that requires that there must be a white-space character or a non-alphanumeric character between two conversion specifications, is omitted.

1.5 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. `HUGE_VAL`), symbolic names of signals (e.g. `SIGABRT`) and error codes (e.g. `EDOM`) that are not implementation-dependent.
`{UPPERCASE}`	Indicates implementation-dependent symbolic constants that are defined in the header file `limits.h` (e.g. `{INT_MAX}`).
italics	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.
[Key]	Used to represent keys.
[Key1] + [Key2]	Indicates keys that must be pressed simultaneously.
[Key1] [Key2]	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" on page 193.

2 The C programming interface

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

2.1 System requirements

The table below lists the software products which are necessary to support the complete functionality of the C library as they are provided with CRTE V10.0B00/V11.0B00 and described in this manual.

Product	Relevant components
BS2000/OSD-BC as of V10.0 or V11.0	– Operating system – Header files for POSIX functions
C/C++ as of V3.2	C and C++ compiler for POSIX subsystem and BS2000
CRTE V10.0B00/ V11.0B00	– Header files for BS2000 functions – Runtime modules of the C library functions
POSIX-BC as of V10.0 or V11.0	– POSIX file system – Basic shell POSIX-HEADER V10.0B00/ V11.0B00
SDF-P	Variable structure `SYSPOSIX` 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" [2]. 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.

2.2 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 51 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").

2.2.1 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 (see table on page 31).

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

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 UNIX 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:

and	&&	compl	~	or_eq	|=
and_eq	&=	not	!	xor	^
bitand	&	not_eq	!=	xor_eq	^=
bitor	|	or	| |		

2.2.2 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 manuals "C Compiler" [3] and "C/C++ Compiler" [4]).

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.

2.2.3 Support for DMS and UFS files > 2 GB

For processing file systems that contain files > 2 gigabytes (GB) a 64-bit variant exists for each of the following 32-bit C Library functions. The 64-bit functions differ from the corresponding 32-bit functions in that they have the suffix “64” in their names.

creat:	creat64
fgetpos:	fgetpos64
fopen:	fopen64
freopen:	freopen64
fseek:	fseek64
fseeko:	fseeko64
fsetpos:	fsetpos64
ftell:	ftell64
ftello:	ftello64
lseek:	lseek64
open:	open64
tmpfile:	tmpfile64

32-bit and 64-bit C/C++ library functions

There is no difference in terms of functionality between the 32-bit variant of a function and the associated 64-bit variant. The only differences concern the data types for parameters and return values if these specify an offset or a file position, since offset and return values > 2 GB must possible in order to process files > 2 GB. Thus, in addition to the 32-bit data type `off_t`, for example, there is also a 64-bit data type called `off64_t`.

The compilation environment makes available all the explicit 64-bit functions and types in addition to the 32-bit functions and types. A program can thus use either interface, as required.

i
- The 64-bit functions are only available with ANSI functionality.
- Since most of the names of the 64-bit functions are no longer unique CRTE-wide when truncated to 8 characters, sources that want to use 64-bit functions have to be generated as LLMs.

Using the 64-bit interface

The `_FILE_OFFSET_BITS` define allows you to choose between two alternatives for using the 64-bit interface:

- using 64-bit functions transparently (`_FILE_OFFSET_BITS 64`)
- calling 64-bit functions explicitly (`_FILE_OFFSET_BITS 32`)

i
- The `_FILE_OFFSET_BITS` define must be set on an include file before the first include.
- You can replace 32-bit functions with 64-bit functions automatically by means of name defines or macro defines.

Using 64-bit functions transparently (`_FILE_OFFSET_BITS 64`)

The `_FILE_OFFSET_BITS 64` define allows the 64-bit interface to be used transparently, since the 32-bit functions contained in the source code are automatically replaced with the associated 64-bit variants during compilation (with the exception of `fseek` and `ftell`, see below). In addition, the compilation environment makes data types available in the appropriate size. The data type `off_t`, for example, is declared as `long long`.

You can use the `_MAP_NAME` preprocessor define to specify whether the 32-bit functions are to be mapped to 64-bit functions by means of the name define method or the macro define method.

A program can process both files > 2 GB and files ≤ 2 GB. Transparent use of the 64-bit functions permits programs that were previously designed only for files ≤ 2 GB to process files > 2GB without the need for any changes to the source code.

i The functions `fseek` and `ftell` cannot be automatically replaced with `fseek64` and `ftell64`. Please use the functions `fseeko` and `ftello` if you want automatic replacement to be carried out.

Calling 64-bit functions explicitly

If the `_FILE_OFFSET_BITS 32` define is set or if `_FILE_OFFSET_BITS` is not defined, you have to use the 64-bit variants of the file processing functions described above in order to process files > 2 GB:

– If you try to process a file > 2 GB using a 32-bit variant, this leads to abortion.

– If you use the 64-bit variants, however, you can also process files ≤ 2 GB.

i You can only use the 64-bit functions explicitly if the `_LARGEFILE64_SOURCE 1` define is set beforehand (prototype generation and further defines).

2.2.4 POSIX thread support in the C runtime library

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

2.2.5 IEEE floating-point arithmetic

The IEEE floating-point arithmetic is supported as follows:

– The C/C++ compiler offers a compiler option with which floating-point numbers can be generated in IEEE format (see page 38).

– For every library function in the C runtime system that works with or returns floating-point numbers, there is a variant for processing IEEE floating-point numbers and a macro define that maps the standard variant (/390 variant) of the function to the associated IEEE variant (see page 39).

For each compiler option you can activate all the IEEE functionality: the C/C++ compiler then generates floating-point numbers in IEEE format in all modules and automatically provides the appropriate IEEE functions for processing the IEEE floating-point numbers.

In addition, you can use the IEEE functionality provided in a modified form:

– You can use the `_IEEE_SOURCE` preprocessor define to specify whether the library functions for /390 floating-point arithmetic are mapped to the associated IEEE variants (see page 40).

– You can use conversion functions to convert floating-point numbers explicitly from /390 format to IEEE format (see page 41).

Notes on the use of IEEE floating point arithmetic

The following points must be noted when using IEEE floating point arithmetic:

– IEEE floating point operations differ semantically from the corresponding /390 floating point operations, e.g. in rounding. In IEEE format, "Round to nearest" is used by default whereas "Round to zero" is used in /390 format.

– In error and exception cases (e.g. argument outside permitted value range) the reactions of IEEE functions differ from those of /390 functions, e.g. some functions return the value `NaN`.

– You must include the relevant include file for each C library function in your program that uses floating point numbers. Otherwise, these functions cannot process the floating point numbers correctly. You must, in particular, include the `<stdio.h>` include file with `#include <stdio.h>` for the printf function.

2.2.5.1 Generating IEEE floating-point numbers by means of a compiler option

For floating-point numbers the C/C++ compiler generates code in /390 format or IEEE format, as required. You specify the format you want by means of the FP-ARITHMETICS clause of the MODIFY-MODULE-PROPERTIES compiler option.

```
MODIFY-MODULE-PROPERTIES                 -
...
                   ⎧*390-FORMAT⎫
FP-ARITHMETICS=    ⎨           ⎬ ,       -
                   ⎩*IEEE-     ⎭
LOWER-CASE-NAMES=*YES,                   -
SPECIAL-CHARACTERS=*KEEP,                -
...
```

FP-ARITHMETICS=*390-FORMAT
: The compiler generates code for constants and arithmetic operations in /390 format. *390-FORMAT is the default.

FP-ARITHMETICS=*IEEE-FORMAT
: The compiler generates code for constants and arithmetic operations in IEEE format. In addition, the `_IEEE` preprocessor define is set to 1. Unless the `_IEEE_SOURCE` preprocessor define is set to 0 (see page 40), the original /390 library functions are automatically mapped to the associated IEEE functions.

LOWER-CASE-NAMES=*YES
SPECIAL-CHARACTERS=*KEEP
: By specifying these, you prevent:

  – the names of the IEEE functions (see page 39) from being truncated to eight characters

  – lowercase letters from being converted to uppercase and the character "_" from being replaced by "$" in the function names

In POSIX you specify the IEEE format by means of the following option:

```
-K ieee_floats
```

To ensure the IEEE function names are processed correctly, you specify:

```
-K llm_keep
-K llm_case_lower
```

2.2.5.2 C library functions that support IEEE floating-point numbers

For every function that works with floating-point numbers or returns a floating-point number, the C runtime system offers:

- an implementation of the function with /390 arithmetic
- an implementation of the function with IEEE arithmetic
- a macro define that maps the original function (/390 function) to the associated IEEE function

The prototype of an IEEE function and the associated define are stored in the include file in which the corresponding original function is declared. This has the advantage that no additional include files are required in order to use the IEEE floating-point arithmetic, with the possible exception of <ieee_390.h> (see page 41).

Names of the IEEE functions

The syntax of the names of the IEEE functions is as follows:

__*originalfunction*_`ieee()`

The name of the original function should be specified for *originalfunction*.

The IEEE variant of `sin()`, for example, is `__sin_ieee()`.

C library functions for which there is an IEEE function

`acos()`	`asin()`	`atan()`	`atan2()`	`atof()`
`ceil()`	`cos()`	`cosh()`	`difftime()`	`difftime64()`
`ecvt()`	`ecvt_r()`	`erf()`	`erfc()`	`exp()`
`fabs()`	`fcvt()`	`fcvt_r()`	`floor()`	`fprintf()`
`frexp()`	`fscanf()`	`gamma()`	`gcvt()`	`hypot()`
`j0()`	`j1()`	`jn()`	`ldexp()`	`llrint()`
`llrintf()`	`llrintl()`	`llround()`	`llroundf()`	`llroundl()`
`log()`	`log10()`	`lrint()`	`lrintf()`	`lrintl()`
`lround()`	`lroundf()`	`lroundl()`	`modf()`	`pow()`
`printf()`	`rint()`	`rintf()`	`rintl()`	`round()`
`roundf()`	`roundl()`	`scanf()`	`sin()`	`sinh()`
`snprintf()`	`sprintf()`	`sqrt()`	`sscanf()`	`strtod()`
`tan()`	`tanh()`	`vfprintf()`	`vprintf()`	`vsnprintf()`
`vsprintf()`	`y0()`	`y1()`	`yn()`	

2.2.5.3 Controlling the mapping of original functions to the associated IEEE variants

You can use the _IEEE_SOURCE preprocessor define to specify whether the original library functions (/390 functions) for floating-point arithmetic are mapped to the associated IEEE variants. The prototypes of the IEEE functions are always generated.

_IEEE_SOURCE can take on the following values:

_IEEE_SOURCE == 0
The /390 functions are not mapped to the corresponding IEEE variants. /390 and IEEE functions can thus be used in parallel. This setting applies regardless of the settings of the compiler (see the _IEEE define on page 38).

_IEEE_SOURCE == 1
The /390 functions are mapped to the corresponding IEEE variants. It is thus not possible to use /390 and IEEE functions in parallel. This setting applies regardless of the settings of the compiler (see the _IEEE define on page 38).

The _MAP_NAME preprocessor define allows you to specify whether the /390 functions are to be mapped to the IEEE functions by means of the name define method or the macro define method.

i If you want to control the mapping of the original functions to the associated IEEE functions by means of the preprocessor define, you have to use the function declarations of the standard include files (i.e. you have to include the standard include files).

_IEEE_SOURCE is not defined
In this case, the following takes place, depending on the compiler option (see the _IEEE define on page 38):

_IEEE == 0 or _IEEE not defined
The /390 functions are not mapped to the corresponding IEEE variants.

_IEEE == 1
The /390 functions are mapped to the corresponding IEEE variants.

i To control the mapping of the original functions to the associated IEEE variants, you have to specify the MODIFY-MODULE-PROPERTIES compiler option as follows:

```
MODIFY-MODULE-PROPERTIES           -
...
LOWER-CASE-NAMES=*YES,             -
SPECIAL-CHARACTERS=*KEEP,          -
...
```

This prevents:

– the names of the IEEE functions (see page 39) from being truncated to eight characters

– lowercase letters from being converted to uppercase and the character “_” from being replaced with “$” in the function names

In POSIX, you specify the following to achieve this:

```
-K llm_keep
-K llm_case_lower
```

2.2.5.4 Explicit conversion of floating-point numbers

In addition to the compiler and runtime system extensions for IEEE support described in the above sections, there are also functions for explicitly converting floating-point numbers between the /390 and IEEE formats.

The following conversion functions are declared in the include file <ieee_390.h>:

```
extern float float2ieee(float num);
extern float ieee2float(float num);

extern double double2ieee(double num);
extern double ieee2double(double num);
```

Conversion functions are described in detail in the chapter “Functions and variables in alphabetical order” on page 193.

2.2.6 ASCII encoding

In addition to the standard EBCDIC encoding of characters and strings, ASCII encoding of characters and strings is also supported:

– The C/C++ compiler offers an option by means of which characters and strings can be generated in ASCII format (see page 42).

– For every library function in the C runtime system that works with characters or strings or that returns a character or a string, there is a variant for processing ASCII characters and strings and a macro define that maps the EBCDIC variant of the function to the associated ASCII variant (see page 45).

For each compiler option you can activate all the ASCII functionality: the C/C++ compiler then generates characters and strings in ASCII format in all modules and automatically provides the appropriate ASCII functions for processing the ASCII characters and strings.

In addition, you can use the ASCII functionality provided in a modified form:

– You can use the `_ASCII_SOURCE` preprocessor define to specify whether the library functions for EBCDIC representation are mapped to the associated ASCII variants (see page 45).

– You can use conversion functions to convert ASCII characters and strings explicitly from EBCDIC format to ASCII format (see page 46).

2.2.6.1 Generating ASCII characters and strings by means of a compiler option

The C/C++ compiler generates code for characters and strings in EBCDIC format (default) or ASCII format, as required. You specify the format you want by means of the LITERAL-ENCODING option of the MODIFY-SOURCE-PROPERTIES .statement.

```
MODIFY-SOURCE-PROPERTIES ..., LITERAL-ENCODING=*NATIVE|*ASCII-FULL
```

LITERAL-ENCODING=*NATIVE
The compiler generates code for characters and strings in EBCDIC format. *NATIVE is the default.

LITERAL-ENCODING=*ASCII-FULL
The compiler generates code for characters and strings in ASCII format. In addition, the `_LITERAL_ENCODING_ASCII` preprocessor define is set to 1. Unless the `_ASCII_SOURCE` preprocessor define is set to 0 (see page 45), the EBCDIC library functions are automatically mapped to the associated ASCII functions.

In POSIX you specify ASCII encoding by means of the following option:

```
-K literal_encoding_ascii_full
```

> **i** If you want to use ASCII support, you have to specify the MODIFY-MODULE-PROPERTIES statement as follows:
>
> ```
> MODIFY-MODULE-PROPERTIES          -
> ...
> LOWER-CASE-NAMES=*YES,            -
> SPECIAL-CHARACTERS=*KEEP,         -
> ...
> ```
>
> This prevents:
>
> – the names of the ASCII functions (see page 44) from being truncated to eight characters
>
> – lowercase letters from being converted to uppercase and the character “_” from being replaced by “$” in the function names
>
> In POSIX, you specify the following to achieve this:
>
> ```
> -K llm_keep
> -K llm_case_lower
> ```

Parameter transfer, environment variables and global variable *tzname*

The LITERAL-ENCODING option also defines the format in which these strings are transferred to the main function. When LITERAL-ENCODING= *ASCII-FULL, the strings specified are consequently by default transferred to the main function in ASCII format. You can thus produce applications which have been ported to BS2000 or were originally generated as EBCDIC applications as ASCII applications without any need for intervention in the source code.

2.2.6.2 C library functions that support ASCII encoding

For every library function in the C runtime system that works with characters and/or strings or returns a character or string (e.g. `printf`), there is:

– an implementation of the function for processing characters and/or strings in EBCDIC format

– an implementation of the function for processing characters and/or strings in ASCII format

– a macro define that maps the original function (EBCDIC format) to the associated ASCII function

The prototype of an ASCII function and the associated define are stored in the include file in which the corresponding original file is declared. This has the advantage that no additional include files are required to use ASCII-encoded characters and strings, with the possible exception of <ascii_ebcdic.h> (see page 46).

Names of the ASCII functions

The syntax of the names of the ASCII functions is as follows:

__*originalfunction*_`ascii()`

The name of the original function should be specified for *originalfunction*.

The ASCII variant of `printf()`, for example, is `__printf_ascii()`.

C library functions for which there is an ASCII function

`asctime_r()`	`asctime()`	`assert()`	`atof()`
`atoi()`	`atol()`	`atoll()`	`basename()`
`bs2exit()`	`bs2fstat()`	`creat()`	`creat64()`
`ctime_r()`	`ctime()`	`ctime64()`	`ecvt_r()`
`ecvt()`	`faccessat()`	`fchownat()`	`fcvt_r()`
`fdopen()`	`fgetc()`	`fgets()`	`fopen()`
`fopen64()`	`fprintf()`	`fputc()`	`fputs()`
`fread()`	`freopen()`	`freopen64()`	`fscanf()`
`fstatat()`	`fstatat64()`	`futimesat()`	`fwrite()`
`gcvt()`	`getc_unlocked()`	`getenv()`	`getpgmname()`
`gets()`	`gettsn()`	`isalnum()`	`isalpha()`
`isascii()`	`iscntrl()`	`isdigit()`	`isgraph()`
`islower()`	`isprint()`	`ispunct()`	`isspace()`
`isupper()`	`linkat()`	`localeconv()`	`mkfifoat()`
`mknod()`	`mknodat()`	`mktemp()`	`open()`
`open64()`	`openat()`	`openat64()`	`perror()`
`printf()`	`remove ()`	`rename()`	`renameat()`
`scanf()`	`setenv()`	`setlocale()`	`snprintf()`
`sprintf()`	`sscanf()`	`strerror()`	`strlower()`
`strptime()`	`strtod()`	`strtol()`	`strtoll()`
`strtoul()`	`strtoull()`	`strupper()`	`symlinkat()`

`tmpnam()`	`tolower()`	`toupper()`	`ungetc()`
`unlinkat()`	`unsetenv()`	`utimensat()`	`vfprintf()`
`vsnprintf()`	`vsprintf()`		

2.2.6.3 Controlling the mapping of original functions to the associated ASCII variants

You can use the `_ASCII_SOURCE` preprocessor define to specify whether the original library functions (EBCDIC functions) for character/string processing are mapped to the associated ASCII variants. The prototypes of the ASCII functions are always generated.

`_ASCII_SOURCE` can take on the following values:

_ASCII_SOURCE == 0
The EBCDIC functions are not mapped to the corresponding ASCII variants. EBCDIC and ASCII functions can thus be used in parallel. This setting applies regardless of the settings of the compiler (see the `_ASCII` define on page 42).

_ASCII_SOURCE == 1
The EBCDIC functions are mapped to the corresponding ASCII variants. EBCDIC and ASCII functions thus cannot be used in parallel. This setting applies regardless of the settings of the compiler (see the `_LITERAL_ENCODING_ASCII` define on page 42).

You can use the `_MAP_NAME` preprocessor define to specify whether the EBCDIC functions are to be mapped to the ASCII functions by means of the name define method or the macro define method.

i If you want to use the ASCII functions by means of the preprocessor define, you have to use the function declarations of the standard include files (i.e. you have to include the standard include files).

_ASCII_SOURCE is not defined
In this case, the following takes place, depending on the settings of the compiler (see the `_LITERAL_ENCODING_ASCII` define on page 42):

LITERAL_ENCODING_ASCII == 0 or
LITERAL_ENCODING_ASCII not defined
The original functions are not mapped to the corresponding ASCII variants.

LITERAL_ENCODING_ASCII == 1
The original functions are mapped to the corresponding ASCII variants.

i **To control the mapping of the EBCDIC functions to the associated ASCII functions, you have to specify the MODIFY-MODULE-PROPERTIES compiler option as follows:**

```
MODIFY-MODULE-PROPERTIES           -
...
LOWER-CASE-NAMES=*YES,             -
SPECIAL-CHARACTERS=*KEEP,          -
...
```

This prevents:

– the names of the ASCII functions (see page 44) from being truncated to eight characters

– lowercase letters from being converted to uppercase and the character “_” from being replaced with “$” in the function names

In POSIX, you specify the following to achieve this:

```
-K llm_keep
-K llm_case_lower
```

2.2.6.4 Explicitly switching between EBCDIC and ASCII encoding

In addition to the compiler and runtime system extensions for ASCII support described in the above sections, there are also functions for explicitly converting characters and strings between EBCDIC and ASCII representation. This permits EBCDIC and ASCII representation to be mixed within a single module. The conversion functions are declared in the include file <ascii_ebcdic.h>.

The following conversion functions and data are available:

```
char *_a2e(char *str);
char *_e2a(char *str);

char *_a2e_n(char *str, size_t n);
char *_e2a_n(char *str, size_t n);

char *_a2e_max(char *str, size_t n);
char *_e2a_max(char *str, size_t n);

char *_a2e_dup(const char *str);
char *_e2a_dup(const char *str);

char *_a2e_dup_n(const char *str, size_t n);
char *_e2a_dup_n(const char *str, size_t n);
```

Conversion functions are described in detail in the chapter “Functions and variables in alphabetical order” on page 193.

2.2.7 Functions that support IEEE and ASCII encoding

The include files <stdio.h> and <stdlib.h> of the C runtime system contain some functions that support both IEEE floating-point arithmetic and ASCII encoding.

The original functions (/390, EBCDIC) are mapped to the corresponding ASCII/IEEE functions when the preprocessor defines `_IEEE_SOURCE` (see page 40) and `_ASCII_SOURCE` (see page 45) are both set to 1.

Names of the ASCII/IEEE functions

The syntax of the names of these ASCII/IEEE functions is as follows:

__*originalfunction*`_ascii_ieee()`

The name of the original function should be used for *originalfunction*.

The ASCII/IEEE variant of `printf()`, for example, is `__printf_ascii_ieee()`.

C library functions for which there is an ASCII/IEEE function

`atof()`	`ecvt()`	`ecvt_r()`	`fcvt()`	`fcvt_r()`
`fprintf()`	`fscanf()`	`gcvt()`	`fprintf()`	`fscanf()`
`gcvt()`	`printf()`	`scanf()`	`snprintf()`	`sprintf()`
`sscanf()`	`srtod()`	`vfprintf()`	`vsnprintf()`	`vsprintf()`

2.2.8 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 `isw`*xxx* 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.

2.2.9 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 two ways from the time functions used in POSIX/UNIX environments:

– 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`.

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`

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.

2.2.10 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 `SYSPOSIX` 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*’)

2.2.11 Scope of the supported C library

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

Key

x in „Other Standards“ column mean:
Function required by a standard newer than XPG5.

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 Version 2" 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*).

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	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`a641()`		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_r()`		y				
`asctime()`		xx	xx			
`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			
`bcopy()`		x	x			
`brk()`		x	x			
`bs2cmd()`				x	x	

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
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			
cbrt()		x	x			
cdisco()					x	
ceil()		xx	xx			
ceilf()						x
ceill()						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			
clock_gettime()		y				
close()		xxd	xxd			
closedir()		x	x			
closelog()		x	x			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
compile()		x	x			
confstr()		x	x			
cos()		x	xx			
cosh()		xx	xx			
cputime()					x	
creat()		xxd	xxd			
crypt()		x	x			
cstxit()					x	
ctermid()		x	x			
ctime_r()		y				
ctime()		xa	xa			
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			
dirfd()	x					
dirname()		x	x			
div()		xx	xx			
drand48()		x	x			
dup()		x	x			
dup2()		x	x			
ebcdic_to_ascii()				y		
ecvt()		xx	xx			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`_edt()`					x	
`encrypt()`		x	x			
`endgrent()`		x	x			
`endpwent()`		x	x			
`endutxent()`		x	x			
`environ`		x	x			
`epoll_create()`	x					
`epoll_ctl()`	x					
`epoll_wait()`	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			
`faccessat()`		x				
`fattach()`		x	x			
`fchdir()`		x	x			
`fchmod()`		x	x			
`fchmodat()`		x				
`fchown()`		x	x			
`fchownat()`		x				
`fclose()`		xxd	xxd			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`fcntl()`		x	x			
`fcvt()`		xx	xx			
`FD_CLR()`			x			
`FD_ISSET()`			x			
`FD_SET()`			x			
`FD_ZERO()`			x			
`fdelrec()`					x	
`fdetach()`		x	x			
`fdopen()`		xxd	xxd			
`fdopendir()`		x				
`feof()`		xxd	xxd			
`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			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
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			
fstat()		xd	xd			
fstatat()		x				
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				
futimesat()		x				
fwide()		y				x
fwprintf()		y				x
fwrite()		xxd	xxd			x
fwscanf()		y				
gamma()		xx	xx			
garbcoll()					x	
gcvt()		xx	xx			
getc_unlocked()		yd				

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
getc()		xxd	xxd			
getchar_unlocked()		yd				
getchar()		xxd	xxd			
getcontext()		x	x			
getcwd()		x	x			
getdate()		x	x			
getdents()				x		
getdtablesize()		x	x			
getegid()		x	x			
getenv()		xx	xx			
geteuid()		x	x			
getgid()		x	x			
getgrent()		x	x			
getgrgid_r()		x				
getgrgid()		x	x			
getgrnam_r()		x				
getgrnam()		x	x			
getgroups()		x	x			
gethostid()		y	y			
getitimer()		x	x			
getlogin_r()		y				
getlogin()		xx	xx			
getmsg()		x	x			
getopt()		x	x			
getpagesize()		x	x			
getpass()		x	x			
getpgit()		x	x			
getpgmname()					x	
getpgrp()		x	x			
getpid()		x	x			
getpmsg()		x	x			
getppid()		x	x			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`getpriority()`		x	x			
`getpwent()`		x	x			
`getpwnam_r()`		x				
`getpwnam()`		x	x			
`getpwuid_r()`		x				
`getpwuid()`		x	x			
`getrlimit()`		x	x			
`getrusage()`		x	x			
`gets()`		xxd	xxd			
`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			
`getwchar()`		yd	yd			
`getwd()`		x	x			
`gmatch()`				x		
`gmtime_r()`		xa				
`gmtime()`		xa	xa			
`grantpt()`		x	x			
`hcreate()`		x	x			
`hdestroy()`		x	x			
`hsearch()`		x	x			
`hypot()`		xx	xx			
`iconv_close()`		x	x			
`iconv_open()`		x	x			
`iconv()`		x	x			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`ilogb()`		y	y			
`index()`		xx	xx			
`initgroups()`				x		
`initstate()`		x	x			
`insque()`		x	x			
`ioctl()`		x	x			
`isalnum()`		xx	xx			
`isalpha()`		xx	xx			
`isascii()`		xx	xx			
`isastream()`		x	x			
`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			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`iswxdigit()`		x	x			
`isxdigit()`		xx	xx			
`j0()`		xx	xx			
`j1()`		xx	xx			
`jn()`		xx	xx			
`jrand48()`		x	x			
`kill()`		xa	xa			
`killpg()`		x	x			
`l64a()`		y	y			
`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			
`linkat()`		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_r()`		xa				
`localtime()`		xa	xa			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`lockf()`		x	x			
`locs`		x	x			
`log()`		xx	xx			
`log10()`		xx	xx			
`log1p()`		y	y			
`logb()`		y	y			
`longjmp()`		xx	xx			
`_longjmp()`		y	y			
`lrand48()`		x	x			
`lrint()`						x
`lrintf()`						x
`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	
`memccpy()`		x	x			
`memchr()`		xx	xx			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
memcmp()		xx	xx			
memcpy()		xx	xx			
memfree()					x	
memmove()		xx	xx			
memset()		xx	xx			
minor()				x		
mkdir()		x	x			
mkdirat()		x				
mkfifo()		x	x			
mkfifoat()		x				
mknod()		x	x			
mknodat()		x				
mkstemp()		x	x			
mktemp()		xa	xa			
mktime()		xa	xa			
mmap()		x	x			
modf()		xx	xx			
mount()				x		
mprotect()		x	x			
mrand48()		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			
nrand48()		x	x			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
offsetof()					x	
open()		xxd	xxd			
openat()		x				
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			
popen()		x	x			
pow()		xx	xx			
printf()		xxd	xxd			
ptsname()		x	x			
putc_unlocked()		yd				
putc()		xxd	xxd			
putchar_unlocked()		yd				
putchar()		xxd	xxd			
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			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`qsort()`		xx	xx			
`raise()`		xa	xa			
`rand_r()`		y				
`rand()`		xx	xx			
`random()`		x	x			
`re_cmp()`		x	x			
`re_exec()`		x	x			
`read()`		xxd	xxd			
`readdir_r()`		x				
`readdir()`		x	x			
`readlink()`		x	x			
`readlinkat()`		x				
`readv()`		x	x			
`realloc()`		xx	xx			
`realpath()`		x	x			
`regcmp()`		x	x			
`regcomp()`		x				
`regerror()`		x				
`regex()`		x	x			
`regexec()`		x				
`regfree()`		x				
`remainder()`		y	y			
`remove()`		xxd	xxd			
`remque()`		x	x			
`rename()`		xxd	xxd			
`renameat()`		x				
`rewind()`		xxd	xxd			
`rewinddir()`		x	x			
`rindex()`		xx	xx			
`rint()`		y	y			
`rintf()`						x
`rintl()`						x

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`rmdir()`		x	x			
`round()`						x
`roundf()`						x
`roundl()`						x
`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			
`setenv()`	x					
`setgid()`		x	x			
`setgrent()`		x	x			
`setgroups()`	x					
`setitimer()`		x	x			
`setjmp()`		xx	xx			
`_setjmp`		y	y			
`setkey()`		x	x			
`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			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
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			
signgam		x	x			
sigpause()		x	x			
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			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`sleep()`		xa	xa			
`snprintf()`					x	
`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			
`strerror()`		xx	xx			
`strfill()`					x	
`strftime()`		xa	xa			
`strlen()`		xx	xx			
`strlower()`					x	
`strncasecmp()`		x	x			
`strncat()`		xx	xx			
`strncmp()`		xx	xx			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
strncpy()		xx	xx			
strnlen()	x					
strpbrk()		xx	xx			
strrchr()		xx	xx			
strspn()		xx	xx			
strstr()		xx	xx			
strtod()		xa	xa			
strtok_r()		y				
strtok()		xx	xx			
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			
symlinkat()		x				
sync()		x	x			
sysconf()		x	x			
sysfs()				x		
syslog()		x	x			
system()		xa	xa			
tan()		xx	xx			
tanh()		xx	xx			
tcdrain()		x	x			
tcflow()		x	x			
tcflush()		x	x			
tcgetattr()		x	x			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
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()		xa	xa			
_tolower()		x	x			
toupper()		xa	xa			
_toupper()		x	x			
towctrans()		x				x
towlower()		x	x			
towupper()		x	x			
truncate()		x	x			
tsearch()		x	x			
ttyname_r()		x				
ttyname()		x	x			
ttyslot()		x	x			
twalk()		x	x			
tzname		x	x			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`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			
`unlinkat()`		x				
`unlockpt()`		x	x			
`unsetenv()`	x					
`usleep()`		x	x			
`utime()`		x	x			
`utimensat()`		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			
`vfwprintf`		y				x
`vprintf()`		xxd	xxd			
`vsnprintf()`		y			x	
`vsprintf()`		xx	xx			
`vswprintf()`		y				x
`vwprintf()`		y				x
`wait()`		x	x			
`wait3()`		x	x			
`waitid()`		x	x			
`waitpid()`		x	x			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`wcrtomb()`		y				
`wcscat()`			x			
`wcschr()`			x			
`wcscmp()`			x			
`wcscoll`		x	x			
`wcscpy()`		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
`wctype()`		x	x			
`wcwidth()`		x	x			

Function	Other Standards	XPG5	XPG4 Version 2	Extensions		ANSI
				Extension	*BS2000*	
`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			

2.3 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 108) 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.

2.3.1 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 51ff) are supported.
- All functions identified as extensions (marked with an x in the "Extension" and "BS2000" columns in the table on page 51ff) 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 75. 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.

2.3.2 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" [6].

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 51ff.) 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 51 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.

2.3.3 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.

2.3.3.1 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`).

2.3.3.2 Setting the PROGRAM_ENVIRONMENT variable

The `PROGRAM_ENVIRONMENT` environment variable is used in BS2000 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 104.

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.

2.3.3.3 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" [1]) 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.

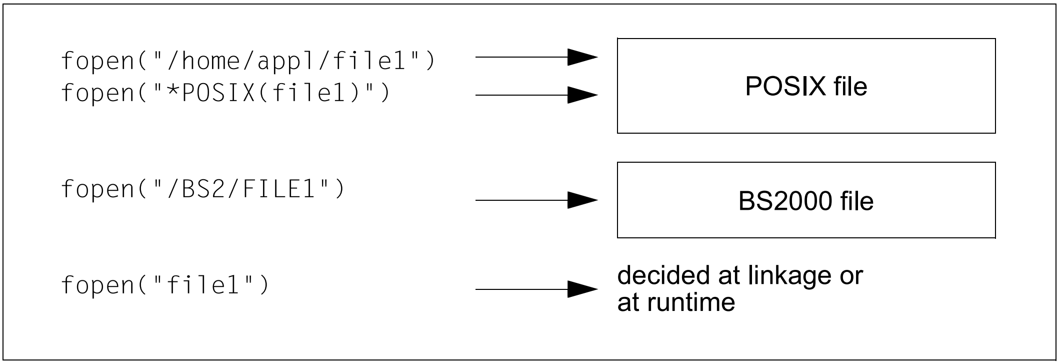

Figure 1: Control options at source code level

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.

2.4 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 either 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 33). 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>.

2.5 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” on page 193). 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 manuals "C Compiler" [3] and "C/C++ Compiler" [4]).

2.6 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.

2.6.1 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
<eigth>	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 "XHCS" manual [13] .

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.

2.6.2 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`

2.7 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 104 and the manual "POSIX Commands" [2]). 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:

`atof()`	`isalnum()`	`isalpha()`	`isascii()`	`iscntrl()`
`isdigit()`	`isgraph()`	`islower()`	`isprint()`	`ispunct()`
`isspace()`	`isupper()`	`isxdigit()`	`localeconv()`	`setlocale()`
`strcoll()`	`strftime()`	`strtod()`	`strxfrm()`	`tolower()`
`toupper()`	`wctomb()`	`wcstombs()`		

The C runtime library provides some predefined locales (see the section “Predefined locales” on page 88). However, users may also define their own locales (see section “User-specific locales” on page 103).

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 104).

2.7.1 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.

2.7.1.1 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*.

2.7.1.2 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 80 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
`decimal_point`	"<period>"
`thousands_sep`	""
`grouping`	""

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

langinfo constant	Value
`YESEXPR`	`"^[yY]"`
`NOEXPR`	`"^[nN]"`
`YESSTR` Will no longer be supported by the X/Open-Standard in future.	`"yes"`
`NOSTR` 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
`int_curr_symbol`	""
`currency_symbol`	""
`mon_decimal_point`	""
`mon_thousands_sep`	""
`mon_grouping`	""

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_AMPM	"%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"

2.7.1.3 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’C0‘` and `X‘D0‘` are in the character class `punct`. In the `"C"` locale, all of these characters belong to the character class `cntrl` (i.e. control characters).

2.7.1.4 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.

2.7.1.5 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

2.7.1.6 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-sqare-bracket>	[	punct	5B	BB
<backslash>	\	punct	5C	BC
<right-sqare-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>	¯	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:

alpha	The character belongs to the `upper` or `lower` class.
alnum	The character belongs to the `alpha` or `digit` class.
print	The character belongs to the `alnum` or `punct` class or the character is the <space> character.
graph	The character belongs to the `alnum` or `punct` class.

The `toupper` and `tolower` 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: ","
thousend_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

2.7.2 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 89). 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*

2.8 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" [2]). 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" [2]).

Supplying the environment variables with default values from BS2000

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" [9]). 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
%N	Value of the name parameter passed to `catopen()`
%L	Value of `LANG`
%l	Value of the language element from `LANG`
%t	Value of the territory element from `LANG`
%c	Value of the codeset element from `LANG`
%%	A single % character

The behavior of the language information function `nl_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()`.

2.9 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. See also the section “Scope of the supported C library” on page 49.

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

Functions that reject BS2000 files

The following functions process only POSIX files (see also the manual "POSIX Basics" [1]). All of these functions - except `sync()` - set `errno` to `EINVAL` if an attempt is made to access BS2000 files.

`access()`	`chmod()`	`chown()`	`dup()`
`dup2()`	`faccessat()`	`fchmod()`	`fchmodat()`
`fchown()`	`fchownat()`	`fcntl()`	`fdopendir()`
`fpathconf()`	`fstatat()`	`fsync()`	`futimesat()`
`isatty()`	`link()`	`linkat()`	`mkdirat()`
`mkfifo()`	`mkfifoat()`	`mknod()`	`mknodat()`
`openat()`	`pathconf()`	`readlink()`	`readlinkat()`
`renameat()`	`symlink()`	`symlinkat()`	`sync() *)`
`sysfs()`	`tcdrain()`	`tcflow()`	`tcflush()`
`tcgetattr()`	`tcgetpgrp()`	`tcsendbreak()`	`tcsetattr()`
`tcsetpgrp()`	`tempnam() *)`	`unlinkat()`	`utime()`
`utimensat()`			

`*)` `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 110).

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.

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

2.9.1 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()`.

2.9.1.1 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()`.

2.9.1.2 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.

2.9.1.3 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 75), standard I/O streams may be associated with POSIX or BS2000 files.

The following association is created when accessing the DMS:

`stdin`	SYSDTA
`stdout`, `stderr`	SYSOUT

In this case, behavior is compatible with the earlier versions of the C runtime system (see also section “BS2000 system files” on page 115).

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 75)

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.

2.9.2 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 113.

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 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" [1].

2.9.3 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.

2.9.4 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.

2.9.4.1 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 `"(SYSTERM)"` 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 `"(SYSTERM)"`. 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 `"(SYSTERM)"` 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("(SYSTERM)", "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 `"(SYSTERM)"`.

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*.

2.9.4.2 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	\ n	\ t	\ f	\ v	\ r	\ b
SAM/ISAM	X	X				
SYSOUT/SYSTERM	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`, `SYSTERM` 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.

2.9.4.3 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,

– Last Byte Pointer (LBP).

2.9.4.4 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 122. 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,*n*)**	**RECSIZE (*r* byte)**	**Maximum number of data bytes**
<u>SAM</u> [1)]	<u>V</u>	PAMKEY	<u>1</u>≤ *n* ≤16	4≤ *r* ≤*n*<u>*2048-4</u>	RECSIZE - 4
		DATA(2K)	<u>1</u>≤ *n* ≤16	4≤ *r* ≤*n*<u>*2048-16</u>	RECSIZE - 4
		DATA(4K)	<u>2</u>≤ *n* ≤16		
	U	PAMKEY	<u>1</u>≤ *n* ≤16		BLKSIZE
		DATA(2K)	<u>1</u>≤ *n* ≤16		BLKSIZE - 16
		DATA(4K)	<u>2</u>≤ *n* ≤16		
ISAM [2)]	V	PAMKEY	<u>1</u>≤ *n* ≤16	12≤ *r* ≤*n*<u>*2048</u>	RECSIZE - 12
		DATA(2K)	<u>1</u>≤ *n* ≤16	12≤ *r* ≤*n*<u>*2048</u>	RECSIZE - 12
		DATA(4K)	<u>2</u>≤ *n* ≤16		

1) SAM files are only created in the KR mode (see also the `SOURCE-PROPERTIES` option in the manuals "C Compiler" [3] and "C/C++ Compiler" [4]) 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,*n*)**	**RECSIZE (*r* byte)**	**Maximum number of data bytes**
<u>SAM</u>	<u>F</u>	PAMKEY	<u>1</u>≤ *n* ≤16	1≤ *r* ≤*n*<u>*2048</u>	RECSIZE
		DATA(2K)	<u>1</u>≤ *n* ≤16	1≤ *r* ≤*n*<u>*2048-16</u>	RECSIZE
		DATA(4K)	<u>2</u>≤ *n* ≤16		
	V	PAMKEY	<u>1</u>≤ *n* ≤16	4≤ *r* ≤*n*<u>*2048-4</u>	RECSIZE - 4
		DATA(2K)	<u>1</u>≤ *n* ≤16	4≤ *r* ≤*n*<u>*2048-16</u>	RECSIZE - 4
		DATA(4K)	<u>2</u>≤ *n* ≤16		
	U	PAMKEY	<u>1</u>≤ *n* ≤16		BLKSIZE
		DATA(2K)	<u>1</u>≤ *n* ≤16		BLKSIZE - 16
		DATA(4K)	<u>2</u>≤ *n* ≤16		

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

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.

2.9.4.5 K and NK block formats

BS2000 supports volumes with different formats:

– **K**ey 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.

– **N**on-**K**ey 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).

Additionally, 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.

Please refer to the "DMS Introductory Guide" manual for a detailed description of the BLOCK-CONTROL-INFO operand, various file and data volume structures and the conversion from K file format to NK file format.

Please refer to the "DMS Introductory Guide" manual [11] for a detailed description of the `BLOCK-CONTROL-INFO` operand, various file and data volume structures and the conversion from K file format to NK file format.

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

2.9.4.6 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).
```

2.9.4.7 Support for the DIV access method

The access method DIV (DATA IN VIRTUAL) 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".

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.

2.9.4.8 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.

2.9.4.9 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`.

With the `creat()` or `freopen()` functions, the file must always be opened in binary mode and the specification of O_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:

`creat()`, `fopen()`, `freopen()`, `open()`	Open
`close()`, `fclose()`	Close
`fread()`, `read()`	Read
`fwrite()`, `write()`	Write
`fsetpos()`, `fgetpos()`	Set file position to determined location
`fseek()`, `lseek()`	Set file position to start/end of file
`rewind()`	Set file position to start of file
`flocate()`	Explicitly positioning in an ISAM file
`fdelrec()`	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.

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

- If `O_RECORD` is specified the file must have `FCBTYPE` SAM, PAM or ISAM.

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.

2.9.5 Last Byte Pointer (LBP)

In BS2000 the length of a PAM file is always an integral multiple of a PAM block, regardless of its content. In BS2000 OSD/BC V10.0 and higher, the catalog entry for PAM files contains the entry Last Byte Pointer (LBP), in which the real length of the file in bytes can be stored. As a result, especially files which are stored on a network server (NAS) can also be read and written by all systems which access them (also UNIX) in a manner which is precise to the byte.

This functionality may also be available in BS2000/OSD as of V8.0.

Previously the length of a PAM file was determined with the help of an auxiliary construction by identifying the actual end of the file with a marker. This auxiliary construction can be dispensed if LBP is used.

All C runtime functions which open PAM files are affected by this interface.

The `fopen()`, `freopen()`, `open()` and `creat()` functions have consequently been extended with the *lbp* switch. Details can be found in the descriptions of the relevant functions.

When existing files are opened or read, these functions behave as follows independently of the *lbp* switch:

– If the file's LBP is not equal to 0, it is evaluated. Any marker which is present is ignored.

– When LBP = 0, a marker is searched for, and the file length is determined from this. If no marker is found, the end of the last complete block is regarded as the end of file.

When files are closed which have been modified or newly created, the behavior depends on the *lbp* switch when the file is opened or on the environment variable LAST_BYTE_POINTER.

Environment variable LAST_BYTE_POINTER

The purpose of the environment variable LAST_BYTE_POINTER is to enable existing programs to use the LBP without any need to intervene in them. Permanently linked programs then only need to be relinked with the current CRTE. For programs linked with PARTIAL-BIND or CRTE-BASYS, it is sufficient if the current CRTE or CRTE-BASYS is installed.

If one of the functions affected is called without the *lbp* switch, its behavior depends on the environment variable LAST_BYTE_POINTER:

`LAST_BYTE_POINTER=YES`

The `fopen()` and `freopen()` functions behave as if `lbp=yes` were specified in the `mode` parameter.

The `open()` and `creat()` functions behave as if O_LBP were specified in the `mode` parameter.

When a file is opened, a check is made to see whether LBP support is possible. If this is not the case, the function concerned will fail and `errno` is set to ENOSYS.

When a file which has been modified or newly created is closed, no marker is written (even if one was present), and a valid LBP is set. In this way files with a marker can be converted to LBP without a marker.
In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

`LAST_BYTE_POINTER=NO`

The `fopen`, `fopen64` and `freopen`, `freopen64` functions behave as if `lbp=no` were specified in the `mode` parameter.

The `open()` and `creat()` functions behave as if O_NOLBP were specified in the `mode` parameter.

When a file which has been **newly created** is closed, the LBP is set to zero (=invalid). A marker is written. In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

When a file which has been **modified** is closed, the LBP is set to zero (=invalid). A marker is written only if a marker existed before. If the file had a valid LBP when it was opened, no marker is written as in this case it is assumed that no marker exists.

In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

If the environment variable is not set, the functions behave as if it had the value `NO`.

Details on using environment variables can be found in section “Environment variables” on page 104).

2.9.6 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.

2.10 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.

2.10.1 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.

2.10.2 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 134.

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 130.

2.10.2.1 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.

2.10.2.2 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 131 . 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 134. 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 143, `tcdrain()`, `tcflow()`, `tcflush()`, `tcsendbreak()` and `tcsetattr()`).

2.10.2.3 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 137) and `c_lflag` (see section “Local modes” on page 143) 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.

2.10.2.4 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 134. 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 134), 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.

2.10.2.5 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.

2.10.2.6 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 139). 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()`.

2.10.2.7 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 `ISIG` flag is set. It generates a `SIGINT` signal (interrupt) which is sent to all processes in the foreground process group associated with the terminal. If the `ISIG` flag is set, the `INTR` 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 `sigaction()` and `signal()`).

`QUIT` Special character on input, which is recognized if the `ISIG` flag is set. It generates a `SIGQUIT` signal (quit) which is sent to all processes in the foreground process group associated with the terminal. If `ISIG` is set, the `QUIT` character is discarded when processed. Its treatment is almost identical to the interrupt signal `SIGINT`, 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 `sigaction()`).

`ERASE` Special character on input, which is recognized if the `ICANON` flag is set. It erases the preceding character, but not beyond the start of a line, i.e. an `NL`, `EOF` or `EOL` character (see also the section “Canonical mode input processing” on page 132). If `ICANON` is set, the `ERASE` character is discarded when processed.

`KILL`	Special character on input, which is recognized if the `ICANON` flag is set. It deletes the entire line as of the last `NL`, `EOF` or `EOL` character. If `ICANON` is set, the `KILL` character is discarded when processed. This character is not supported on BS2000 block terminals.
`EOF`	Special character on input, which is recognized if the `ICANON` flag is set. When `EOF` is received, all the bytes waiting to be read are immediately passed to the program without waiting for an `NL` character, and the `EOF` is discarded. If no bytes are present, i.e. the `EOF` is at the beginning of a line, `read()` returns a value of 0. A return value of 0 for a read operation is the default end-of-file identifier. If `ICANON` is set, the `EOF` character is discarded when processed.
`NL`	Special character on input, which is recognized if the `ICANON` flag is set. `NL` is the standard line delimiter `\n` and cannot be changed.
`EOL`	Special character on input, which is recognized if the `ICANON` flag is set. `EOL` is an additional line delimiter and serves the same function as `NL`. It is normally not used.
`SUSP`	If job control is supported (section "Special control characters" on page 144) on an X/Open-compatible system, the `SUSP` special character is recognized on input. If the `ISIG` flag is set, receipt of the `SUSP` character causes a `SIGTSTP` signal to be sent to all processes in the foreground process group associated with the terminal. After it is processed, the `SUSP` character is discarded as well. This character has no effect in the POSIX subsystem, since job control is not supported in its current implementation.
`STOP`	Special character on both input and output, which is recognized if either the `IXON` (for input) or `IXOFF` (for output) flag is set. `STOP` can be used to suspend output temporarily. It is useful with CRT terminals to prevent output from disappearing before it can be read. If `IXON` is set, the `STOP` character is discarded when processed. So long as the output is suspended, additional `STOP` characters are ignored and not read. The `STOP` character can be neither changed nor escaped. This character is not supported on BS2000 block terminals.
`START`	Special character on both input and output, which is recognized if either the `IXON` (for input) or `IXOFF` (for output) flag is set. The `START` character is used to resume output that has been suspended by a `STOP` character. So long as output continues, subsequent `START` characters are ignored and not read. If `IXON` is set, the `START` character is discarded when processed. The `START` character can be neither changed nor escaped. This character is not supported on BS2000 block terminals.

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.

2.10.2.8 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 141), 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.

2.10.2.9 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.

2.10.3 Settable parameters

2.10.3.1 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
`tcflag_t`		`c_iflag`	input modes
`tcflag_t`		`c_oflag`	output modes
`tcflag_t`		`c_cflag`	control modes
`tcflag_t`		`c_lflag`	local modes
`cc_t`	`NCCS`	`c_cc[]`	control characters

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

2.10.3.2 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 141). 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.

2.10.3.3 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
OPOST	Post-process output
OLCUC	Map lowercase letters to uppercase on output. Will no longer be supported by the X/Open-Standard in future.
ONLCR	Map NL to CR-NL on output
OCRNL	Map CR to NL on output
ONOCR	No CR output at column 0
ONLRET	NL performs CR function
OFILL	Use fill characters for delay
OFDEL	The fill character is DEL (otherwise NUL)
NLDLY NL0 NL1	Select newline (NL) delays NL character type 0 NL character type 1
CRDLY CR0 CR1 CR2 CR3	Select carriage return (CR) delays: CR delay type 0 CR delay type 1 CR delay type 2 CR delay type 3
TABDLY TAB0 TAB1 TAB2 TAB3	Select delays for horizontal tabs: Horizontal-tab delay type 0 Horizontal-tab delay type 1 Horizontal-tab delay type 2 Expand tabs to spaces
BSDLY BS0 BS1	Select delays for backspace: Backspace-delay type 0 Backspace-delay type 1

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

2.10.3.4 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`
`CS5`
`CS6`
`CS7`
`CS8`
	Character size (number of bits per byte):
	5 bits
	6 bits
	7 bits
	8 bits

`CSTOPB`	Send 2 stop bits (else 1)
`HUPCL`	Hang up on last `close()`
`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

Name	Definition
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.

2.10.3.5 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
NOFLSH	Disable flushing of input or output queues after INTERRUPT or QUIT from keyboard
TOSTOP	Send SIGTTOU 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 132.

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 133 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.

2.10.3.6 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 134), 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.

2.10.4 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

2.10.5 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.

2.11 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.

2.11.1 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	INTR
SIGQUIT	QUIT
SIGSTOP	STOP
SIGTSTP	SUSP
SIGCONT	START

The STOP and START keys are not supported on block terminals (see also section “Block terminal support” on page 145).

2.11.2 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()	execle()	exit()
execlp()	fork()	execv()

2.11.2.1 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 *xxx*`get()`. 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 *xxx*`get()` with which a new element can be created or an existing one made available to a process. The parameters of the *xxx*`get()` 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 *xxx*`get()` 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 *xxx*`get()` system calls will fail. Thus, any permissions that extend beyond those available may not be requested. In order to modify access permissions, the *xxx*`ctl()` system call must be executed (see below). When the `IPC_CREAT` flag is additionally OR-ed with the `IPC_EXCL` flag, *xxx*`get()` 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 *xxx*`get()` system calls will fail.

The *xxx*`get()` 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 *xxx*`ctl()` 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

xxx`ctl()`

Each facility has a corresponding system call *xxx*`ctl()`, 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 *xxx*`get()` 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.

2.11.2.2 Shared memory

Shared memory is provided for the C library functions (see the manual "Executive Macros" [10]).Shared memory segments are created by the `shmget()` function; 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`.

2.11.3 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 "Executive Macros" [10].

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

2.11.3.1 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:

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

2.11.3.2 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.

2.11.3.3 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.

2.11.3.4 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 _mnpc      4       /* evccomessnopostcode */
#define _nmpc      8       /* evcnocomesspostcode */
#define _mpc       12      /* evccomesspostcode */
```

```
#define _etnm      0       /* evcelapsedtimenocomess */
#define _etm       4       /* evcelapsedtimecomess */
#define _disnm     16      /* evceventdisablednocomess */
#define _dism      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 `lonjmp()` 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.

2.11.3.5 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 stxrout(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.

2.12 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 in CRTE a thread-safe variant is also supplied.

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” (see page 193).

- 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>

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” (see page 193).

In addition, the functions of the _POSIX_THREAD_SAVE_FUNCTIONS group were added. A list of these functions can be found on page 174. These functions are described in detail in the chapter “Functions and variables in alphabetical order” (see page 193).

2.13 Programming notes

2.13.1 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);
```

2.13.2 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” on page 193ff 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” on page 193ff.

2.13.3 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).

3 Functions and variables arranged by theme

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

3.1 File processing

File management

"basename - return last element of pathname" on page 217

"chdir - change working directory" on page 245

"chmod, fchmodat - change mode of file" on page 246

"chown, fchownat - change owner and group of file" on page 248

"chroot - change root directory" on page 251

"clearerr - clear end-of-file and error indicators" on page 252

"closedir - close directory" on page 257

"creat - create new file or overwrite existing one" on page 265

"dirfd - extract file descriptor" on page 283

"dirname - parent directory of pathname" on page 284

"fchdir - change current directory" on page 323

"fchmod - change mode of file" on page 324

"fchown - change owner or group of file" on page 327

"fcntl - control open file" on page 331

"FD_CLR, FD_ISSET, FD_SET, FD_ZERO - macros for synchronous I/O multiplexing" on page 337

"fstat, fstatat - get file status of open file" on page 423

"fstatvfs, statvfs - read file system information" on page 427

"ftw - traverse (walk) file tree" on page 439

“fwide - specify file orientation” on page 443
“getcwd - get pathname of current working directory” on page 468
“getdtablesize - get size of descriptor table” on page 477
“getwd - get pathname of current working directory” on page 524
“lchown - change owner/group of file” on page 589
“lstat - query file status” on page 626
“link, linkat - create link to file” on page 594
“mkdir, mkdirat - make directory” on page 647
“mknod, mknodat - make directory, special file, or text file” on page 651
“mkstemp - make unique temporary file name” on page 654
“mktemp - make unique temporary file name (extension)” on page 655
“nftw - traverse file tree” on page 683
“opendir, fdopendir - open directory” on page 699
“readlink, readlinkat - read contents of symbolic link” on page 737
“remove - remove files” on page 763
“rename, renameat - rename file” on page 765
“rewinddir - reset file position indicator to start of directory stream” on page 770
“rmdir - remove directory” on page 773
“seekdir - set position of directory stream” on page 778
“stat - get file status” on page 854
“statvfs - read file system information” on page 858
“symlink, symlinkat - make symbolic link to file” on page 901
“sync - update superblock” on page 904
“telldir - get current location of named directory stream” on page 926
“tempnam - create pathname for temporary file” on page 927
“tmpfile - create temporary file” on page 933
“tmpnam - create base name for temporary file” on page 934
“umask - get and set file mode creation mask” on page 949
“unlink, unlinkat - remove link” on page 955

File access

64-bit functions to support NFS V3.0

"mmap64 - map memory pages" on page 660

"open64 - open file" on page 689

"readdir64 - read from directory" on page 734

"setrlimit64 - set limit value for a resource" on page 807

"stat64 - query file status" on page 854

"statvfs64 - read file system information" on page 858

3.2 I/O on terminal

“fgetc - get byte from stream” on page 348

“fgets - get string from stream” on page 351

“fgetwc - get wide character string from stream” on page 352

“fgetws - get wide character string from stream” on page 354

“fprintf, printf, sprintf - write formatted output on output stream” on page 378

“fputc - put byte on stream” on page 392

“fputs - put string on stream” on page 394

“fputwc - put wide-character code on stream” on page 395

“fputws - put wide character string on stream” on page 397

“fread - read binary data” on page 398

“fscanf, scanf, sscanf - read formatted input” on page 404

“fwprintf, swprintf, vfwprintf, vswprintf, vwprintf, wprintf - output formatted wide characters” on page 444

“fwrite - output binary data” on page 451

“fwscanf, swscanf, wscanf - formatted read” on page 453

“getc - get byte from stream” on page 461

“getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked - standard I/O with explicit lock by the client” on page 463

“getmsg - get message from STREAMS file” on page 490

“getopt, optarg, optind, opterr, optopt - command option parsing” on page 493

“getpass - read string of characters without echo” on page 497

3.3 Processes

Process administration

Process control and signals

Interprocess communication

Diagnostics and messages

3.4 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 208

"ctime_r - thread-safe conversion of date and time to string" on page 277

"getgrgid_r - get group file entry for group ID (thread-safe)" on page 481

"getgrnam_r - get group file entry for group name (thread-safe)" on page 483

"getlogin_r - get login name (thread-safe)" on page 489

"getpwnam_r - get user name (thread-safe)" on page 505

"gmtime_r - convert date and time to UTC (thread-safe)" on page 527

"localtime_r - convert date and time to string (thread-safe)" on page 608

"rand_r - pseudo-random number generator (int, thread-safe)" on page 729

"readdir_r - read directory (thread-safe)" on page 736

"strtok_r - split string into tokens (thread-safe)" on page 889

"ttyname - find pathname of terminal" on page 942

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 358

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 463

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 197

“alarm - schedule alarm signal” on page 204

“atexit - register function to run at process termination” on page 212

“exit, _exit - terminate process” on page 316

“fcntl - control open file” on page 331

“fork - create new process” on page 375

“getcontext, setcontext - display or modify user context” on page 466

“getpid - get process ID” on page 499

"getrlimit - get limit value for a resource" on page 508

"getpriority - call process priority" on page 501

“kill - send signal to process or process group” on page 585

“lockf - lock file section” on page 609

“msgrcv - receive message from queue” on page 673

“msgsnd - send message to queue” on page 675 "nice - change priority of process" on page 686

"open, openat - open file" on page 689

"pause - suspend process until signal is received" on page 705

"raise - send signal to calling process" on page 727

"read - read bytes from file" on page 731

"semop - semaphore operations" on page 786

"setcontext - modify user context" on page 791

"setlocale - set or query locale" on page 799

"sigaction - examine and change signal handling" on page 820

"sigpause - remove signal from signal mask and deactivate process" on page 840

"sigpending - examine pending signals" on page 840

"sigsetjmp - set label for non-local jump using signal" on page 844

"sigsuspend - wait for signal" on page 847

"sleep - suspend process for fixed interval of time" on page 849

"usleep - suspend process for defined interval" on page 960

"wait, waitpid - wait for child process to stop or terminate" on page 978

"wait3 - wait for status change of child processes" on page 982

"waitid - wait for status change of child processes" on page 983

"write - write bytes to file" on page 1025

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 329

"fflush - flush stream" on page 345

"fputc - put byte on stream" on page 392

"fputwc - put wide-character code on stream" on page 395

"fseek - reposition file position indicator in stream" on page 416

"write - write bytes to file" on page 1025

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 206 [1]

“basename - return last element of pathname” on page 217

“brk, sbrk - modify size of data segment” on page 219

“chroot - change root directory” on page 251

“ctime, ctime64 - convert date and time to string” on page 276 [1]

“cuserid - get login name” on page 278

"dbmclearerr - function for administering dbm databases" on page 280

“div - divide with integers” on page 285

“ecvt, fcvt, gcvt - convert floating-point number to string” on page 291

“endgrent, getgrent, setgrent - group management” on page 294

“endpwent, getpwent, setpwent - manage user catalog” on page 296

“endutxent, getutxent, getutxid, getutxline, pututxline, setutxent - manage utmpx entries” on page 298

“fcvt - convert floating-point number to string” on page 337

“gamma - compute logarithm of gamma function” on page 459

“gcvt - convert floating-point number to string” on page 460

“getdtablesize - get size of descriptor table” on page 477

“getenv - get value of environment variable” on page 478

“getgrent - get group file entry” on page 479

“getpwent - read user data from user catalog” on page 503

“getutxent, getutxid, getutxline - get utmpx entry” on page 519

“getgrgid - get group file entry for group ID” on page 480 [1]

“getgrnam - get group file entry for group name” on page 482 [1]

“getlogin - get login name” on page 488 [1]

“getpagesize - get current page size” on page 496

“getpass - read string of characters without echo” on page 497

[1] use reentrant function ("_r" extension)

“getpwnam - get user name” on page 504 [1]

“getw - read word from stream” on page 520

"initstate - generate pseudo-random number" on page 538 [2]

“localtime, localtime64 - convert date and time to local time” on page 606 [1]

“longjmp - execute non-local jump” on page 616 [3]

“ptsname - name of pseudoterminal” on page 714

“putenv - change or add environment variables” on page 717

“pututxline - write utmpx entry” on page 723

“putw - put word on stream” on page 724

“rand - pseudo-random number generator (int)” on page 729 [2]

“readdir - read directory” on page 734 [3]

“sbrk - modify size of data segment” on page 776

“setgrent - reset file position indicator to beginning of group file” on page 794

“setpwent - delete pointer to search user catalog” on page 804

“setutxent - reset pointer to utmpx file” on page 810

“siglongjmp - execute non-local jump using signal” on page 836 [3]

“signgam - variable for sign of lgamma” on page 840

“sigprocmask - examine or change blocked signals” on page 841 [4]

“sigset - modify signal handling” on page 843

“strtok - split string into tokens” on page 888 [1]

“ttyname - find pathname of terminal” on page 942 [1]

“ttyslot - find entry of current user in utmp file” on page 944

“wait3 - wait for status change of child processes” on page 982

1 use reentrant function ("_r" extension)

2 use `rand_r()` reentrant function

3 The result of calling this function is undefined when the `jmp_buf` structure was not initialized in the calling thread.

4 use `pthread_sigmask` function

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.

3.5 Memory management and memory operations

"bcmp - compare memory areas" on page 218

"bcopy - copy memory area" on page 218

"brk, sbrk - modify size of data segment" on page 219

"bzero - initialize memory with X'00'" on page 231

"calloc - allocate memory" on page 232

"free - free allocated memory" on page 400

"garbcoll - release memory space to system (BS2000)" on page 460

"malloc - memory allocator" on page 631

"memalloc - memory allocator (BS2000)" on page 638

"memchr - find byte in memory" on page 640

"memcmp - compare bytes in memory" on page 641

"memfree - free memory area (BS2000)" on page 643

"memmove - copy bytes in memory with overlapping areas" on page 644

"memset - initialize memory area" on page 645

"mmap - map memory pages" on page 660

"mprotect - modify access protection for memory mapping" on page 667

"msync - synchronize memory" on page 677

"munmap - unmap memory pages" on page 679

"offsetof - get offset of structure component from start of structure (BS2000)" on page 688

"realloc - memory reallocator" on page 741

"swab - swap bytes" on page 900

"valloc - request memory aligned with page boundary" on page 970

3.6 System environment

3.7 Characters and strings

Single character processing

"ffs - seek first set bit" on page 347

"isalnum - test for alphanumeric character" on page 556

"isalpha - test for alphabetic character" on page 557

"isascii - test for 7-bit ASCII character" on page 558

"iscntrl - test for control character" on page 561

"isdigit - test for decimal digit" on page 562

"isebcdic - test for EBCDIC character (BS2000)" on page 563

"isgraph - test for visible character" on page 564

"islower - test for lowercase letter" on page 565

"isprint - test for printing character" on page 567

"ispunct - test for punctuation character" on page 568

"isspace - test for white-space character" on page 569

"isupper - test for uppercase letter" on page 570

"iswalnum - test for alphanumeric wide character" on page 571

"iswalpha - test for alphabetic wide character" on page 572

"iswcntrl - test for control wide character" on page 573

"iswctype - test wide character for class" on page 574

"iswdigit - test for decimal digit wide character" on page 575

"iswgraph - test for visible wide character" on page 576

"iswlower - test for lowercase wide character" on page 577

"iswprint - test for printing wide character" on page 578

"iswpunct - test for punctuation wide character" on page 579

"iswspace - test for white-space wide character" on page 580

"iswupper - test for uppercase wide character" on page 581

"iswxdigit - test for hexadecimal digit wide character" on page 582

"isxdigit - test for hexadecimal digit" on page 583

String processing

Character and string conversions

3.8 Conversion of entities

"atof - convert string to double-precision number" on page 213

"atoi - convert string to integer" on page 214

"atol - convert string to long integer" on page 215

"atoll - convert string to long long integer (long long int)" on page 216

"ecvt, fcvt, gcvt - convert floating-point number to string" on page 291

"fcvt - convert floating-point number to string" on page 337

"gcvt - convert floating-point number to string" on page 460

"getdents - convert directory entries" on page 475

"getsubopt - get suboptions from string" on page 516

"l64a - convert 32-bit integer number to string" on page 588

"strftime - convert date and time to string" on page 872

"strtod - convert string to double-precision number" on page 886

"strtol - convert string to long integer" on page 890

"strtoll - convert string to long long integer" on page 892

"strtoul - convert string to unsigned long integer" on page 894

"strtoull - convert string to unsigned long long" on page 896

"toascii - convert integer to legal value" on page 935

"toebcdic - convert integer to legal value (BS2000)" on page 936

"wcsftime - convert date and time to wide character string" on page 992

"wcstod - convert wide character string to double-precision number" on page 1002

"wcstol - convert wide character string to long integer" on page 1005

"wcstoll - convert wide character string to long long integer" on page 1007

"wcstoul - convert wide character string to unsigned long" on page 1010

"wcstoull - convert wide character string to unsigned long long" on page 1012

3.9 Regular expressions

3.10 Time functions

“getsubopt - get suboptions from string” on page 516
“localtime, localtime64 - convert date and time to local time” on page 606
“localtime_r - convert date and time to string (thread-safe)” on page 608
“mktime, mktime64 - convert local time into time since the Epoch” on page 657
“_ _TIME_ _ - macro for compilation time” on page 929
“time, time64 - get time since the Epoch” on page 930
“timezone - variable for difference between local time and UTC” on page 932
“tzname - array variable for timezone strings” on page 945
“tzset - set timezone conversion information” on page 946
“ualarm - set interval timer” on page 947
“usleep - suspend process for defined interval” on page 960

3.11 Math functions

Arithmetic with integers

"abs - return integer absolute value" on page 198
"div - divide with integers" on page 285
"labs - return long integer absolute value" on page 588
"ldiv - long division of integers" on page 592
"llabs - return absolute value of an integer (long long int)" on page 597
"lldiv - division of integers (long long int)" on page 598
"llrint, llrintf, llrintl - round to nearest integer value (long long int)" on page 599
"llround, llroundf, llroundl - round up to next integer value (long long int)" on page 600
"lrint, lrintf, lrintl - round to nearest integer value (long int)" on page 619
"lround, lroundf, lroundl - round up to next integer value (long int)" on page 620
"rint, rintf, rintl - round to nearest integer value" on page 772
"round, roundf, roundl - round up to next integer value" on page 775

Arithmetic with floating-point numbers

"cabs - calculate absolute value of complex number *(BS2000)" on page 231*
"ceil, ceilf, ceill - round up floating-point number" on page 239
"cbrt - cube root" on page 237
"errno - variable for error return values" on page 310
"exp - use exponential function" on page 319
"expm1 - compute exponential function" on page 319
"fabs - compute absolute value of floating-point number" on page 320
"floor, floorf, floorl- round off floating point number" on page 360
"fmod - compute floating-point remainder value function" on page 361
"frexp - extract mantissa and exponent from double precision number" on page 403
"gamma - compute logarithm of gamma function" on page 459
"hypot - Euclidean distance function" on page 531
"ilogb - get exponent part of floating-point number" on page 536

Trigonometric, hyperbolic and arc functions

Random numbers

“drand48 - generate pseudo-random numbers between 0.0 and 1.0” on page 286

"erand48 - generate pseudo-random numbers between 0.0 and 1.0 with initialization value" (see drand48 on page 286)

“initstate, random, setstate, srandom - generate pseudo-random numbers” on page 538

"jrand48 - generate pseudo random numbers between -2^{31} and 2^{31} with initialization value" (see drand48 on page 286)

„lcong48 - pseudo-random number (signed long int) generator" (see drand48 on page 286)

"lrand48 - generate pseudo random numbers between 0 and 2^{31}" (see drand48 on page 286)

"mrand48 - generate pseudo random numbers between -2^{31} and 2^{31}" (see drand48 on page 286)

"nrand48 - generate pseudo random numbers between 0 and 2^{31} with initialization value" (see drand48 on page 286)

“rand - pseudo-random number generator (int)” on page 729

“seed48 - set seed (int) for pseudo-random numbers” on page 777" (see drand48 on page 286)

“srand - generate pseudo-random numbers with seed” on page 852

"srand48 - seed (double-precision) pseudo-random number generator" (see drand48 on page 286)

3.12 Search and sort procedures

“bsearch - conduct binary search of sorted array” on page 229

“hsearch, hcreate, hdestroy - manage hash tables” on page 529

"lfind - find entry in linear search table" (see lsearch on page 621)

“lsearch, lfind - linear search and update” on page 621

“qsort - sort table of data” on page 726

"tdelete - delete node from binary search tree" (see tsearch on page 940)

"tfind - find node in binary search tree" (see tsearch on page 940)

“tsearch, tfind, tdelete, twalk - process binary search trees” on page 940

3.13 Terminal interface and data transmissions

3.14 Database functions

3.15 List processing

“insque, remque - Insert element in queue or remove element from queue” on page 540

3.16 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 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.

4 Functions and variables in alphabetical order

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.

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 Syntax of the function call or variable declaration and of the header file in which the relevant interface is defined or declared.

A syntax line may be additionally identified as follows:

Optional
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. ❑

Description — Describes the functionality of the respective function, macro or external variable and explains the arguments to be specified.

Return val. — Lists and describes the possible return values of a function.

Some functions do not have a return value. In such cases and when describing external variables, the "Return value" section is omitted.

Errors Lists and describes the error codes stored in the external variable `errno` in the case of an invalid function call or an execution error.

Some functions do not store an error code in `errno` in the event of an error. In such cases and when describing external variables, the "Errors" section is omitted.

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, 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 val.

`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(l))` 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 146.

abs - return integer absolute value

Syntax

#include <stdlib.h>

int abs(int *i*);

Description `abs()` computes the absolute value of an integer *i*.

Return val. 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, faccessat - check access permissions for file

Syntax #include <unistd.h>

int access(const char *_path_, int _amode_);
int faccessat(int _fd_, const char *_path_, int _amode_, int _flag_);

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`.

The `faccessat()` function is equivalent to the `access()` function except when the _path_ parameter specifies a relative path. In this case the file whose access rights are to be checked is not searched for in the current directory, but in the directory connected with the file descriptor _fd_. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

In the _flag_ parameter, the value `AT_EACCESS`, which is defined in the `fnctl.h` header, can be transferred. In this case the effective user and group numbers are used for the check instead of the real ones.

When the value `AT_FDCWD` is transferred to the `faccessat()` function for the _fd_ parameter, the current directory is used.

Return val.

0	The requested access is permitted.
-1	The requested access is not permitted; `errno` is set to indicate the error.

Errors `access()` and `faccessat()` 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` *path* is an invalid address.

`EINTR` A signal was caught during the `access()` system call. ❑

`EINVAL` An attempt was made to access a BS2000 file.

`ELOOP` The maximum number of symbolic links `in` *path* was exceeded (or the maximum number of symbolic links is defined by `MAXSYMLINKS` in the header file `sys/param.h`).

`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.

In addition, `faccessat()` fails if the following applies:

`EACCES` The *fd* parameter was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

`ENOTDIR` The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

`EINVAL` The value of the *flag* parameter is invalid.

Note `access()` and `faccessat()` are executed for POSIX files only.

See also `chmod()`, `stat()`, `fcntl.h`, `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 val. arc cosine(*x*)

if successful. A floating-point number of type `double` in the range [0, π] 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` ≠ 0, an error has occurred.

See also `asin()`, `atan()`, `atan2()`, `cos()`, `sin()`, `tan()`, `math.h`.

acosh, asinh, atanh - inverse hyperbolic functions

Syntax

```
#include &lt;math.h&gt;
```

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 $\lvert x \rvert > 1.0$. `errno` is set to indicate the error.

Errors `acosh()` will fail if:

`EDOM`	$x < 1.0$.

`atanh()` will fail if:

`EDOM`	$\lvert x \rvert > 1.0$.

See also `cosh()`, `sinh()`, `tanh()`, `math.h`.

advance - pattern match given compiled regular expression

Syntax #include <regexp.h>

int advance(const char **string*, const char **exbuf*);

Description See `regexp()`.

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> ❑

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)`. ❑

Return val.

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). ❑

See also

`exec()`, `fork()`, `pause()`, `sigaction()`, `signal.h`, `unistd.h`, section “Signals” on page 146.

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 val. 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. No check is made here to see whether the time specification is meaningful, i.e. whether, for instance, the specified number of days fits the specified month. An error exists only when the data entered cannot be displayed in the time format. Consequently the earliest possible date which can be displayed is -999, and the latest date which can be displayed is 9999.

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 val. 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\0`

`EOVEFLOW` In case of an error `NULL` und `errno`.

Notes The `asctime()`, `ctime()`, `ctime64()`, `gmtime()`, `gmtime64()`, `localtime()` and `localtime64()` functions write their result into the same internal C data area. This means that each of these function calls overwrites the previous result of any of the other functions.

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 asctime_r(), clock(), 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 val. Pointer to the string that *buf* points to
if successful.

`EOVEFLOW` In case of an error `NULL` und `errno`.

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 val.

arc sine(*x*)	if successful. A floating-point number of type `double` in the range [-π/2, +π/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` ≠ 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 manuals "C Compiler" [3] and "C/C++ Compiler" [4])

– 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 val. arc tangent(*x*)

if successful. A floating-point number of type `double` in the range [-π/2, +π/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 the calculated.

y is the divisor of the expression for which the arc tangent is to the calculated.

Return val. arc tangent(*x*/*y*)

if neither argument is 0.0.
A floating-point number of type `double` in the range [-π/2, +π/2] is returned.

-π/2 or +π/2

if the divisor is 0.0, depending on the sign of the dividend.

0 if the dividend is 0.0.

π/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 val.

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:

$$[\textit{tab}\ldots]\,[\left\{\begin{matrix}+\\-\end{matrix}\right\}]\,[\textit{digit}\ldots]\,[.]\,[\textit{digit}\ldots]\,[\left\{\begin{matrix}E\\e\end{matrix}\right\}\,[\left\{\begin{matrix}+\\-\end{matrix}\right\}]\,\textit{digit}\ldots]$$

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 val.

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:

$$[tab\ \ldots][\left\{\begin{matrix}+\\-\end{matrix}\right\}]digit\ldots$$

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 val. 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*...] [{+|-}] *digit*...

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 val. 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

$$[\,tab\ldots\,][\left\{\begin{matrix}+\\-\end{matrix}\right\}]\,digit\ldots$$

All characters that produce white space are legal for *tab* (see the definition under `isspace()`).

Return val.

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 `strtoll(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

<libgen.h> is included as follows:

```
#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 val.

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 val.

0	All *n* bytes are the same, or *n*=0.
≠ 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!):

bcopy(*s1*, *s2*, *n*) ≅ 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 val. `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`.

bs2cmd - execute BS2000 commands by means of the CMD macro

Syntax

#include <bs2cmd.h>

int bs2cmd(const char *cmd, bs2cmd_rc *rc, int maxoutput, int flag
[, int *outbuflen, char *outbuf [, int *errbuflen, char *errbuf]]);

`bs2cmd()` can be used to execute a BS2000 command by means of the BS2000 CMD macro. Only commands for which the CMD macro is permissible can be used. In particular, it makes no sense to execute commands that lead to the unloading of the calling program, since the interface does not include any precautionary features that prevent this.

The command outputs can be buffered optionally. In this case the interface can also be used by an rlogin task without a SYSFILE environment.

Parameters

const char *cmd

This parameter contains the command to be executed or a list of commands separated by semicolons. Except for strings enclosed in apostrophes, all characters are converted to uppercase letters in *cmd* before the call.

bs2cmd_rc *rc

rc is a pointer to the structure `bs2cmd_rc`, which contains return information.

`bs2cmd_rc` is structured as follows:

```
typedef struct bs2cmd rc {
    unsigned char  subcode2;
    unsigned char  subcode1;
    unsigned short maincode;
    unsigned short progrc;
    char cmdmsg[8];
  } bs2cmd rc;
```

If the NULL pointer is passed when `bs2cmd` is called with *rc*, no return information is made available.

int maxoutput

This parameter specifies the size of the buffer to be created for command output in bytes. When setting the buffer size you must take into account that administration information is also output in addition to the command output itself.

The following constants can be specified:

BS2CMD_DEFAULT

A standard buffer of 256 KB is used.

BS2CMD_NOBUFFER
Output is not buffered. With this setting, commands that generate output can only be executed under rlogin tasks if the user provides a buffer (specification of BS2CMD_FLAG_USER_BUFFER in the parameter *flag*).

If the buffer is set too small for the pending output, command execution is aborted.

int flag
This parameter specifies the interface configuration flags. The following flags and flag combinations (linked with "|") can currently be specified:

BS2CMD_FLAG_STRIP
The print control characters in the command output are removed before output is made.

BS2CMD_FLAG_SPLIT
The command outputs are split between stdout and stderr. Messages are output to stderr.

BS2CMD_FLAG_USER_BUFFER
`bs2cmd` is called with a variable parameter list. The parameters of the variable parameter list are then evaluated. These parameters must be specified completely, otherwise the behaviour of the `bs2cmd` function is undefined.

Parameters of the variable parameter list:

The following parameters allow command outputs to be sent to a memory area provided by the user if BS2CMD_FLAG_USER_BUFFER is set in the parameter *flag*.

int *outbuflen
Length of the memory area for stdout outputs. After `bs2cmd` is executed, *outbuflen* contains the number of bytes actually written to `outbuf`, or -1 if `outbuf` is set too small for the output.

char *outbuf
Address of the memory area for stdout outputs.

int *errbuflen

Length of the memory area for stderr outputs. After `bs2cmd` is executed, *errbuflen* contains the number of bytes actually written to `errbuf`, or -1 if `errbuf` was set too small for the output.
**errbuflen* is only relevant if BS2CMD_FLAG_SPLIT is set in the parameter *flag*.

char *errbuf

address of the memory area for stderr outputs. **errbuf* is only relevant if the BS2CMD_FLAG_SPLIT is set in the parameter *flag*.

Notes

The messages are written into the memory area passed by the user and terminated with \n. Depending on the values specified in the parameter *flag*, the messages are either only written to `outbuf` or split over `outbuf` and `errbuf`, either with or without print control characters in each case.
If the size of the memory area is big enough for the pending data, the output is terminated with \0.
The \0 byte is not included in the returned length.

If the size of the memory area is too small for the pending data, the value -1 is returned and EFBIG is set in `errno`. To discriminate between whether one of the user memory areas or the internal buffer is too small, the value -1 is entered in `outbuflen` or `errbuflen` if `outbuf` or `errbuf` is too small.

If the value BS2CMD_NOBUFFER is specified for *maxoutput* and the value BS2CMD_FLAG_USER_BUFFER is simultaneously set for *flag*, no internal buffering is used and command outputs are sent directly to the buffer `outbuf` provided by the user. The structure of the outputs to `outbuf` is described in the "Macro Calls to the Runtime Section" manual.

Caution!

In the case described, the address of the memory area must be aligned to word boundaries, otherwise `errno` is set to EFAULT.

If no buffering is used, the flag values BS2CMD_FLAG_STRIP and BS2CMD_FLAG_SPLIT are not evaluated. Specifying these values is ignored.

Return val. `maincode` If the command is executed successfully, `errno` is not set.

-1 In the event of an error, `errno` is set to one of the following values:

EINVAL
: One of the arguments has an impermissible value (e.g. an empty command or a negative buffer size).

ENOMEM
: There is not enough memory available for the buffers to be created.

EFAULT
: After the command is executed, the contents of the output buffer cannot be interpreted or there is an `outbuf` alignment error.

EFBIG
: The output buffer is not large enough for the outputs.

In the event of an error, the contents of the user buffer are undefined.

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 `bf2stat` at each function call. The function calls are made automatically by `bs2fstat` (in a `while` loop) .

Return val.
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 val.

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 ≠ 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 "Executive Macros" [10]). 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 val.

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 val. 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 val. 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`:

```
#typdef struct{double x, y;} _ _complex
```

Return val. 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 val. 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 val.

0	if successful.
-1	if unsuccessful. `errno` is set to indicate the error.

Errors
`catclose()` will fail if:

`EBADF`	The catalog descriptor is not valid.
`EINTR`	`catclose()` is interrupted by a signal.

See also
`catgets()`, `catopen()`, `nl_types.h`, section “Locale” on page 86.

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 val.

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 86.

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 86).

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 104).

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 val. 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 86 and section “Environment variables” on page 104.

cbrt - cube root

Syntax #include <math.h>

double cbrt (double *x*);

Description `cbrt()` returns the cube root of *x*.

Return val. 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 152 and in the manual "Executive Macros" [10].

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()`:

`secind` "Secondary Indicator", as stored in the most significant byte of register 15 (values 4 or 20) after execution of the `ENACO` macro.

`rcode1` "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 val. 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` ≠ 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 152 and the manual "Executive Macros" [10].

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:

`coname` 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 `strfill()` function, for example, is suitable for supplying `coname` This input is mandatory.

`level` Priority level of the contingency process. This input is mandatory. Values from 1 - 126 are legal.

`econt` Start address of the contingency routine. This input is mandatory.

`comess` Contingency message. This input is optional. The value is passed to the contingency routine as a parameter.

Entries supplied by `cenaco()`:

`coidret` Short ID of the contingency process. This short ID must be used in further macros (e.g. `SOLSIG`) for the identification of the contingency process.

`secind` "Secondary Indicator", as stored in the most significant byte of register 15 (values 4 or 20) after execution of the `ENACO` macro.

`rcode1` "Return Code", as stored in the least significant byte of register 15 (value 0 or 4) after execution of the `ENACO` 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 val. Input baud rate of type `speed_t`
if successful.

See also `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`, `tcgetattr()`, `termios.h`, section “General terminal interface” on page 129.

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 val. 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 129.

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 val.

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 129.

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 val. 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 129.

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 val.

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, fchmodat - change mode of file

Syntax

#include <sys/stat.h>

Optional
#include <sys/types.h> ❑

int chmod(const char **path*, mode_t *mode*);
int fchmodat(int *fd*, const char **path*, mode_t *mode*, int *flag*);

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.

The `fchmodat()` function is equivalent to the `chmod()` function except when the *path* parameter specifies a relative path. In this case the file to be updated is not searched in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

In the *flag* parameter, the value `AT_SYMLINK_NOFOLLOW`, which is defined in the `fnctl.h` header, can be transferred. If *path* specifies a symbolic link, the symbolic link is updated.

When the value `AT_FDCWD` is transferred to the `fchmodat()` function for the *fd* parameter, the current directory is used.

Return val.

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 `errno` is set to indicate the error.

Errors
`chmod` and `fchmodat()` 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 *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 file that does not exist or to an empty string.

`ENOTDIR` A component of *path* 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.

In addition, `fchmodat()` fails if the following applies:

`EACCES` The *fd* parameter was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

`ENOTDIR` The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

`EINVAL` The value of the *flag* parameter is invalid.

Notes
`chmod()` and `fchmodat()` are executed only for POSIX files

See also
`chown()`, `fchmod()`, `mkdir()`, `mkfifo()`, `open()`, `stat()`, `fcntl.h`, `sys/types.h`, `sys/stat.h`.

chown, fchownat - 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*);
int fchownat(int *fd*, const char **path*, uid_t *owner*, gid_t *group*, int *flag*);

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.

The `fchownat()` function is equivalent to the `chown()` or `lchown()` function except when the *path* parameter specifies a relative path. In this case the file whose user and group numbers are to be updated is not searched for in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

In the *flag* parameter, the value `AT_SYMLINK_NOFOLLOW`, which is defined in the `fnctl.h` header, can be transferred. If *path* specifies a symbolic link, the user and group numbers of the symbolic link are updated.

When the value `AT_FDCWD` is transferred to the `fchownat()` function for the *fd* parameter, the current directory is used.

Return val. 0 if successful.

-1 if an error occurs; `errno` is set to indicate the error.

Errors `chown()` and `fchownat()` 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.

`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 *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 file that does not exist or to an empty string.

`ENOTDIR` A component of *path* 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.

In addition, `fchownat()` fails if the following applies:

`EACCES` The *fd* parameter was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

`ENOTDIR` The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory, or the *flag* parameter has the value `AT_REMOVEDIR`, and *path* does not specify a directory.

`EINVAL` The value of the *flag* parameter is invalid.

Notes `chown()` and `fchownat()` are executed only for POSIX files

See also `chmod()`, `fcntl.h`, `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 val. 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 49) 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 val. 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 49.

clock_gettime, clock_gettime64 - get time of a specified clock

Syntax

```
#include <time.h>

int clock_gettime(clockid_t clk_id,
                  struct timespec *tp);
int clock_gettime64(clockid_t clk_id,
                  struct timespec64 *tp);
```

Description

In the structure which points to *tp*, `clock_gettime()` and `clock_gettime64()` supply the time of the clock that is pecified by *clk_id* as the number of seconds and milliseconds which have passed since the reference date (epoch). Only the system-wide real time clock CLOCK_REALTIME is supported. It supplies the number of seconds and nanoseconds which have passed since the reference date (epoch).

The reference date is 1/1/1970 00:00:00.

Return val.

0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

Error

`clock_gettime()` will fail, if:

`EINVAL` the specified *clk_id* is not supported.

See also

gettimeofday

close - close file

Syntax #include <unistd.h>

int close(int *fildes*) ;

Description *fildes* is a file descriptor obtained from a `creat()`, `open()`, `dup()`, `fcntl` or `pipe()` system call. `close()` closes the file descriptor indicated by *fildes*. All existing record locks owned by the process on the file specified with *fildes* 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 *fildes* 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 *fildes* 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 val.

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`	*fildes* 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 val. 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

By default `syslog()` writes a message *message* to the `/var/adm/messages` file. Optionally the system administrator can also define different log files for the syslog daemon. Details of how the syslog daemon works and is controlled are provided in the "POSIX Basics" manual [1].

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 log file (by default: `/var/adm/syslog)` . This option can be used safely in processes which do not have a control terminal, because `syslog()` generates a child process before the console is opened.
`LOG_NDELAY`	Opens the log file (by default: `/var/adm/syslog)` on execution of `openlog()`. 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 `syslog()` 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 `syslog()` 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 val. `setlogmask()`:

Previous log priority mask.

See also `printf()`, `syslog.h`.

compile - produce compiled regular expression

Syntax
#include <regexp.h>

int compile(char **instring*, char **exbuf*, const char **endbuf*, int *eof*);

Description
See `regexp()`.

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 val.
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 val. 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 val. 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 val. CPU time used in ten thousandths of a second.

See also `clock()`, `stdlib.h`, section “Scope of the supported C library” on page 49.

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*);
int creat64(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:
Only the *lbp* switch, the Nosplit switch, and the `O_RECORD` specification are evaluated in this parameter. All other specifications in this parameter are ignored. However, it is required for the creation of portable programs since it controls the protection bit assignment in the UNIX operating system.

lbp switch

The *lbp* switch controls handling of the Last Byte Pointer (LBP). It is only relevant for binary files with PAM access mode and can be combined with all specifications permissible for `open`. If `O_LBP` is specified as the *lbp* switch, a check is made to see whether LBP support is possible. If this is not the case, the `creat()`, `creat64()` function will fail and `errno` is set to `ENOSYS`. The switch has further effects only when the file is closed.

`O_LBP`
: When a file which has been newly created is closed, no marker is written and a valid LBP is set.
In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

`O_NOLBP`
: When a file which has been newly created is closed, the LBP is set to zero (= invalid). A marker is written. In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.
In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

If the *lbp* switch is specified in both variants (`O_LBP` and `O_NOLBP`), the `creat()`, `creat64()` function fails and `errno` is set to `EINVAL`.

If the *lbp* switch is not specified, the behavior depends on the environment variable LAST_BYTE_POINTER (see also section "Environment variables" on page 104):

`LAST_BYTE_POINTER=YES`

The function behaves as if `O_LBP` were specified.

`LAST_BYTE_POINTER=NO`

The function behaves as if `O_NOLBP` were specified.

Nosplit switch

This switch controls the processing of text files with SAM access mode and variable record length when a maximum record length is also specified. It can be combined with any of the other constants.

`O_NOSPLIT`
When writing with `write()`, records which are longer than the maximum record length are truncated to the maximum record length.

If the switch is not specified, the following applies when writing:
A record which is longer than the maximum record length will be split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it.

The constant `O_RECORD` can be specified in the *modus* parameter to open files with record-oriented input/output (record I/O). It can always be combined with every other constant except `O_LBP`.

`O_RECORD`

This switch functions as follows:

The `write` function writes a record to the file. In the case of SAM and PAM files the record is written to the current file position. In the case of ISAM files the record is written to the position which corresponds to the key value in the record. If the number *n* of the characters to be written is greater than the maximum record length, only a record with the maximum record length is written. The remaining data is lost. In the case of ISAM files a record is written only if it contains at least a complete key. If in the case of files with a fixed record length *n* is less than the record length, binary zeros are used for padding. When a record is updated in a SAM or PAM file, the length of the record may not be modified. The `write` function returns the number of actually written characters also in the case of record I/O.

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 val. 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 val. 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 stxitp *stxitpar*);

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 152 and in the manual "Executive Macros" [10].

The structure of `stxit` is defined in `stxit.h` as follows:

```
struct stxitp
{

  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. .

`old_stx` 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.

`new_stx` 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 `contp`.

`del_stx` 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:

`ful_diag`, `min_diag`, `no_diag` 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.

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 val. 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, ctime64 - convert date and time to string

Syntax

```
#include <time.h>

char  *ctime(const time_t  *clock);
char *ctime64(const time64_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*`))`.

Calling `ctime64()` has the same effect as `asctime64(localtime(`*clock*`))`, the latest and earliest displayable dates being 31.12.9999 23:59:59 hrs. local time and 1.1.1900 00:00:00 hrs.

`ctime()` is not thread-safe. Use the reentrant function `ctime_r()` when needed.

Return val.

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. `Thu Jun 14 15:20:54 2018\n\0`

`EOVEFLOW`
: In case of an error `NULL` und `errno`.

Notes

The `asctime()`, `ctime()`, `ctime64()`, `gmtime()`, `gmtime64()`, `localtime()` and `localtime64()` functions write their result into the same internal C data area. This means that each of these function calls overwrites the previous result of any of the other functions.

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 val. 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 val.
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" [1].

_ _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 689), 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 val. `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, difftime64 - compute difference between two calendar time values

Syntax #include <time.h>

double difftime(time_t *time1*, time_t *time0*);
double difftime(time64_t *time1*, time64_t *time0*);

Description `difftime()` and `difftime64()` compute the difference between two calendar times.

time1 and *time0* are time values of type `time_t` or `time64_t`. These time values are supplied by the `mktime()`, `mktime64()` and `time()`, `time64()` functions.

Return val. *time1* - *time0* if successful. The time difference is indicated in seconds and is of type `double`.

See also `ctime()`, `mktime()`, `time()`, `time.h`.

dirfd - extract file descriptor

Syntax #include <dirent.h>
int dirfd(DIR **dirp*);

Description The function `dirfd()` extracts the file descriptor from the DIR object to which `dirp` points. If an attempt is made to close or modify the file descriptor with functions other than `closedir()`, `readdir()`, `readdir_r()`, `rewinddir()` or `seekdir()`, the behaviour is undefined.

Return val. File descriptor of the DIR object
if successful.

-1 if an error occurs. `errno` is set to indicate the error.

Errors `dirfd()` fails, if:

`EINVAL` *dirp* does not point to an open directory stream.

See also `closedir()`, `readdir()`, `readdir_r()`, `rewinddir()`, `seekdir()`

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 val.

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 = strup(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:

quotient * *divisor* + *remainder* = *dividend*

Return val. 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, *Xi*, according to the linear congruential formula:

$X_{n+1} = (aX_n + c)$ mod m $\qquad n \geq 0$

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 *Xi* 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 *Xi* and transformed into the returned value.

The `drand48()`, `lrand48()` and `mrand48()` functions store the last 48-bit *Xi* 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 *Xi* 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 *Xi* 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 Xi to the value of the `{LONG_BIT}` bits contained in its argument. The low-order 16 bits of Xi are set to the arbitrary value $330E_{16}$.

The initializer function `seed48()` sets the value of *Xi* to the 48-bit value specified in the argument array. In addition, the previous value of *Xi* 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 *Xi* 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 *Xi*, the multiplier value *a*, and the addend value *c*. Argument array elements *param[0]* to *param[2]* specify *Xi*, *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 val. 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 val.

Non-negative integer (the file descriptor)
if successful.

-1 if an error occurs; `errno` is set to indicate the error.

Errors `dup()` and `dup2()` will fail if:

`EBADF` *fildes* is not a valid open file descriptor or the argument *fildes2* is negative or greater than or equal to {`OPEN_MAX`}.

`EINTR` `dup2()` was interrupted by a signal.

`Extension`
`EINVAL` *fildes* and *fildes2* designate BS2000 files. ❑

`EMFILE` The number of file descriptors in use by the process would exceed {`OPEN_MAX`}, or no *fildes2* file descriptors are available.

Notes `dup()` and `dup2()` are executed only for POSIX files.

See also `close()`, `fcntl()`, `open()`, `unistd.h`.

ebcdic_to_ascii - convert EBCDIC string to ASCII string *(extension)*

Syntax int ebcdic_to_ascii(char **in*, char **out*);

Description `ebcdic_to_ascii` converts EBCDIC strings to ASCII strings, where *in* is the input string in EBCDIC code, and *out* 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 val.

0	if successful.
1	if an error occurs.

See also `ascii_to_ebcdic`.

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 val. `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 val.

`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 val.

`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 `endpwent()` will fail if:

`EACCES` the user ID (uid) of the caller is invalid.

`getpwent()`, `setpwent()` and `endpwent() will fail if:`

`EFAULT` errors occur during creation of the `passwd` structure.

`ENOENT` the user does not exists.

Notes The return value of `getpwent()` can point to an area that will be overwritten by a subsequent call of `getpwuid()`, `getpwnam()` or `getpwent()`.

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" [1]).

These functions are only supported for reasons of compatibility.

The characteristics of a current process can be defined as follows:

– `getpwuid(geteuid())` returns the name of the effective user ID of the process
– `getlogin()` returns the login name of the process
– `getpwuid(getuid())` returns the name of the real user ID of the process.

If error situations are to be investigated, `errno` must be set to 0 before `getpwent()` is called.

See also `endgrent()`, `getlogin()`, `getpwnam()`, `getpwuid()`, `putpwent()`, `pwd.h`, manual "POSIX Basics" [1].

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 ut_type	Other components
EMPTY	No further data
BOOT_TIME	ut_tv
OLD_TIME	ut_tv
NEW_TIME	ut_tv
USER_PROCESS	ut_id, ut_user (login name), ut_line, ut_pid, ut_tv
INIT_PROCESS	ut_id, ut_pid, ut_tv
LOGIN_PROCESS	ut_id, ut_user (implementation-specific name of the login process), ut_pid, ut_tv
DEAD_PROCESS	ut_id, ut_pid, ut_tv

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 val. `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 104).

Notes The `environ` array should not be directly accessed by the application.

See also `exec`, `getenv()`, `putenv()`, `setenv()`, `unsetenv()`, section “Environment variables” on page 104.

epoll_create - create an epoll instance

Syntax
#include <sys/epoll.h>
int epoll_create (int *size*)

Description
The epoll functions are a scalable I/O event notification mechanism and thus an alternative to the present POSIX functions `select()` and `poll()`.

`epoll_create()` creates an `epoll` instance and returns a file descriptor referring to the new `epoll` instance. This file descriptor is used for all the subsequent calls to the `epoll` interface. When no longer required, the file descriptor returned by `epoll_create()` should be closed by using `close()`.

The `size` argument is ignored, but must be greater than zero.

Return val.
File descriptor
if successful.

-1 if an error occurs; `errno` is set to indicate the error.

Errors
`epoll_create()` will fail if:

`EINVAL` `size` is not positive.

`ENFILE` The system-wide limit on the total number of open files has been reached.

`ENOMEM` There was insufficient memory to create the kernel object .

See also
`epoll_ctl()`, `epoll_wait()`

epoll_ctl - control epoll instance

Syntax

#include <sys/epoll.h>
int epoll_ctl (int *epfd*, int *op*, int *fd*, struct epoll_event **event*)

Description

This system call performs control operations on the `epoll` instance `epfd`. It requests that the operation `op` be performed for the target file descriptor, `fd`.

Parameters

int op

Valid values for the `op` argument are :

EPOLL_CTL_ADD
Register the target file descriptor `fd` on the `epoll` instance referred to by the file descriptor `epfd` and associate the event `event` with the internal file linked to `fd`.

EPOLL_CTL_MOD
Change the event `event` associated with the target file descriptor `fd`.

EPOLL_CTL_DEL
Remove (deregister) the target file descriptor `fd` from the `epoll` instance referred to by `epfd`. The `event` argument is ignored and can be NULL.

struct epoll_event *event

The `event` argument describes the events to be monitored for file descriptor `fd` as well as application specific data, which are to be returned if one of the events occurs.

The *struct e poll_event* is defined as:

```
typedef union epoll_data {
   void *ptr;
   int fd;
   uint32_t u32;
   uint64_t u64;
} epoll_data_t;

struct epoll_event {
   uint32_t events; /* Epoll events */
   epoll_data_t data; /* User data variable */
};
```

The `data` member can be supplied with application specific data, which contain additional information, e.g. on the file descriptor.

The `events` member is a bit mask composed using the following available event types:

`EPOLLIN`
: Data other than high-priority data may be read without blocking. For STREAMS, this flag is set in `events` even if the message is of zero length.

`EPOLLPRI`
: Data other than high-priority data may be read without blocking. For STREAMS, this flag is set in `events` even if the message is of zero length.

`EPOLLOUT`
: Normal data (priority band equals 0) may be written without blocking.

`EPOLLERR`
: An error has occurred on the device or stream. The function `epoll_wait()` will always wait for this event; it is not necessary to set it in `events` for the `epoll_ctl()` function.

`EPOLLHUP`
: A hangup has occured in the stream. The device has been disconnected. `EPOLLHUP` and `EPOLLOUT` are mutually exclusive; a stream can never be writable if a hangup has occurred. However, this event and `EPOLLIN`, `EPOLLRDNORM`, `EPOLLRDBAND` or `EPOLLPRI` are not mutually exclusive. The function `epoll_wait()` will always wait for this event; it is not necessary to set `EPOLLHUP` in `events` for the `epoll_ctl()` function.

`EPOLLRDNORM`
: Normal data (priority band equals 0) may be read without blocking. For STREAMS, this flag is set in `events` even if the message is of zero length.

`EPOLLWRNORM`
: as `EPOLLOUT`.

`EPOLLRDBAND`
: Data from a non-zero priority band may be read without blocking. For STREAMS, this flag is set in `events` even if the message is of zero length.

`EPOLLWRBAND`
: Priority data (priority band > 0) may be written.

`EPOLLRDHUP`
: as `EPOLLHUP`.

`EPOLLET`
: This functionality is not supported in POSIX.

Return val.	Null	if successful.
	-1	if an error occurs; `errno` is set to indicate the error.
Errors	`epoll_ctl()`	will fail if:
	`EBADF`	`epfd` or `fd` is not a valid file descriptor.
	`ENOENT`	`op` was EPOLL_CTL_MOD or EPOLL_CTL_DEL, and fd is not registered with the `epoll` instance `epfd`.
	`EEXIST`	op was EPOLL_CTL_ADD, and the supplied file descriptor `fd` is already registered with the `epoll` instance `epfd`.
	`EINVAL`	epfd is not an `epoll` file desriptor, or `fd` is the same as `epfd`, or the requested operation `op` is not supported by this interface.
See also	`epoll_create()`, `epoll_wait()`	

epoll_wait - wait for events (epoll instance)

Syntax

#include <sys/epoll.h>
int epoll_wait (int *epfd*, struct epoll_event **events*, int *maxevents*, int *timeout*)

Description

The `epoll_wait()` system call waits for events on the `epoll` instance referred to by the file descriptor `epfd`. The memory area pointed to by `events` will contain the events that will be available for the caller. It must be an array of `struct epoll_event` structures and the number of array members must be specified in `maxevents`. The `maxevents` argument must be greater than zero. For each file descriptor for that an event occured, the `epoll_wait()` system call provides a structure in that array. The system call return events for up to `maxevents` file descriptors.

The call will block until one of the following events occurs:

– a file descriptor delivers an event
– the call is intrrupte by a signal
or
– the time specified by `timeout` expires.

The `timeout` argument specifies the number of milliseconds that `epoll_wait()` will block.

Specifying a timeout of -1 causes `epoll_wait()` to block indefinitely, while specifying a `timeout` equal to zero causes `epoll_wait()` to return immediately , even if no events are available.

The data of each returned structure will contain the same data the user set with an `epoll_ctl()` while the `events` member will contain the returned event bit field.

Return val.

Number of file descriptors ready for the requested I/O
if successful.

zero — if no file descriptor became ready until time spezified by `timeout` expires.

-1 — if an error occurs; `errno` is set to indicate the error.

Errors

`epoll_wait()` will fail if:

`EBADF` — `epfd` is not a valid file descriptor.

`EINVAL` — `epfd` is not an `epoll` file descriptor,or `maxevents` is less than or equal zero.

`EFAULT` — The memory area pointed to by `events` is not accessible with write permissions.

`EINTR` — The call was interrupted by a signal handler bevor either any of the requested events occurred or the `timeout` expired

See also `epoll_create()`, `epoll_ctl()`

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 - erf(`x`)`.

Return val. 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 161).

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 161.

exec: execl, execv, execle, execve, execlp, execvp - execute file

Syntax

#include <unistd.h>
extern char **environ;

int execl (const char *path, const char *arg0, ... , (char *)0);
int execv (const char *path, char *const argv[ ]);
int execle (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 104). The environment is typically supplied by the POSIX shell (see also the manual "POSIX Basics" [1]). Other X/Open-compatible systems may define alternate mechanisms for this purpose.

If the process image file is not a valid executable object, `execlp()` 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 (`execl()`, `execv()`, `execlp()` 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 val. -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`*nn* (where 00 <= *nn* <= 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 104.

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()` f`unction` 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 `MAINCODE`, 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 ≠ 0 and ≠ 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 val. 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 val. 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`.

faccessat - check access permissions for file

Syntax #include <unistd.h>

int faccessat(int *fd*, const char **path*, int *amode*, int *flag*);

Description See `access()`.

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 val. 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 val.

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` *fildes* 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` *fildes* 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 *fildes*);

Description

Like `chdir()`, `fchdir()` also changes the current directory. The new directory is identified by the file descriptor *fildes*. The current directory is the starting point for the search for pathnames which do not begin with “/“. The *fildes* 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 val.

0	if successful.
-1	if an error occurs. The current working directory remains unchanged. `errno` is set to indicate the error.

Errors

`fchdir()` will fail if:

`EACCES`	There is no search permission for *fildes*.
`EBADF`	*fildes* 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 `fchdir()` system call.
`EIO`	An I/O error occurred during reading or writing from the file system.
`ENOLINK`	*fildes* 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 *fildes*, 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 *fildes*. The file mode bits are interpreted as follows (see also `sys/stat.h`):

Symbolic name	Bit pattern	Meaning
S_ISUID	04000	Set user ID on execution
S_ISGID	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
S_ISVTX	01000	Save text segment after execution
S_IRWXU	00700	Read, write or execute (search) by owner
S_IRUSR	00400	Read by owner
S_IWUSR	00200	Write by owner
S_IXUSR	00100	Execute (search if a directory) by owner
S_IRWXG	00070	Read, write or execute (search) by group
S_IRGRP	00040	Read by group
S_IWGRP	00020	Write by group
S_IXGRP	00010	Execute by group
S_IRWXO	00007	Read, write or execute (search) by others
S_IROTH	00004	Read by others
S_IWOTH	00002	Write by others
S_IXOTH	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 val.

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.`

fchmodat - change mode of file

Syntax #include <sys/stat.h>

Optional
#include <sys/types.h> ❑

int fchmodat(int *fd*, const char **path*, mode_t *mode*, int *flag*);

Description See `chmod()`.

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 val.

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 `errno` is set to indicate the error.

Errors `fchown()` will fail if:

`EABDF`	*fildes* 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.
	group or *owner* 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`.

fchownat - change owner and group of file

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

int fchownat(int *fd*, const char **path*, uid_t *owner*, gid_t *group*, int *flag*);

Description See `chown()`.

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 val.

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

Syntax

#include <fcntl.h>

Optional
#include <sys/types.h>
#include <unistd.h>

int fcntl(int *fildes*, int *cmd*, ... / * *arg* */);

Description

`fcntl()` provides for control over open files.

fildes 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:

`F_DUPFD` Returns a new file descriptor with the following characteristics:

- Lowest numbered available (i.e. open) file descriptor greater than or equal to the integer value passed as the third argument (*arg*).
- Same open file (or pipe) as the original file.
- Same file position indicator as the original file (i.e. both file descriptors share one file position indicator).
- Same access mode (read, write, or read/write) as the original file.
- Same file status bits as the original file.
- The close-on-exec flag (see `F_GETFD`) associated with the new file descriptor is set to remain open across `exec` system calls.

`F_GETFD` Gets the close-on-exec flag associated with file descriptor *fildes*. If the low-order bit is 0, the file will remain open across `exec`. Otherwise, the file will be closed upon execution of `exec`.

`F_SETFD` Sets the close-on-exec flag associated with *fildes* to the low-order bit of the integer value given as the third argument (`0` or `1` as above).

`F_GETFL` Gets the *fildes* status flag.

`F_SETFL` Sets the *fildes* status flag to the integer value given as the third argument. Only certain flags can be set (see `fcntl()`).

Extension

F_FREESP
: Frees storage space associated with a section of the ordinary file *fildes*. 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 val.

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` *cmd* is `F_FREESP`, the file exists, mandatory file/record locking is set, and there are outstanding record locks on the file.

Extension

EAGAIN *cmd* is F_SETLK or F_SETLKW, and the file is currently being mapped to virtual memory using mmap() (see mmap()).

EBADF *fildes* is not a valid open file descriptor.

cmd is F_SETLK or F_SETLKW, the type of lock (l_type) is a write lock (F_WRLCK), and *fildes* is not a valid file descriptor open for reading.

cmd is F_SETLK or F_SETLKW, the type of lock (l_type) is a write lock (F_WRLCK), and *fildes* is not a valid file descriptor open for writing.

cmd is F_FREESP, and *fildes* is not a valid file descriptor open for writing.

Extension

EDEADLK *cmd* is F_FREESP, mandatory record locking is enabled, O_NDELAY and O_NONBLOCK are clear, and a deadlock condition was detected.

EDEADLK *cmd* 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.

Extension

EFAULT *cmd* is F_FREESP, and the value pointed to by *arg* is located at an address outside the address space used by the process.

cmd is F_GETLK, F_SET_LK or F_SETLKW, and the value pointed to by *arg* is located at an address outside the address space used by the process.

EINTR A signal was caught during the fcntl() system call.

EINVAL *cmd* is F_DUPFD, and *arg* is either negative or greater than or equal to the value for the maximum number of open file descriptors permitted for each user.

cmd is not a valid value.

cmd is F_GETLK, F_SETLK or SETLKW, and *arg* or the data it points to is not valid, or *fildes* refers to a file that does not support locking.

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.

EMFILE *cmd* 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	*fildes* 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.
EOVERFLOW	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, unistdt.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 val.

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 val.

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 *fildes*, const char **mode*);

Description

`fdopen()` associates a stream with a file descriptor.

mode a character string having one of the following values:

`r` or `rb`	open a file for reading
`w` or `wb`	open a file for writing
`a` or `ab`	open a file for writing at end of file
`r+`, `r+b` or `rb+`	open a file for update (reading and writing)
`w+`, `w+b` or `wb+`	open a file for update (reading and writing)
`a+` , `a+b` or `ab+`	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 val.

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` *fildes* 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 107.

fdopendir - open directory

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

int fchownat(int *fd*, const char **path*, uid_t *owner*, gid_t *group*, int *flag*);

Description See `opendir()`.

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 val. ≠ 0 `EOF` is set for *stream*; the end of file was reached.

0 `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 val. ≠ 0 if the error indicator is set for *stream*.

0 if the error indicator is not set for *stream*.

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 val.

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 val.

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).

`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 val.
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 following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

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 val. 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 val. 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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

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 val.

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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

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()`.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

Return val.

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 val. `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 112.

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:

_KEY_FIRST	Sets the file position indicator to beginning of file. The *key* and *keylen* parameters are ignored. Positioning works even if the file is empty.
_KEY_LAST	Sets the file position indicator to end of file. The *key* and *keylen* parameters are ignored. Positioning works even if the file is empty.
_KEY_EQ	Sets the file position indicator to the first record with the specified *key*.
_KEY_GE	Sets the file position indicator to the first record with a key value greater than or equal to the specified *key*.

Return val.

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 val.

`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 val. 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 val. 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()`, `ceill()`, `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 `gettxt()`, `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 `MM_NULLMC`
: indicates that no classification component is specified for the message.

label 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 *label*. For example, the content `UX:cat` for *label* indicates that the package UNIX system V and the application `cat` are meant.

severity indicates the severity level of the status. Identifiers for the severity levels are:

`MM_HALT` indicates that the application has come across a critical error and processing is being halted. The string “`HALT`“ is output.

`MM_ERROR` indicates that the application has detected an error. The string “`ERROR`“ is output.

`MM_WARNING`
: indicates that an unusual state has arisen which could involve a problem that needs monitoring. The string “`WARNING`“ is issued.

`MM_INFO` provides information on a state which does not represent an error. The string “`INFO`“ is output.

`MM_NOSEV` indicates that no severity level exists for the message.

text describes the cause of the message. The *text* string is not limited to a particular length. If the string is empty, the text that is output is undefined.

action describes the first action to be executed in the error recovery process. `fmtmsg()` writes the prefix “`TO FIX:`“ before this string. The *action* string is not limited to a particular length.

tag An identifier that refers to the online documentation for the message. It is recommended that *tag* contain the origin of the message addressed via *label* and a unique number. An example of *tag* is `UX:cat:146`.

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 c`omponent 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 val.

`MM_OK` if successful.

`MM_NOTOK` The function has completely failed.

`MM_NOMSG` The function could not generate a message via the standard error output, but was otherwise successful.

`MM_NOCON` 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
label	char*	(char*) NULL	MM_NULLLBL
severity	int	0	MM_NULLSEV
class	long	0L	MM_NULLMC
text	char*	(char*) NULL	MM_NULLTXT
action	char*	(char*) NULL	MM_NULLACT
tag	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
```

Beispiel 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
  - (SYSTERM), 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.

`w+r`, `w+` 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.

`a+r`, `a+` 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.

`r+b`, `rb+` Open binary file for reading and writing. The file must already exist. The old contents are preserved.

`w+b`, `wb+` 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.

`a+b`, `ab+` 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:
In *mode* optionally further functions may be controlled by additional specifications:

Additional specification	Function
tabexp=yes/no	Handling of the tab character (\t)
lbp=yes/no	Handling of the Last Byte Pointers (LBP)
split=yes/no	Processing text files with specification of a maximum record length

Tab character (\t)

Additionally to 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.

Last Byte Pointer (LBP)

In the *mode* parameter an optional entry controlling how the Last Byte Pointer (LBP) is to be handled can be specified in addition to the access mode. This is relevant only for binary files with PAM access mode. If `lbp=yes` is specified, a check is made to see whether LBP support is possible. If this is not the case, the `fopen()`, `fopen64()` function will fail and `errno` is set to ENOSYS. The switch has further effects only when the file is closed.

When an existing file is opened and read, the LBP is always taken into account independently of the *lbp* switch:

– If the file's LBP is not equal to 0, it is evaluated. Any marker which is present is ignored.

– When LBP = 0, a marker is searched for, and the file length is determined from this. If no marker is found, the end of the last complete block is regarded as the end of file.

"...,lbp=yes"
When a file which has been modified or newly created is closed, no marker is written (even if one was present), and a valid LBP is set. In this way files with a marker can be converted to LBP without a marker.
In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

"...,lbp=no"

When a file which has been **newly created** is closed, the LBP is set to zero (=invalid). A marker is written. In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

When a file which has been **modified** is closed, the LBP is set to zero (=invalid). A marker is written only if a marker existed before. If no marker existed, none is written and the file ends with the complete last block. If the file had a valid LBP when it was opened, no marker is written as in this case it is assumed that no marker exists. In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

If the *lbp* switch is not specified, the behavior depends on the environment variable LAST_BYTE_POINTER (see also section "Environment variables" on page 104):

`LAST_BYTE_POINTER=YES`

The function behaves as if `lbp=yes` were specified.

`LAST_BYTE_POINTER=NO`

The function behaves as if `lbp=no` were specified.

Split/Nosplit switch

This switch controls the processing of text files with SAM access mode and variable record length when a maximum record length is also specified.

"...,split=yes"

– The following applies when reading:
  If a record has the maximum record length, it is assumed that the following record is the continuation of this record and the records are concatenated.

– The following applies when writing:
  A record which is longer than the maximum record length will be split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it.

"...,split=no"

When reading, records of maximum length are not concatenated with the following record.
When reading with one of the functions `fwrite`, `fprintf()`, `printf()`, `vfprintf()`, `vprintf()`, `fwprintf()`, `wprintf()`, `vfwprintf()`, `vwprintf()`, `fputs()`, `fputws()` or `puts()`, records which are longer than the maximum record length are truncated.

If the switch is not specified, "...,split=yes" applies.

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 val.

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 *mode* requires write access.

`ETXTBSY`	The file is a pure procedure file (shared text file) that is currently executing and write protection is required for `mode`.
`EOVERFLOW`	The file named is a regular file and the size of the file cannot be represented correctly in an object of type `off_t`.

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:

`stdin`	file pointer for standard input (terminal)
`stdout`	file pointer for standard output (terminal)
`stderr`	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}]"
```

`type=record`	The file is opened for record I/O. If this option is omitted, the file is opened for stream I/O.
`forg=seq`	The file is organized sequentially. Sequential files may be SAM or PAM files.
`forg=key`	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 off 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 val. 0 upon successful completion. `fork()` 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 `errno` 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 `{CHILD_MAX}` 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 *fildes*, 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* s 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:

```
                                                         ⎧[{h|l}] {d|i|o|u|x|X}⎫
                                                         ⎪[{h|l}] n            ⎪
                                     ⎧n⎫     ⎧m ⎫        ⎪[L] {e|E|f|g|G}      ⎪
{ %[a$] } [‘][-][+][␣][#][0] [ ⎨ ⎬] [ ⎨  ⎬]  ⎨{c|C|p|s|S|n}        ⎬
                                     ⎩*⎭     ⎩.*⎭        ⎪{D|O|U}              ⎪
                                                         ⎩%                    ⎭
________ _________________ _______ _______ ____________________________
   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 [−]*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`	The `unsigned int` argument is converted to unsigned octal format 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.
`u`	The `unsigned int` argument is converted to unsigned decimal format 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`	The unsigned `int` argument is converted to unsigned hexadecimal format in the style *dddd*; 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*`.`*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`	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:

```
                                    ⎧[l] {d|o|u|x}  ⎫
                                    ⎪{D|O|U|X}      ⎪
                  ⎧n⎫      ⎧m ⎫     ⎪               ⎪
 %  [-][+][0] [   ⎨ ⎬] [   ⎨  ⎬]    ⎨               ⎬
                  ⎩*⎭      ⎩.*⎭     ⎪{e|f|g}        ⎪
                                    ⎪{c|s}          ⎪
                                    ⎩%              ⎭
___ _____________________ ___________________________
 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: `[-]`*d.ddd*`e{+|-}`*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: :

```
                                                  ⎧[{h|l|ll}] {d|i|o|u|x|X}  ⎫
                                                  ⎪[{h|l|ll}]  n             ⎪
                             ⎧n⎫     ⎧m ⎫         ⎪[L] {e|E|f|g|G}           ⎪
 %  [-][+][␣][#][0] [ ⎨ ⎬] [ ⎨  ⎬] ⎨{c|p|s}                   ⎬
                             ⎩*⎭     ⎩.*⎭         ⎪{D|O|U}                   ⎪
                                                  ⎩%                         ⎭
____ ________________________________  _____________________________
 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 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 (`float` or `double`) in the form `[-]`*d.ddd*`e{+|-}`*dd*. The radix character is determined by the locale (category `LC_NUMERIC`). 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 *.m*; 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 (`float` or `double`) in `f` or `e` form (or for `G` conversions, in `E` form). The number of significant positions defects on the precision specified in *.m*. The `e` or `E` 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 (`char`). The null byte is ignored.

`p` Conversion of an argument of type pointer to `void`. The output occurs as an 8-digit hexadecimal number (analogous to the entry `%08.8x`).

`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.

`n` No conversion and output of the argument occurs. The argument is of type pointer to `int`. This integer variable is assigned the number of bytes that were generated for output by the `printf` functions up to that point.

`%` Output of the character `%`, without conversion. ❑

Return val. 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 `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 for writing.
`EFBIG`	An attempt was made to write 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 it´s controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTOU`, 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 `SIGPIPE` 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 110).

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`).

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records which are longer than the maximum record length are truncated to the maximum record length when they are written. By default or with the specification `split=yes`, these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

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 86.

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 val.

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 110).

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 117).

`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 val. 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 117).

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification split=no was entered for fopen(), records which are longer than the maximum record length are truncated to the maximum record length when they are written. By default or with the specification split=yes, these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

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.

fputwc - put wide-character code on stream

Syntax #include <wchar.h>

Optional
#include <stdio.h> ❑

wint_t fputwc(wint_t *wc*, FILE **stream*);

Description `fputwc()` 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 between successful execution of `fputwc()` 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 val.

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 `fputwc()` 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, `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.

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 val. 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 `fputwc()`.

Notes `fputws()` does not append a newline character.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records which are longer than the maximum record length are truncated to the maximum record length when they are written. By default or with the specification `split=yes`, these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

See also `fopen()`, `fputwc()`, `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 = *size* * *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 val. 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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

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 val. 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 *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 `freopen()` system call.

`EISDIR`
The named file is a directory and *mode* 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 *filename* 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 `off_t`.

`EROFS`
The named file resides on a read-only file system and *mode* requires write access.

`ETXTBSY`
The file is a pure procedure file (shared text file) that is currently executing and write protection is required for `mode`.

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 val.

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)*
␣	**blank**	x	x	x
`\n`	**newline character**	x	x	x
`\t`	**horizontal tab**	x	x	x
`\f`	**form-feed**	x	x	-
`\v`	**vertical tab**	-	x	-
`\r`	**carriage return**	-	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:

```
                                [{h|l}] {d|i|o|u|x|X}
               n                [{h|l}] n
{ %[a$] } [ {     } ]  {        [l|L] {e|E|f|g|G}         }
               *                {c|C|p|s|S}
                                {[...]|[^...]}
                                %
```

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:

```
                  [{h|l}] {d|o|x}
         ┌ n ┐    [l] {e|f}
% [ ─────┤   ├]   [D|E|F|O|X]
         └ * ┘    {c|s}
                  {[...]|[^...]}
                  %
```

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:

```
                 ⎧ [{h|l|ll}] {d|i|o|u|x|X} ⎫
      ⎧ n ⎫      ⎪ [{h|l|ll}] n             ⎪
 % [  ⎨   ⎬ ]    ⎨ [l|L] {e|E|f|g|G}        ⎬
      ⎩ * ⎭      ⎪ {c|p|s}                  ⎪
                 ⎪ {[...]|[^...]}           ⎪
                 ⎩ %                        ⎭
```

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.

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`, `i`, `o`, `u`, `x`, `X`:
conversion of an argument of type pointer to `long int` (`d, i`) or `unsigned long int` (`o, u, x, X`).

`l` before `e`, `E`, `f`, `g`, `G`:
conversion of an argument of type pointer to `double`.

`l` before `n`:
The argument is of the 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`.

`h`	`h` before `d`, `i`, `o`, `u`, `x`, `X`: conversion of an argument of type pointer to `short int` (`d, i`) or `unsigned short int` (`o`, `u`, `x`, `X`). `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 val. 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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

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 f`seek64()`.

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:

offset 0L or value determined by a previous *ftell* call.

whence `SEEK_SET` (beginning of file)
`SEEK_END` (end of file)

Meaningful combinations and their effects:

offset	*whence*	**Effect**
`ftell` value	`SEEK_SET`	Position to the location marked by `ftell()`
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:

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:

offset	*whence*	**Effect**
0L	`SEEK_SET`	Position to the beginning of the file.
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.
`ftell` value	`SEEK_SET`	Position to the location marked by an `ftell` call.

Return val. 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 `ulimit()`).

`EINTR` The write operation was terminated due to the receipt of a signal, and no data was transferred.

`EINVAL`	The *whence* argument is invalid. The resulting file-position indicator would be set to a negative value.
`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.
`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.
`ENXIO`	The device does not exist or it cannot be accessed.
`EOVERFLOW`	For *fseek()*: The resulting file offset value cannot be represented correctly in an object of type `long`.
`EOVERFLOW`	For *fseeko()*: The resulting file offset value cannot be represented correctly in an object of type `off_t`.

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 (SYSDTA, 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 val. 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 (SYSDTA, 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, `fseetpos()` 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, fstatat - get file status of open file

Name **fstat, fstat64, fstatat, fstatat64**

Syntax #include <sys/stat.h>

Optional
#include <sys/types.h>

int fstat(int *fildes*, struct stat **buf*);
int fstat64(int *fildes*, struct stat64 **buf*);
int fstatat(int *fd*, const char **path*, struct stat **buf*, int *flag*);
int fstatat64(int *fd*, const char **path*, struct stat64 **buf*, int *flag*);

Description `fstat()` obtains information on an open file associated with the file descriptor *fildes*, 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:

st_mode The mode of the file is defined in the system call mknod(). Apart from the modes defined in mknod(), the mode of a file can be S_IFLNK if the file is a symbolic link, or S_IFSOCK if a socket descriptor is involved.

st_ino Uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.

st_dev Uniquely identifies the file system that contains the file.

st_rdev 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.

st_nlink May be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For regular files, this is the address of the end of the file. It is undefined for block special or character special files. For PAM files this member contains the file size. Any existing marker is not considered. If the LBP is zero, the entire last block counts to the size.

st_atime Time when file data was last accessed. Modified by the following system calls: creat(), mknod(), pipe(), utime() and read().

st_mtime Time when data was last updated. Modified by the following system calls: creat(), mknod(), pipe(), utime() and write().

st_ctime Time when the file status was last changed. Modified by the following system calls: chmod(), chown(), creat(), link(), mknod(), pipe(), unlink(), utime() and write().

st_blksize A hint as to the 'best' unit size for I/O operations. This field is not defined for block special or character special files.

st_blocks 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:

mode_t *st_mode* File mode containing access permissions and file type.

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 S_IFDVSBS2=X'10000000'. This type is, however, not disjoint to S_IFPOSIXBS2. The S_ISDVSBS2(*mode*) macro can be used to query. Introduction of a new file type S_IFDVSNODE=X'20000000'. This type is also not disjoint to S_IFPOSIXBS2. The S_ISDVSNODE(*mode*) macro can be used to query. A node file is also a BS2000 DVS file. I.e. for node files the bit `S_IFDVSBS2` is always set.
time_t *st_atime*	Last access time as is usual in BS2000, but in seconds since 1.1.1970 UTC).
time_t *st_mtime*	Last modification time.
time_t *st_ctime*	Creation time.
long *st_blksize*	Block size, 2K (i.e. 1 PAM page).
long *st_blocks*	Number of blocks on the disk that are occupied by the file.
dev_t *st_dev*	Contains the 4-byte `catalog ID`.

The two consecutive fields

uid_t *st_uid*	and
gid_t *st_uid*	contain the 8-byte BS2000 user ID.

All other fields are set to 0.

The `fstatat()` and `fstatat64()` functions are equivalent to the `stat()` and `stat64()` and the `lstat()` and `lstat64()` functions depending on the value *flag* except when the *path* parameter specifies a relative path. In this case the file whose status is to be determined is not searched for in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the functions check whether a search is permitted in the connected directory with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

In the *flag* parameter, the value `AT_SYMLINK_NOFOLLOW`, which is defined in the `fnctl.h` header, can be transferred. If *path* specifies a symbolic link, the status of the symbolic link is returned.

Return val.	0	if successful.
	-1	if an error occurs; for POSIX files `errno` is set to indicate the error.

Errors `fstat()`, `fstat64()`, `fstatat()` and `fstatat64()` will fail if:

`EBADF` *fildes* 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` *fildes* 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.

In addition, `fstatat()` and `fstatat64()` fail when the following applies:

`EACCES` The *fd* parameter was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

`ENOTDIR` The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

`EINVAL` The value of the *flag* parameter is invalid.

See also `chmod()`, `chown()`, `creat()`, `link()`, `lstat()`, `mknod()`, `stat()`, `unlink()`, `write()`, `fcntl.h`, `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 fildes, struct statvfs *buf);
int statvfs (const char *path, struct statvfs *buf);
int fstatvfs64 (int fildes, 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 *fildes* 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.

fildes 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 *fildes* 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 f`statvfs64()` / `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[FSTYPSZ]; /* 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 val.

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` *fildes* 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 *fildes*);

Description `fsync()` causes all the modified data and attributes of *fildes* that are still in the buffer to be written to the physical storage medium.

Return val.

0	if successful
-1	if an error occurs; `errno` is set to indicate the error.

Errors `fsync()` will fail if:

`EBADF`	*fildes* is not a valid file descriptor.
`EINTR`	A signal was caught during the `fsync()` system call.
`EINVAL`	*fildes* 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 val. 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` .

`EOVERFLOW` 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, ftime64 - get date and time

Syntax

#include <sys/timeb.h>

int ftime(struct timeb *tp*);
int ftime64(struct timeb64 **tp*);

Description

`ftime()` enters in the structure pointed to by *tp* the exact time in milliseconds since January 1, 1970, 00:00:00. As of 19.1.2038 03:14:08 hrs UTC `ftime()` outputs the message CCM0014 and ends the program.

The `ftime64()` function behaves like `ftime()` with the difference that it also returns correct results after 19.1.2038 03:14:07 hrs.

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         */
};
```

or

```
struct timeb64 {
   time64_t time;
   unsigned short millitm;
   short timezone;
   short dstflag;
   short filler;
};
```

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, 1970 00:00:00) and also includes milliseconds.

For portability reasons, additional options have been included in the structures `timeb` and `timeb64` . However, they are not supplied in the BS2000 environment. ❑

Return val.

0 always.

Notes

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 val.

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 val. 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`
fildes is not a file descriptor that is opened for writing.

`EINVAL` *fildes* 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 *path*.

`EISDIR` The file addressed via *path* is a directory.

`ELOOP` Too many symbolic links were encountered in resolving *path*.

`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 *path* references an empty string.

`ENOTDIR` A component of the path prefix from *path* is not a directory.

`EROFS` The file addressed via *path* 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 val.

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

`ftw()` will fail if:

`EACCES` Search permission is denied for any component of *path* or read permission is denied for *path*.

Extension
`EBADF` 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}`.

`ENOENT` *path* points to the name of a file that does not exist or to an empty string.

`ENOTDIR` A component of *path* is not a directory.

Notes

Since `ftw()` is recursive, it is possible for it to terminate with a memory error when applied to very deep file structures.

`ftw()` uses `malloc()` to allocate dynamic storage during its operation. If `ftw()` is forcibly terminated, such as by `longjmp()` or `siglongjmp()` being executed by the function pointed to by *fn* or a signal-handling routine, `ftw()` 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 *fn* return a non-zero value at its next invocation.

`ftw()` is executed only for POSIX files.

See also

`longjmp()`, `malloc()`, `siglongjmp()`, `stat()`, `ftw.h`.

futimesat - setting file access and update times

Syntax

#include <sys/time.h>

int futimesat(int *fd*, const char **path*, const struct timeval *times*[2]);

Description

The `futimesat()` function sets the access and update times of a file to the values specified in *times*. The times of the file are changed to which the *path* parameter points relative to the directory connected with the file descriptor *fd*. The function permits time specifications which are accurate to the microsecond.

The *times* parameter is an array consisting of two structures of the type *timeval*. The access time is set to the value of the first element, and the update time to the value of the second element. The times in the *timeval* structure are specified in seconds and microseconds since the epoch.

When *times* is the null pointer, the access and update times are set to the current time. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

A process may call `futimesat()` with the null pointer for *times* parameter only if it has one of the following properties:

– owner of the file,
– write authorization for the file, or
– special rights.

When the value `AT_FDCWD` is transferred to the `futimesat()` function for the *fd* parameter, the current directory is used.

Return val.

0 in the case of success,

-1 in the case of an error `errno` is set to display the error.

Errors

`futimesat()` fails when the following applies:

`EACCES` A component of the path may not be searched, or *times* is a null pointer and the effective user number is not that of the system administrator and not that of the owner of the file, and write access is rejected
or
the *fd* parameter was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

Extension

`EFAULT` *times* is not equal to zero and points beyond the process's assigned address space, or *path* points beyond the process's assigned address space.

`EINTR` A signal was intercepted during the system call `utime()` .

`EINVAL` An attempt was made to access a BS2000 file or the value of the *flag* parameter is invalid.

`ELOOP` During the compilation of *path* too many symbolic links occurred to ❑.

`ENAMETOOLONG`
The length of *path* exceeds `{PATH_MAX}` or the length of a component of *path* exceeds `{NAME_MAX}`.

`ENOENT` The specified file does not exist.

`ENOTDIR` A component of the path is not a directory, or the *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

`EPERM` The effective user number if not that of the system administrator and not that of the owner of the file, and *times* is not equal to zero.

`EROFS` The file system containing the file has been mounted write-protected.

See also `sys/time.h`.

funlockfile - unlock standard input/output

Syntax #include <stdio.h>

void funlockfile(FILE **file*);

Description See `flockfile()`.

fwide - specify file orientation

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

int fwide(FILE **dz*, int *mode*);

Description `fwide()` specifies the orientation of the file with the file pointer *dz* 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 `fwide()` does not change this orientation.

`fwide()` attempts to set the orientation depending on the *mode* argument in the following manner:

mode > 0	FIle is wide character-oriented.
mode < 0	FIle is byte-oriented.
mode = 0	The file orientation is not changed.

Returnwert

> 0	if *dz* is wide character-oriented after calling `fwide()`.
< 0	if *dz* is byte-oriented after calling `fwide()`.
0	if *dz* 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.

`vswprintf()` 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 117).

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:

```
                                                  ⎧[{h|l|ll}] {d|i|o|u|x|X}⎫
                                                  ⎪[{h|l|ll}] n            ⎪
                               ⎧n⎫      ⎧m⎫       ⎪[L] {e|E|f|g|G}         ⎪
 %      [-][+][␣][#][0]    [   ⎨ ⎬ ] [. ⎨ ⎬ ]     ⎨[l] {c|s|p}             ⎬
                               ⎩*⎭      ⎩*⎭       ⎪{D|O|U|C|S}             ⎪
                                                  ⎩%                       ⎭
________ _________________ _______ _______ ____________________________
   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 `l`, 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 `l`, 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 `ls`.

`C` Same as `lc`.

`p` The argument must be a pointer to `void`. The value is output as an 8-digit hexadecimal number.

`n` The argument must be a pointer to `int` into which is written the number of bytes written to the output so far by this call to one of the `fwprintf` functions. No argument is converted.

`%` The wide character `%` is output; no argument is converted. The complete conversion specification must be of the form %%.

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 val. 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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records which are longer than the maximum record length are truncated to the maximum record length when they are written. By default or with the specification `split=yes`, these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

See also `btowc()`, `fprintf()`, `mbrtowc()`, `printf()`

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 = *size* * *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 val. 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 110).

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 117).

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records which are longer than the maximum record length are truncated to the maximum record length when they are written. By default or with the specification `split=yes`, these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

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`.

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:

```
                 ⎧[{h|l|ll}]{d|i|o|n|u|x|X}⎫
                 ⎪[l] {c|s}                ⎪
          ⎧m⎫    ⎪[l|L] {e|E|f|g|G}        ⎪
{ % } [ ⎨   ⎬] ⎨{p}                      ⎬
          ⎩*⎭    ⎪[l] {[...]|[^...]}       ⎪
                 ⎩%                        ⎭
```

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 val. 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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

See also `scanf()`, `sscanf()`, `fscanf()`, `wcstod()`, `wcstol()`, `wcstoul()`, `wcrtomb()`

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:

$$\int_0^{\infty} e^{-t} \quad t^{x-1} \quad dt$$

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 val.

`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 ≤ 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 val. 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`.

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

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, putc_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 val. 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 val.
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 val. `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 `SYSPOSIX.HOME` exists, the `HOME` variable of the C programming environment is initialized with the value of the `SYSPOSIX.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 val. 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` *size* is equal to 0.

`ENOMEM` Insufficient storage space is available.

`ERANGE` *size* is less than 0, or is greater than 0, but smaller than the length of the pathname + 1.

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 `DATEMSK` 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:

`%%`	same as `%`
`%a`	abbreviated weekday name
`%A`	weekday name in full
`%b`	abbreviated month name
`%B`	month name in full
`%c`	local date and time representation
`%d`	day of the month (01 - 31; the leading 0 is optional)
`%e`	same as `%d`
`%D`	date as `%m/%d/%y`
`%h`	abbreviated month name
`%H`	hour (00 - 23)
`%I`	hour (01 - 12)
`%m`	month number (01 - 12)
`%M`	minute (00 - 59)
`%n`	same as `\n`
`%p`	local equivalent of AM or PM
`%r`	time as `%I:%M:%S %p`
`%R`	time as `%H:%M`
`%S`	second (00-61). Leap seconds are allowed, but the effects of using algorithms are unpredictable.
`%t`	insert tab
`%T`	time as `%H:%M:%S`
`%w`	weekday number (0 - 6; Sunday = 0)
`%x`	local date representation

%X	local time representation
%y	year in current century (00 - 99)
%Y	year as `ccyy` (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 `getdate()`, an input error occurs. `getdate()` 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 s`tructure 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 val. 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 `DATEMSK` 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");
```

Beispiel 3 Local time and date specifications are also supported. Example 3 shows how these can be defined in templates.

Call	**Line in template file**
`getdate("11/27/86");`	`%m/%d/%y`
`getdate("27.11.86");`	`%d.%m.%y`
`getdate("86-11-27");`	`%y-%m-%d`
`getdate("Friday 12:00:00");`	`%A %H:%M:%S`

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 val. 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 val. 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 val. Effective group ID of the calling process.
The function is always successful.

See also `getgid()`, `setgid()`, `sys/types.h`, `unistd.h`. manual "POSIX Basics" [1].

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 val. 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 49).

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 104). ❑

See also `exec`, `environ`, `putenv()`, `setenv()`, `unsetenv()`, `stdlib.h`, section "Scope of the supported C library" on page 49, and section "Environment variables" on page 104.

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 val. Effective user ID of the calling process
The function is always successful.

See also `getuid()`, `setuid()`, `sys/types.h`, `unistd.h`, manual "POSIX Basics" [1].

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 val. Real group ID of the calling process.
The function is always successful.

Siehe auch `getegid()`, `getuid()`, `setgid()`, `sys/types.h`, `unistd.h`, `td.h`, manual "POSIX Basics" [1].

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" [1]).

`getgrgid()` is not thread-safe. Use the reentrant function `getgrgid_r()` when needed.

Return val. 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 o 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" [1].

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 val. 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" [1]).

`getgrnam()` is not thread-safe. Use the reentrant function `getgrnam_r()` when needed.

Return val. 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`,

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 val. 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" [1].

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 val.

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" [1].

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 val. 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 val. 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 val. 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 val. 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 val. 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 *fildes*, struct strbuf **ctlptr*, struct strbuf **dataptr*, int **flagsp*);

int getpmsg(int *fildes*, 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()`.

fildes 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 val. 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` *fildes* 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 *fildes* is directly or indirectly linked downstream with a multiplexer.

`ENOSTR` No stream is assigned to the *fildes* 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" [2]. 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" [2]). 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 val. 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 or if the next option character is the question mark (?) from the command line.
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" [2]).

An error has occurred only if the optopt variable does not contain a question mark (?). Otherwise the question mark is the next option character from the command line, and the function was concluded successfully.

-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 manuals "C Compiler" [3] and "C/C++ Compiler" [4]). ❑

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, 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" [2]).

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 val.

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 val. 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 val.

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.

`getpgmname()` returns the path name of the `exec()` function via which the program was started, that was passed as the first parameter. This path name may differ from `argv[0]`. For example, `getpgmname()` always returns the fully qualified path name for programs started directly from the shell, but `argv[0]` contains the name just as it was specified by the user.

BS2000

The result corresponds to `argv[0]` of the `main` function. ❑

Return val.

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 val. 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" [1].

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 val. 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 *fildes*, 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> ❑

pid_t getppid(void);

Description `getppid()` returns the parent process ID of the calling process.

Return val. 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 val.
`getpriority()`:

-20 ≤ return value ≤ 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);

DescriptionSee 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" [1]).

`getpwnam()` is not thread-safe. Use the reentrant function `getpwnam_r()` when needed.

Return val. 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" [1].

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 val. 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 val. 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" [1].

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
`RLIMIT_CORE`	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.
`RLIMIT_CPU`	Maximum CPU time used by a process.	`SIGXCPU` is sent to the process. If the process blocks, catches or ignores `SIGXCPU`, the behavior is undefined.
`RLIMIT_DATA`	Maximum size of the data segment of a process in bytes. Under POSIX the size is unlimited, because `sbrk()`, `brk()` and `malloc()` use independent memory.	`brk`(), `malloc()` and `sbrk() will` fail and `errno` will contain `ENOMEM`.
`RLIMIT_FSIZE`	Maximum length of a file in bytes that can be generated by a process. A length of 0 prevents files from being generated.	`SIGXFSZ` is sent to the process. If the process blocks, catches or ignores `SIGXFSZ`, further attempts to enlarge the file will fail and `errno` will contain `EFBIG`.
`RLIMIT_NOFILE`	Maximum number of open file descriptors that a process can have.	Functions which create new file descriptors will fail and `errno` will contain `EMFILE`.
`RLIMIT_STACK`	Maximum size of the process stack in bytes. The system does not let the stack grow automatically beyond this limit.	`SIGSEGV` is sent to the process. If the process blocks, ignores or catches `SIGSEGV` and does not use an alternative stack (see `sigaltstack()`), `SIG_DFL` is set as the handling mode of `SIGSEGV`.
`RLIMIT_AS`	Maximum length of the address area of a process in bytes.	The functions `brk()`, `malloc()`, `mmap()` and `sbrk()` will fail and `errno` will contain `ENOMEM`. 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 val. 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:

```
struct timeval ru_utime    The total time the execution takes in user mode.
                           The interval is specified in seconds and micro-
                           seconds.
struct timeval ru_stime    The total time the execution takes in system
                           mode. The interval is specified in seconds and
                           microseconds.
```

Return val. 0 if successful. The `rusage` structure is filled up with the corresponding values.

-1 if an error occurs. `errno` is set to indicate the error.

Errors `getrusage()` will fail if:

EINVAL The *who* argument does not contain a valid value.

Extension
`EFAULT` The address specified by the *r_usage* 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 val.

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 `gets()` 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 val. 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()`, `getpdig()`, `setpgid()`, `setsid()`, `unistd.h`.

getsubopt - get suboptions from string

Syntax #include <stdlib.h>

int getsubopt (char * **optionp*, char * const **tokens*, char * **valuep*);

Description `getsubopt()` extracts subobtions 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 val. 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, gettimeofday64 - read current time of day

Syntax

#include <sys/time.h>

int gettimeofday(struct timeval **tp*, void **tzp*);
int gettimeofday64(struct timeval64 **tp*, void **tzp*);

Description

`gettimeofday()` und `gettimeofday64()` read 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` or `timeval64`, containing the following members:

```
long     tv_sec;    /* seconds since January 1, 1970 */
```
or
```
time64_t tv_sec;    /* seconds since January 1, 1970 */
```
and
```
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 val.

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 val. 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 val. 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 val. 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()`.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records of maximum length are not concatenated with the subsequent record when they are read. By default or with the specification `split=yes`, when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated ❑.

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 val.

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 val. 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 val.
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 val.
$\neq 0$ if the string matches the pattern.

0 if no match was found.

gmtime, gmtime64 - convert date and time to UTC

Syntax
#include <time.h>

struct tm *gmtime(const time_t **clock*);
struct tm *gmtime64(const time64_t **clock*);

Description
The functions `gmtime()` and `gmtime64()` interpret the time specification of the value to which `clock` points as the number of seconds that have elapsed since 1.1.1970 00:00:00 hrs UTC (epoch). They calculate from this the date and time in UTC and store it in a type *tm* structure. Negative values are interpreted as seconds before the epoch. The following points in time are considered invalid:

– with `gmtime()` points in time before 13.12.1901 20:45:52 hrs UTC and after 19.01.2038 03:14:07 Uhr UTC
– with `gmtime64()` points in time before 1.1.1900 00:00:00 hrs UTC and after 31.12.9999 23:59:59 hrs UTC.

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, 1970, 00:00:00 local time. 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 val. Pointer to a structure of type `struct tm`
if successful.

`EOVEFLOW` In case of an error `NULL` und `errno`.

Notes The `asctime()`, `ctime()`, `ctime64()`, `gmtime()`, `gmtime64()`, `localtime()` and `localtime64()` functions write their result into the same internal C data area. This means that each of these function calls overwrites the previous result of any of the other functions.

`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 val. 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 val.

0	if successful.
-1	if there is an error. `errno` is set to indicate the error.

Errors `grantpt()` will fail if:

`EBADF`	*fildes* is not a valid open file descriptor
`EINVAL`	*fildes* 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 val.

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 val. 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 val. `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 *cd* 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 val. 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 val.

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 *fromcode* and *tocode* 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 xs 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 val. 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 val. 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`.

initgroups - initialize group access lists

Syntax

```
#include <grp.h>
#include <sys/types.h>
```

initgroups(const char **name*, gid_t *basegid*);

Description

The `initgroups()` function can only be called by the system administrator. The `initgroups()` function initializes the additional group access list of the calling process. To do this, `initgroups()` reads the group database /`etc/group` and uses all groups whose user specified by the name parameter is a member. Groups that are additionally specified by the *basegid* parameter are also entered in the list.

Generally, the primary group number is passed in *basegid*, as defined in the BS2000 SRPM (System Resources and Privileges Management) with the /MOD-POSIX-USER- ATTR or /MOD-POSIX-USER-DEFAULTS command.

The `initgroups()` function also exists as an ASCII function.

Return val.

0	Execution was successful.
-1	Otherwise. `errno` indicates the cause of the error.

Errors

`initgroups()` schlägt fehl, wenn gilt:

`EPERM`	The effective user number is not the user number of the system administrator.

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 x (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.

`inistate()` is not thread-safe. Use the reentrant function `rand_r()` when needed.

Return val. `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 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", random());
}
```

See also `drand48drand48()`, `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 *fildes*, 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 *fildes* and are interpreted by the device driver.

fildes 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:

EINVAL Invalid module name.
EFAULT *arg* references a point outside the reserved address space.
ENXIO The open() function of the new module failed.
ENXIO Hang-up signal received for *fildes*.

I_POP 'Pops' the module directly below the stream head out of the stream referenced by *fildes*. 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:

EINVAL No module stream exists.
ENXIO Hang-up signal received for *fildes*.

I_LOOK Determines the name of the module directly below the stream head in the stream specified by *fildes*, 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_PCPROTO) 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 ≠ 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` *fildes*;
`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. *fildes* 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 *fildes* 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 `strioctl` structure containing the following components:

`int` *ic_cmd*;
`int` *ic_timout*;
`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_timout* 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 `strioctl` 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_timout* is less than -1.

`ENXIO`
: Hang-up signal received for *fildes*.

`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 *fildes* 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` *fildes* is not linked with a pipe.

`ENXIO` Hang-up signal received for *fildes*.

`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 *fildes*.

`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 *fildes*. 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 *fildes* 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 *fildes*.

`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 *fildes* 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.
- *fildes* 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 *fildes*, 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 *fildes* and *arg*. *fildes* 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 *fildes* 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 *fildes*.

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 *fildes* is not the stream for which the I_LINK operation supplied by *arg* was executed.

EINVAL *fildes* 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 *fildes* 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 *fildes* 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 *fildes*, 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 *fildes*.

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 *fildes* 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.
  - *fildes* 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 *fildes* 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 *fildes* and *arg*. *fildes* 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 *fildes* 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 *fildes*.

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
: *fildes* 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 *fildes* 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 val. 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 *fildes* is not a valid open file descriptor.

EINTR A signal was caught during the `ioctl()` system call.

EINVAL The stream or multiplexer identified by *fildes* 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 *fildes* 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 *fildes* identifies a valid STREAMS file, but the associated device driver does not support the `ioctl()` function.

ENOLINK *fildes* 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 val. ≠ 0 Alphanumeric

0 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 val.

≠ 0	Letter
0	Not a letteρ

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()`, `iscntrl()`, `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 val.

≠ 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 *fildes*);

Description The `isastream()` function checks whether a file descriptor represents a `STREAMS` file. *fildes* refers to an open file.

Return val.

1	if *fildes* represents a `STREAMS` file.
0	if *fildes* does not represent a `STREAMS` file.
-1	if an error occurs. `errno` is set to indicate the error.

Errors `isastream()` will fail if:

`EBADF`	*fildes* is not a valid open file descriptor.

See also `stropts.h`.

isatty - test for terminal device

Syntax #include <unistd.h>

int isatty(int *fildes*);

Description `isatty()` tests whether the file descriptor specified with *fildes* is associated with a terminal device.

Return val. 1 if successful. *fildes* is associated with a terminal.

0 if an error occurs. `errno` is set to indicate the error.

Errors `isatty()` will fail if:

`EBADF` *fildes* is not a valid file descriptor.

`ENOTTY` *fildes* is not associated with a terminal.

See also `unistd.h`.

iscntrl - test for control character

Syntax #include <ctype.h>

int iscntrl(int *c*);

Description `iscntrl()` 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 val.

$\neq 0$	Control character
0	Not a control character

Notes `iscntrl()` 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 iscntrl`).

The behavior of `iscntrl()` 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 val.

≠ 0	Decimal digit
0	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`.

isebcdic - test for EBCDIC character *(BS2000)*

Syntax #include <ctype.h>

int isebcdic(int *c*);

Description `isebcdic` tests whether the value of the character *c* represents an EBCDIC character (values 0 - 255).

Return val.	≠ 0	The value of *c* represents an EBCDIC character (values 0 - 255).
	0	Not an EBCDIC character (values ≠ 0 - 255).

Notes `isebcdic` 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 isebcdic`).

`isebcdic` is a synonym for `isascii`.

See also `isalpha()`, `isalnum()`, `isascii()`, `iscntrl()`, `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 val.

≠ 0	Character with a visible representation
0	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 val.

≠ 0	Lowercase letter
0	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 val.

0 if *x* 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 val. $\neq 0$ 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 val.

≠ 0	Punctuation character
0	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 val.

≠ 0	White-space character
0	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 val.

≠ 0	Uppercase letter
0	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 val.

≠ 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 (`#undefiswalnum`).

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 val.

≠ 0	Alphabetic
0	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 val.

$\neq 0$	Control wide character code
0	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 val.

≠ 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:

```
iswalnum(wc)       iswctype(wc, wctype("alnum"))
iswalpha(wc)       iswctype(wc, wctype("alpha"))
iswcntrl(wc)       iswctype(wc, wctype("cntrl"))
iswdigit(wc)       iswctype(wc, wctype("digit"))
iswgraph(wc)       iswctype(wc, wctype("graph"))
iswlower(wc)       iswctype(wc, wctype("lower"))
iswprint(wc)       iswctype(wc, wctype("print"))
iswpunct(wc)       iswctype(wc, wctype("punct"))
iswspace(wc)       iswctype(wc, wctype("space"))
iswupper(wc)       iswctype(wc, wctype("upper"))
iswxdigit(wc)      iswctype(wc, wctype("xdigit"))
```

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 val.

≠ 0	Decimal digit
0	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 val. ≠ 0 Character with a visible representation

0 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 val.

≠ 0	Lowercase letter
0	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 val.

≠ 0	Printing character (alphanumeric character, special character or blanks)
0	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 val.

≠ 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 `iswpunc()` 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`.

iswspace - test for white-space wide character

Syntax
#include <wchar.h>

int iswspace(wint_t *wc*);

Description
`iswspace()` 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 val.

$\neq 0$	White-space wide character
0	Not a white-space wide character

Notes
`iswspace()` 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 iswspace`).

The behavior of `iswspace()` 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 val.

≠ 0	Uppercase letter
0	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 val.

≠ 0	Hexadecimal digit wide character code
0	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 val.

≠ 0	Hexadecimal digit
0	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 val. 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 K2
SIGILL	PROCHK	Execution of an invalid instruction
SIGABRT	–	`raise` signal for program abort with `_exit(-1)`
SIGFPE	PROCHK	Error in a floating-point operation
SIGKILL	–	`raise` signal for program abort with `exit(-1)`
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 val. 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 val.
See `kill()`.

Errors
See `kill()`.

See also
`getpgid()`, `getpid()`, `kill()`, `raise()`, `signal.h`.

l64a - convert 32-bit integer number to string

Syntax #include <stdlib.h>

char *l64a (long *value*);

Description See `a64l()`.

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 val. 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 val.

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 *path*.
`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 is 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 `{PATH_MAX}`.

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 * 2exp
where x is the mantissa and exp is the exponent.

`ldexp()` is the inverse function of `frexp()`.

Return val.

Value of x * 2exp	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 val. 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:

$$\int_0^\infty e^{-t} \quad t^{x-1} \quad dt$$

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 val.

`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, linkat - create link to file

Syntax

#include <unistd.h>

int link(const char **path1*, const char **path2*);
int linkat(int *fd1*, const char **path1*, int *fd2*, const char **path2*, int *flag*);

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.

The `linkat()` function is equivalent to the `link()` function except when symbolic links are to be handled in accordance with the value transferred in the *flag* parameter (see below), or when the *path1* or *path2* parameter specifies a relative path. If *path1* specifies a relative pathname, this is interpreted as a path relative to the directory connected with the file descriptor *fd1*. If *path2* specifies a relative pathname, this is interpreted as a path relative to the directory connected with the file descriptor *fd2*. If a file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

When the value `AT_FDCWD` was transferred to the `linkat()` function for the *fd1* or *fd2* parameter, the current directory for determining the file of the corresponding path is used.

In the *flag* parameter, the value `AT_SYMLINK_FOLLOW`, which is defined in the `fnctl.h` header, can be transferred. If *path1* specifies a symbolic link, a new symbolic link is generated for the destination.

Return val. 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()` and `linkat()` 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` *path1* or *path2* points outside the allocated address space.

`EINTR` A signal was caught during the `link()` system call.

`EINVAL` An attempt was made to access a BS2000 file.

`ELOOP` Too many symbolic links were encountered in resolving *path1* or *path2*. ❑

`EMLINK` The number of links to the file named by *path1* would exceed `{LINK_MAX}`.

`ENAMETOOLONG`
The length of *path1* or *path2* exceeds `{PATH_MAX}`, or a pathname component is longer than `{NAME_MAX}`.

`ENOENT` A component of either path prefix does not exist; the file named by *path1* does not exist; or *path1* or *path2* 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 *path1* 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 *path2* and the file named by *path1* are on different file systems.

In addition, `linkat()` fails if the following applies:

`EACCES`	The file descriptor *fd1* or *fd2* was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.
`EBADF`	The *path1* parameter does not specify an absolute pathname, and the *fd1* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching, or the *path2* parameter does not specify an absolute pathname, and the *fd2* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor for reading or searching.
`ENOTDIR`	The *path1* or *path2* parameter does not specify an absolute pathname, and the corresponding file descriptor *fd1* / *fd2* is not connected with a directory.
`EINVAL`	The value of the *flag* parameter is invalid.

Notes `link()` and `linkat()` are executed only for POSIX files

See also `readlink()`, `remove`, `symlink()`, `unlink()`, `fcntl.h`, `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 val.

`|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:

Quotient * Divisor + Remainder = Dividend

Return val.
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 val.

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 val. 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" [2].

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 alphabetical 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 val. 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
`int_curr_symbol`	`"EUR"`	`"NOK "`	`"CHF "`
`currency_symbol`	`"?"`	`"kr"`	`"SFrs."`
`mon_decimal_point`	`","`	`","`	`"."`
`mon_thousands_sep`	`"."`	`"."`	`","`
`mon_grouping`	`3;3`	`"\3"`	`"\3`
`positive_sign`	`""`	`""`	`""`
`negative_sign`	`"-"`	`"-"`	`"C"`
`int_frac_digits`	2	2	2
`frac_digits`	2	2	2
`p_cs_precedes`	0	1	1
`p_sep_by_space`	1	0	0
`n_cs_precedes`	0	1	1
`n_sep_by_space`	1	0	0
`p_sign_posn`	1	1	1
`n_sign_posn`	1	2	2

See also

`isalpha()`, `isascii()`, `nl_langinfo()`, `printf()`, `scanf()`, `setlocale()`, `strcat()`, `strchr()`, `strcmp()`, `strcoll()`, `strcpy()`, `strftime()`, `strlen()`, `strpbrk()`, `strspn()`, `strtok()`, `strxrfm()`, `strtod()`, `langinfo.h`, `local.h`, section “Locale” on page 86.

localtime, localtime64 - convert date and time to local time

Syntax
#include <time.h>

struct tm *localtime(const time_t *clock);
struct tm *localtime64(const time64_t *clock);

Description
The functions `localtime()` and `localtime64()` interpret the time specification of the value to which `clock` points as the number of seconds that have elapsed since 1.1.1970 00:00:00 hrs UTC (epoch). They calculate from this the date and time in UTC and store it in a type *tm* structure. Negative values are interpreted as seconds before the epoch. The following points in time are considered invalid:

- with `localtime()` points in time before 13.12.1901 20:45:52 hrs UTC and after 19.01.2038 03:14:07 Uhr UTC
- with `localtime64()` points in time before 1.1.1900 00:00:00 hrs UTC and after 31.12.9999 23:59:59 hrs UTC.

The local time zone information is used as if the `tzset` function has been called.

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, 1970, 00:00:00 local time. 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 val. Pointer to the `tm` structure
if successful.

`EOVEFLOW` In case of an error `NULL` und `errno`.

Notes The `asctime()`, `ctime()`, `ctime64()`, `gmtime()`, `gmtime64()`, `localtime()` and `localtime64()` functions write their result into the same internal C data area. This means that each of these function calls overwrites the previous result of any of the other functions.

`localtime()` does not support local date and time formats; to ensure maximum portability, `strftime()` 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()`, `strftime()`, `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 val.

Pointer	to a string pointed to by *result* 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 d`eadlocks 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 `lockf64()` 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 val.	0	if successful.
	-1	if an error occurs. `errno` is set to indicate the error. Existing locks are not changed.

Errors `lockf()` and `lockf64()` will fail if:

`EBADF` *fildes* is not a valid open file descriptor, or *function* is `F_LOCK` or `F_TLOCK` and the file addressed via *fildes* is not opened for writing.

`EACCES` *function* is `F_TLOCK` or `F_TEST`, and the section is already locked by another process.

`EDEADLK` *function* is `F_LOCK` and a deadlock would occur.

`EINTR` A signal was caught during execution of the function.

`EAGAIN` *function* is `F_LOCK` or `F_TLOCK` and the file was generated with `mmap()`.

`ENOLCK` *function* is `F_LOCK`, `F_TLOCK` or `F_ULOCK`, and there is no longer enough storage space for additional entries in the lock table.

`EINVAL` *fildes* points to a file type that cannot be locked in this implementation, or the contents of *function* are invalid, or
the sum of *size* plus the current file offset is less than 0 or greater than the highest permissible file offset.

`ECOMM` *fildes* 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 `off_t`.

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 <regexp.h>

extern char *locs;

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" [2].

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 val.

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 val.

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 val.

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 e`xponent part of *x* as `int`, but as a double-precision signed floating-point number.

Return val. 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 `_lngjmp()` 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 val.

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 val.

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 val.

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 val.

**key*	`lfind()`: if successful. `lsearch()`: if successful, and also for a newly added element.
Null pointer	`lfind()`: 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:

offset	*whence*	**Effect**
`tell/lseek` value	`SEEK_SET`	Position to the location marked by `tell()` or `lseek()`.
`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:

offset	*whence*	**Effect**
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 val. 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 *fildes* is not an open file descriptor.

EINVAL *whence* is not a proper value, or the resulting file offset would be invalid.

ESPIPE *fildes* 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`.

lstat - query file status

Name **lstat, 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 val. 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 val. 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 val.

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 val. 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 val.

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 val.

-1	if *n* = 0.
0	if *s* 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 compete 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 val.
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 an 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 val.

Value ≠ 0	if *ps* 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 `mbrtowc()` 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 val.

`(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 val. 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 val.

-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 val.

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 val.

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 val.

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 val. 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 val. 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 `memalloc()`. 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 `memalloc()` 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 `memalloc()`.

The values passed to `memfree()` must match those of the corresponding `memalloc()` call. Random values will lead to critical errors in the working memory!

See also
`memalloc()`.

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 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 val. 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 val.

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 val.

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, mkdirat - make directory

Syntax #include <sys/stat.h>

Optional
#include <sys/types.h> ❑

int mkdir(const char **path*, mode_t *mode*);
int mkdirat(int *fd*, 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.

The `mkdirat()` function is equivalent to the `mkdir()` function except when the *path* parameter specifies a relative path. In this case the new directory is not created in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

When the value `AT_FDCWD` is transferred to the `mkdirat()` function for the *fd* parameter, the current directory is used.

Return val. 0 if successful.

-1 if an error occurs. No directory is created and `errno` is set to indicate the error.

Errors `mkdir()` and `mkdirat()` 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.

In addition, `mkdirat()` fails if the following applies:

`EACCES` The file descriptor *fd* was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

`ENOTDIR` The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

Notes `mkdir()` and `mkdirat()` are executed only for POSIX files

See also `chmod()`, `mknod()`, `umask()`, `stat()`, `fcntl.h`, `sys/stat.h`, `sys/types.h`.

mkfifo, mkfifoat - create FIFO file

Syntax #include <sys/stat.h>

Optional
#include <sys/types.h> ❑

int mkfifo(const char ***path*, mode_t *mode*);
int mkfifoat(int *fd*, 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`).

The `mkfifoat()` function is equivalent to the `mkfifo()` function except when the *path* parameter specifies a relative path. In this case the new FIFO device file is not created in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

When the value `AT_FDCWD` is transferred to the `mkfifoat()` function for the *fd* parameter, the current directory is used.

Return val. 0 if successful.

-1 if no FIFO was created. `errno` is set to indicate the error.

Errors `mkfifo()` and `mkfifoat()` 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.

In addition, `mkfifoat()` fails if the following applies:

`EACCES` The file descriptor *fd* was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

`ENOTDIR` The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

Notes Bits other than the file permission bits in *mode* are ignored.

path can only be a POSIX file.

See also `umask()`, `fcntl.h`, `sys/stat.h`, `sys/types.h`.

mknod, mknodat - make directory, special file, or text file

Syntax

#include <sys/stat.h>

int mknod(const char **path*, mode_t *mode*, dev_t *dev*);
int mknodat(int *fd*, 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:

`S_IFIFO`	FIFO special file
`S_IFCHR`	Character-special file (not portable)
`S_IFDIR`	Directory (not portable)
`S_IFBLK`	Block-special file (not portable)
`S_IFPOSIXBS2`	File in the POSIX file system (not portable)
`S_IFREG`	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
`S_ISUID`	04000	Set user ID on execution
`S_ISGID`	020#0	Set group ID on execution
`S_IRWXU`	00700	Read, write or execute (search if a directory is involved) by owner
`S_IRUSR`	00400	Read by owner
`S_IWUSR`	00200	Write by owner
`S_IXUSR`	00100	Execute by owner (search if a directory is involved)
`S_IRWXG`	00070	Read, write or execute (search) by group
`S_IRGRP`	00040	Read by group
`S_IWGRP`	00020	Write by group
`S_IXGRP`	00010	Execute (search) by group
`S_IRWXO`	00007	Read, write or execute (search) by others
`S_IROTH`	00004	Read by others

Symbolic name	Bit pattern	Meaning
S_IWOTH	00002	Write by others
S_IXOTH	00001	Execute by others
S_ISVTX	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).

The mknodat() function is equivalent to the mknod() function except when the *path* parameter specifies a relative path. In this case the new directory or the new file is not created in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without O_SEARCH, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with O_SEARCH, the check is not performed.

When the value AT_FDCWD is transferred to the mknodat() function for the *fd* parameter, the current directory is used.

Return val. 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() and mknodat() 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.

In addition, `mknodat()` fails if the following applies:

EACCES
: The file descriptor *fd* was not opened with O_SEARCH, and the authorizations applicable for the directory do not permit the directory to be searched.

EBADF
: The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value AT_FDCWD, nor does it contain a valid file descriptor opened for reading or searching.

ENOTDIR
: The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

Notes
: `mknod()` and `mknodat()` are 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 val.

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 val. 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, mktime64 - convert local time into time since the Epoch

Syntax

#include <time.h>

time_t mktime(struct tm **timeptr*);
time64_t mktime64(struct tm **timeptr*);

Description

The `mktime()` and `mktime64()` functions convert the date and time of the local time which are specified in a structure of the type `tm` into the number of seconds which have passed since 1.1.1970 00:00:00 hrs UTC (Universal Time Coordinated).

The two functions differ merely in the range of dates which can be displayed:

– `mktime()`:
  13.12.1901 20:45:52 hrs UTC through 19.1.2038 03:14:07 hrs

– `mktime64()`:
  1.1.1900 20:45:52 hrs UTC through 31.12.9999 23:59:59 hrs

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, 1970. ❑

Return val. Number of seconds
if successful.

`(time_t)` - 1 or
`(time64_t)` - 1
if the calendar time cannot be represented. Furthermore, `errno` is set to `EOVERFLOW`.

BS2000
For local times as of January 1, 1970, 00.00.00, the number of seconds that have elapsed since then (positive value).

For local times prior to January 1, 1970, 00.00.00, the number of elapsed seconds up to that point (negative value). ❑

Example Which day of the week was 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:

`PROT_READ`	Page can be read.
`PROT_WRITE`	Page can be written.
`PROT_EXEC`	Page can be executed.
`PROT_NONE`	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:

MAP_SHARED	Changes are shareable
MAP_PRIVATE	Changes are private
MAP_FIXED	*addr* 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 *fildes*. This reference is not deleted when a `close()` is issued for *fildes*, but only when no more mappings to the file exist.

Return val.

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` — *fildes* is not opened for reading, regardless of the specified *prot* argument, or *fildes* 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` — *fildes* is not a valid open file descriptor.

`ENXIO` — Addresses in the [*off, off* + *len*) area are invalid for *fildes*.

`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` — *fildes* 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 *fildes*.

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 val.
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 **datptr*, 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 val.

0	upon successful completion.
-1	if an error occurs. `errno` is set to indicate the error.

Errors `mount()` will fail if:

`EBUSY` *dir* is already mounted at the time of the call, or
dir has some other owner, or
dir is otherwise busy, or
the special file associated with *spec* is currently mounted, or
no more mount table entries are available.

`EFAULT` *spec*, *dir* or *datalen* points outside the allocated address space of the process.

`EINVAL` The super-block has an invalid magic number or *fstyp* is invalid.

`ELOOP` Too many symbolic links were encountered in resolving *dir*.

`ENAMETOOLONG`
The length of the *dir* argument exceeds `PATH_MAX` or `NAME_MAX`.

`ENOENT` One of the named files is not recognized.

`ENOSPC` The file system state in the super-block is not FsOKAY and *mflag* requests write permission.

`ENOTBLK` *spec* is not a block-special file.

`ENOTDIR` A component of *spec* or *dir* is not a directory.

`ENXIO` The special file associated with *spec* is not recognized.

`EPERM` The effective user ID is not that of a process with appropriate privileges.

`EREMOTE` *spec* is not local and cannot be mounted.

`EROFS` *spec* is write-protected, and *mflag* 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" [2].

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:

`PROT_READ`	Page can be read.
`PROT_WRITE`	Page can be written.
`PROT_EXEC`	Page can be executed.
`PROT_NONE`	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 val.

0	if successful.
-1	if an error occurs. `errno` is set to indicate the error.

Errors

Under the conditions described below, the `mprotect()` function will fail and set `errno` to the following values:

`EACCES` *prot* contains a value that does not match the access permissions of the process for the underlying file.

`EAGAIN` *prot* contains the value `PROT_WRITE` for a mapping of type `MAP_PRIVATE` and a memory bottleneck occurs, i.e. the storage resources for reserving and locking the private page are not sufficient.

`EINVAL` *addr* is not a multiple of the page size specified by `sysconf()`, or

the *len* argument contains a value less than or equal to 0.

`ENOMEM` Addresses in the area [*addr, addr + len*) 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 *msqid*, 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 *msqid* 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 `msgqid_ds` data structure associated with *msqid* 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 *msqid*. Only a process with appropriate privileges can raise the value of `msg_qbytes`.

`IPC_RMID` Remove the message queue identifier specified by *msqid* and destroy the message queue and the data structure associated with it. `IPC_RMD` 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 *msqid*.

Return val.

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` *buf* points to an invalid address. ❑

`EINVAL` *msqid* is not a valid message queue identifier,
or *cmd* is not a valid operation,
or *cmd* is `IPC_SET`, and `msg_perm.uid` or `msg_perm.gid` is invalid.

`EPERM` *cmd* 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 `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with *msqid*.

`EPERM` *cmd* is `IPC_SET`, an attempt is being made to increase to the value of `msg_qbytes`, 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 147.

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 val.
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:

`EACCES` A message queue identifier exists for the argument *key*, but the access permissions specified by the low-order 9 bits of *msgflg* are not granted (see section “Interprocess communication” on page 147).

`EEXIST` A message queue identifier exists for the argument *key*, but the value of `((`*msgflg* `& IPC_CREAT) && (`*msgflg* `& IPC_EXCL))` is non-zero.

`ENOENT` A message queue identifier exists for the argument *key* and `(`*msgflg* `& IPC_CREAT)` is 0.

`ENOSPC` 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 147.

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 *msqid*:

– `msg_qnum` is decremented by 1.

– `msg_lrpid` 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 val.

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 *msqid* is removed from the system.
`EINTR`	`msgrcv()` was interrupted by a signal.
`EINVAL`	*msqid* 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 147.

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 val.

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 *msgid* 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 147.

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:

MS_ASYNC	Perform asynchronous write accesses
MS_SYNC	Perform synchronous write accesses
MS_INVALIDATE	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 val.

0	if successful.
-1	if an error occurs. errno 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 val.
0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

Errors
`munmap()` will fail if:

`EINVAL` *addr* is not a multiple of the page size defined by `sysconf()`, or

addresses in the area [*addr, addr + len*) are not valid for the address area of the process, or

the *len* argument contains a value less than or equal to 0.

See also
`mmap()`, `sysconf()`, `signal.h`, `sys/mman.h`.

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 val.

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()` fails 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 val.
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 *path* are traversed. If `FTW_MOUNT` is not set, mounted directories are also traversed.

`FTW_PHYS` The directory hierarchy is physically traversed; `nftw()` does not follow any symbolic links, but reports the links instead.
If `FTW_PHYS` is not set, `nftw()` follows symbolic links. `nftw()` 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 *flags* 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 *flags* does not contain the value `FTW_PHYS`).

   `FTW_DNR` The object is a directory which cannot be read. *fn()* 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 val.

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 ≠ 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 val.

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 val. 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 86 and section “Environment variables” on page 104.

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 #include <stddef.h>

size_t offsetof(*type*, *component*);

Description `offsetof()` returns the offset in bytes between the named structure *component* and the start of the structure of the specified *type.*
`offsetof()` is a macro.

type is the name of the structure type (label).

component is the name of the structure component.

Return val.	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, openat - open file

Name **open, open64, openat, openat64**

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**/);
int openat (int *fd*, const char **path*, int *oflag* , ...);
int openat64 (int *fd*, const char **path*, int *oflag* , ...);

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.

`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 four values (file access modes) below in the value of *oflag*:

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing. The result is undefined if this flag is applied to a FIFO file.

`O_SEARCH` Open directory for searching. The result is undefined if this flag is not applied to a directory.

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 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 it 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
- (SYSTERM), terminal I/O
- (INCORE), temporary binary file that is created in virtual memory only.

mode is a constant defined in the <stdio.h> header which specifies the desired access mode (or the corresponding octal value), namely:

`O_RDONLY`
`0000`
: Open for reading. The file must already exist.

`O_WRONLY`
`0001`
: Open for writing. The file must already exist. The previous contents are retained.

`O_TRUNC|O_WRONLY`
`01001`
: Open for writing. If the file exists, the previous contents are deleted. If the file does not exist, it is created.

`O_RDWR`
`0002`
: Open for reading and writing. The file must already exist. The previous contents are retained.

`O_TRUNC|O_RDWR`
`01002`
: Open for reading and writing. If the file exists, the previous contents are deleted. If the file does not exist, it is created.

`O_WRRD`
`0003`
: Open for writing and reading. If the file exists, the previous contents are deleted. If the file does not exist, it is created.

`O_APPEND_OLD|O_TRUNC|O_WRONLY`
`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 previous contents are preserved and the new text is appended to the end of the file.

`O_APPEND_OLD|O_RDWR`
`0402`
: Open for appending to the end of the file and for reading. The file must already exist. The old contents are preserved and the new text is appended to the end of the file. After it is opened, the file is positioned to the end of the file when KR functionality is being used (applies to C/C++ versions prior to V3.0 only), with ANSI functionality to the start of the file.

lbp switch

The *lbp* switch controls handling of the Last Byte Pointer (LBP). It is only relevant for binary files with PAM access mode and can be combined with all specifications permissible for `open`. If `O_LBP` is specified as the *lbp* switch, a check is made to see whether LBP support is possible. If this is not the case, the `creat()`, `creat64()` function will fail and `errno` is set to `ENOSYS`. The switch has further effects only when the file is closed.

When an existing file is opened and read, the LBP is always taken into account independently of the *lbp* switch:

– If the file's LBP is not equal to 0, it is evaluated. Any marker which is present is ignored.

– When LBP = 0, a marker is searched for, and the file length is determined from this. If no marker is found, the end of the last complete block is regarded as the end of file.

`O_LBP`

When a file which has been modified or newly created is closed, no marker is written (even if one was present), and a valid LBP is set. In this way files with a marker can be converted to LBP without a marker.
In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

`O_NOLBP`

When a file which has been modified or **newly created** is closed, the LBP is set to zero (=invalid). A marker is written. In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

When a file which has been **modified** or newly created is closed, the LBP is set to zero (=invalid). If the file had a valid LBP when it was opened, no marker is written as in this case it is assumed that no marker exists.
In the case of NK files the last logical block is padded with binary zeros, in the case of K files the file is padded to the physical end of file.

If the *lbp* switch is specified in both variants (`O_LBP` and `O_NOLBP`), the `creat()`, `creat64()` function fails and `errno` is set to `EINVAL`.

If the *lbp* switch is not specified, the behavior depends on the environment variable LAST_BYTE_POINTER (see also section “Environment variables” on page 104):

`LAST_BYTE_POINTER=YES`

The function behaves as if `O_LBP` were specified.

`LAST_BYTE_POINTER=NO`

The function behaves as if `O_NOLBP` were specified.

Nosplit switch

This switch controls the processing of text files with SAM access mode and variable record length when a maximum record length is also specified. It can be combined with any of the other constants.

`O_NOSPLIT`

When reading with `read`, records of maximum length are not concatenated with the following record.
When writing with `write`, records which are longer than the maximum record length are truncated to the maximum record length.

If the switch is not specified, the following applies:

– When writing
A record which is longer than the maximum record length will be split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it.

– When reading
If a record has the maximum record length, it is assumed that the following record is the continuation of this record and the records are concatenated.

The constant `O_RECORD` can be specified in the *modus* parameter to open files with record-oriented input/output (record I/O). It can always be combined with every other constant except `O_LBP`. Only in the case of ISAM files is adding to the end of the file not permitted, i.e. the combination with `0401` and `0402`. With ISAM files the position is determined from the key in the record.

`O_RECORD`

This switch functions as follows:

– In the case of record I/O the `read()` function reads a record (or block) from the current file position. If the number *n* of the characters to be read is greater than the current record length, nevertheless only this record is read. If *n* is less than the current record length, only the first *n* characters are read. The data of the next record is read when the next read access takes place.

– The `write()` function writes a record to the file. In the case of SAM and PAM files the record is written to the current file position. In the case of ISAM files the record is written to the position which corresponds to the key value in the record. If the number *n* of the characters to be written is greater than the maximum record length, only a record with the maximum record length is written. The remaining data is lost. In the case of ISAM files a record is written only if it contains at least a complete key. If in the case of files with a fixed record length *n* is less than the record length, binary zeros are used for padding. When a record is updated in a SAM or PAM file, the length of the record may not be modified. The `write()` function returns the number of actually written characters also in the case of record I/O.

The `openat()` and `openat64()` functions are equivalent to the `open()` and `open64()` functions except when the *path* parameter specifies a relative path. In this case the file to be opened is not opened in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the functions check whether a search is permitted in the connected directory with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

The *oflag* parameter and the optional fourth parameter *fmode* correspond exactly to the parameters of `open()` / `open64()`.

When the value `AT_FDCWD` is transferred to the `openat()` / `openat64()` function for the *fd* parameter, the current directory is used.

Return val. 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()`, `open64()`, `openat()` and `openat64()` 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 `chmod()`).

`EEXIST` `O_CREAT` and `O_EXCL` are set, and the named file exists.

`EFAULT` *path* points beyond the assigned address space of the process.

`EINTR` A signal was caught during the `open()` system call.

`EINVAL` The value of the *oflag* 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 *oflag* includes O_WRONLY or O_RDWR.
EMFILE	{OPEN_MAX} file descriptors are currently open in the calling process.
EMULTIHOP	Components of *path* require hops to several remote computers, but the file system does not permit this.
ENAMETOOLONG	The length of the *path* argument 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	O_CREAT is not set and the named file does not exist, or O_CREAT is set and either the path prefix does not exist or *path* points to an empty string.
ENOLINK	*path* refers to a remote computer to which there is no active connection.
ENOSPC	The file does not exist, and O_CREAT 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 O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set and no process has the file open for reading.
EROFS	The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if file does not exist) or O_TRUNC is set in the *oflag* argument.
EOVERFLOW	O_LARGEFILE is not set for a file and the size of the file cannot be represented correctly in an object of type off_t.

In addition, `openat()` and `openat64()` fail when the following applies:

`EACCES`	The *fd* parameter was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.
`EBADF`	The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.
`ENOTDIR`	The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.
`EINVAL`	The implementation does not support `O_SEARCH` for the POSIX file system `bs2fs`.

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, fdopendir - open directory

Syntax

#include <dirent.h>

Optional
#include <sys/types.h> ❑

DIR *opendir(const char **dirname*);
DIR *fdopendir(int *fd*);

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.

The `fdopendir()` function is equivalent to the `opendir()` function, with the difference that the directory is specified by the file descriptor *fd* instead of by a pathname.

After a successful return from `fdopendir()`, the file descriptor is under system control. If an attempt is made to close the file descriptor or to change the status of the directory by functions other than `closedir()`, `readdir()`, `rewinddir()` or `seekdir()`, the behavior is undefined. `closedir()` also closes the file descriptor.

Return val.

Pointer to a `DIR` object
if successful.

Null pointer
if an error occurs. `errno` is set to indicate the error.

Errors

`opendir()` and `fdopendir()` 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.

In addition, `fdopendir()` fails if the following applies:

EBADF The *fd* parameter contains no valid file descriptor which is opened for reading.

ENOTDIR The file descriptor *fd* is not connected with 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 *name* (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 val. 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. ❑

`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. ❑

`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` *fildes* 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 val.

-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 val. 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.

The message text output can also contain inserts for POSIX error messages.

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" [6]).
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 73.

pipe - create pipe

Syntax

#include <unistd.h>

int pipe(int *fildes*[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 *fildes*[0] and *fildes*[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 *fildes*[1] and read from file descriptor *fildes*[0]. A read on the file descriptor *fildes*[0] accesses the data written to file descriptor *fildes*[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. *fildes*[0]; the same applies to writing and the write end, i.e. *fildes*[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 val.

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*[], nfds_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 `revents` 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 `revents` even if the message has the length 0.

`POLLRDBAND` Data with priority ≠ 0 can be read without blocking. For STREAMS this flag is set in `revents` 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 `revents` 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 ≠ 0 can be written.

`POLLMSG` An `M_SIG` or `M_PCSIG` message containing an `ASIGPOLL` 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 `revents` bit mask; in the `events` 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 val.

Value ≥ 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 `nfds` 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:
`execl (`*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 val. 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" [2].

pow - power function

Syntax

#include <math.h>

double pow(double x, double y);

Description

`pow()` computes the value of xy.

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 val.

Value of xy	if x, y and the result lie in the permitted floating-point interval.
`+/-HUGE_VAL`	(depending on the sign) if an overflow occurs; `errno` is set to indicate the error.
1.0	if x and y are both 0.
`-HUGE_VAL`	if x is 0 and y is less than 0; `errno` is set to indicate the error.
undefined	if x is less than 0 and y is not an integer; `errno` is set to indicate the error.

Errors

`pow()` will fail if:

`EDOM`	The value of x is negative, and y is not an integer.
	The value of x is 0, and y is negative.
`ERANGE`	The value of x 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 val. 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 463 under `getc_unlocked( )` ...) is functionally equivalent to `putc()` except that it's 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 val. 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 110).

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 117). ❑

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 463 under `getc_unlocked( )` ...) is functionally equivalent to `putchar()` except that it's 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 val. 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 110).

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 117.

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 val. 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()`, `setenv()`, `unsetenv()`, `stdlib.h`, section “Environment variables” on page 104.

putmsg, putpmsg - send message to STREAMS file

Syntax

#include <stropts.h>

int putmsg(int *fildes*, const struct strbuf **ctlptr*,
const struct strbuf **dataptr*, int *flags*);

int putpmsg(int *fildes*, 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.

fildes 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 ≥ 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* ≠ 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 val.

0 — if successful.

-1 — if an error occurs. `errno` is set to indicate the error.

Errors

`putmsg()` and `putpmsg()` will fail if:

`EAGAIN` — A message without priority was specified, the `O_NDELAY` or `O_NONBLOCK` flag is set, and the read/write queue of the STREAM is full because of internal control flow conditions
or
no buffer could be allocated for the message to be created.

`EBADF` — *fildes* is not a valid file descriptor open for writing.

`EINTR` — A signal was caught during the `putmsg()` system call.

`EFAULT` — *ctlptr* or *dataptr* point outside the allocated address space.

`EINVAL` An undefined value was specified in *flags*, or
flags is set to `RS_HIPRI` or `MSG_HIPRI` and no control section was provided, or
the STREAM or multiplexer referenced by *fildes* is connected downstream via a multiplexer.

For `putpmsg()` only:
flags is set to `MSG_HIPRI` and *band* ≠ 0.

`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 *fildes*.

`ENXIO` A hang-up was generated downstream for the specified stream.

`EPIPE` or `EIO`
fildes references a STREAM-based pipe and the other end of the pipe is closed. The SIGPIPE signal is generated for the calling process.

`ERANGE` 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.
`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.

`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

<pwd.h> included via: #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 val.

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" [1]).

See also

getpwent() and the manual "POSIX Basics" [1].

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 val. 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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records which are longer than the maximum record length are truncated to the maximum record length when they are written. By default or with the specification `split=yes`, these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

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 117.

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 val. 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 val.

0	if successful.
≠ 0	if an error occurs. The error indicator for the stream is set, and `errno` 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 110).

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 117).

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 val. 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 val. 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 K2
SIGILL	PROCHK	Execution of an invalid instruction
SIGABRT	–	raise signal for program abort with `_exit(-1)`
SIGFPE	PROCHK	Error in a floating-point operation
SIGKILL	–	raise signal for program abort with `exit(-1)`
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 val. 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 val. 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 val. 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 val.

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\0", random());

}
```

See also

`drand48()`, `rand()`, `rand_r()`, `srand()`, `stdlib.h`

read - read bytes from file

Syntax #include <unistd.h>

ssize_t read(int *fildes*, void **buf*, size_t *nbyte*);

Description `read()` reads *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*.

fildes 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 *fildes*. 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 val. 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` *fildes* 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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification O_NOSPLIT was entered for `open`, records of maximum record length are not concatenated with the subsequent record when they are read. By default (i.e. without the specification O_NOSPLIT), when a record with maximum record length is read, it is assumed that the following record is the continuation of this record and the records are concatenated. ❑

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 129.

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 val. 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.

EOVERFLOW 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 val.

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, readlinkat - read contents of symbolic link

Syntax

#include <unistd.h>

int readlink(const char **path*, char **buf*, size_t *bufsize*);
int readlinkat(int *fd*, const char **path*, char **buf*, size_t *bufsize*);

Description

`readlink()` places the contents of the symbolic link referred to by *path* in the buffer *buf*, which has size *bufsize*. The contents of the link are not terminated with a null byte when returned.

The `readlinkat()` function is equivalent to the `readlink()` function except when the *path* parameter specifies a relative path. In this case the symbolic link whose content is to be read is not searched for in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

When the value `AT_FDCWD` was transferred to the `readlinkat()` function for the *fd* parameter, the current directory is used.

Return val.

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()` and `readlinkat()` 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.

In addition, `readlinkat()` fails if the following applies:

`EACCES`	The file descriptor *fd* was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.
`EBADF`	The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.
`ENOTDIR`	The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

Notes `readlink()` will only access POSIX files.

See also `stat()`, `symlink()`, `fcntl.h`, `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 < *iovcnt* ≤ `{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 val.

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. `errno is set to indicate the error. The contents of the buffer are undefined.`

Errors

`readv()` will fail if:

`EAGAIN`	The `O_NONBLOCK` 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 `O_NONBLOCK` is set, or there is no message in a stream waiting to be read, and `O_NONBLOCK` is set. ❑
`EBADF`	*fildes* 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`	*iov* 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 *iov-len* values in the *iov* field caused a `ssize_t` overflow or *iovcnt* ≤ 0 or *iovcnt* >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` *fildes* describes a directory that cannot be read with `readv()`. `readdir()` should be used instead.

`ENXIO` A request was made for a non-existent device
or the request exceeded the capabilities of the device.

`ENOLINK` *fildes* 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 val.

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 val. 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 nth 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 val. 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 `regexec()` functions are recommended instead of the ones described here.

See also `regcmp()`, `regexec()`, `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 val. 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()`.

regcomp, regexec, regerror, regfree - interpret regular expression

Syntax

```
#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);
int regexec(const regex_t *preg, cont char *string, size_t nmatch, regmatch_t pmatch[],
            int eflags);
size_t regerror(int errcode, const regex_t *preg,char *errbuf, size_t errbuf_size);
void regfree(regex_t *preg);
```

Description

These functions interpret basic and extended regular expressions as described in the XBD specification, Chapter 7, Regular Expressions.

The structure type `regex_t` contains at least the following member:

`size_t re_nsub`	Number of parenthesised subexpressions.

The structure type `regmatch_t` contains at least the following members:

`regoff_t rm_so`	Byte offset from start of string to start of substring.
`regoff_t rm_eo`	Byte offset from start of string of the first character after the end of substring.

The `regcomp()` function compiles the regular expression contained in the string pointed to by the *pattern* argument and places the results in the structure pointed to by *preg*.

The *cflags* argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in the header `regex.h`:

`REG_EXTENDED`
Use Extended Regular Expressions.

`REG_ICASE` Ignore case in match.

`REG_NOSUB` Report only success/fail in `regexec()`.

`REG_NEWLINE` Change the handling of newline characters, as described in the text.

The default regular expression type for *pattern* is a Basic Regular Expression. The application can specify Extended Regular Expressions using the `REG_EXTENDED` flag in the *cflags* argument.

On successful completion, it returns 0; otherwise it returns non-zero, and the content of *preg* is undefined.

If the `REG_NOSUB` flag was not set in *cflags*, then `regcomp()` will set *re_nsub* to the number of parenthesised subexpressions (delimited by \(\) in basic regular expressions or ( ) in extended regular expressions) found in *pattern*.

The `regexec()` function compares the null-terminated string specified by *string* with the compiled regular expression *preg* initialised by a previous call to `regcomp()`. If it finds a match, `regexec()` returns 0; otherwise it returns non- zero indicating either no match or an error. The *eflags* argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in the header `regex.h`:

`REG_NOTBOL` The first character of the string pointed to by string is not the beginning of the line. Therefore, the circumflex character (ˆ), when taken as a special character, will not match the beginning of string.

`NEG_NOTEOL` The last character of the string pointed to by string is not the end of the line. Therefore, the dollar sign ($), when taken as a special character, will not match the end of string.

If *nmatch* is 0 or `REG_NOSUB` was set in the *cflags* argument to `regcomp()`, then `regexec()` will ignore the *pmatch* argument. Otherwise, the *pmatch* argument must point to an array with at least *nmatch* elements, and `regexec()` will fill in the elements of that array with offsets of the substrings of *string* that correspond to the parenthesised subexpressions of *pattern*: *pmatch*[*i*].*rm_so* will be the byte offset of the beginning and *pmatch*[*i*].*rm_eo* will be one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open parenthesis, counting from 1.) Offsets in *pmatch*[0] identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*[*nmatch*-1] will be filled with -1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), then `regexec()` will still do the match, but will record only the first *nmatch* substrings.

When matching a basic or extended regular expression, any given parenthesised subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring even though the pattern as a whole did match.

The following rules are used to determine which substrings to report in *pmatch* when matching regular expressions:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch*[*i*] will delimit the last such match.
2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in *pmatch*[*i*] will be -1.

   A subexpression does not participate in the match when:

   – * or \{ \} appears immediately after the subexpression in a basic regular expression, or *, ?, or { } appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched 0 times)

   or:

   – **|** is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], then the match or non-match of subexpression *i* in *pmatch*[*i*] will be reported as described in 1. and 2. above, but within the substring reported in *pmatch*[*j*] rather than the whole string.
4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are -1, then the pointers in *pmatch*[*i*] also will be -1.
5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] will be the byte offset of the character or null terminator immediately following the zero-length string.

If, when `regexec()` is called, the locale is different from when the regular expression was compiled, the result is undefined.

If `REG_NEWLINE` is not set in *cflags*, then a newline character in pattern or string will be treated as an ordinary character.

If `REG_NEWLINE` is set, then newline will be treated as an ordinary character except as follows:

1. A newline character in string will not be matched by a period outside a bracket expression or by any form of a non-matching list (see the XBD specification, Chapter 7, Regular Expressions).
2. A circumflex (ˆ) in *pattern*, when used to specify expression anchoring, will match the zero-length string immediately after a newline in *string*, regardless of the setting of `REG_NOTBOL`.
3. A dollar-sign ($) in *pattern*, when used to specify expression anchoring, will match the zero-length string immediately before a newline in *string*, regardless of the setting of `REG_NOTEOL`.

The `regfree()` function frees any memory allocated by `regcomp()` associated with *preg*.

The following constants are defined as error return values:

`REG_NOMATCH` `regexec()` failed to match.

`REG_BADPAT` Invalid regular expression.

`REG_ECOLLATE` Invalid collating element referenced.

`REG_ECTYPE` Invalid character *class* type referenced.

`REG_EESCAPE` Trailing \ in pattern.

`REG_ESUBREG` Number in *digit* invalid or in error.

`REG_EBRACK`	[ ] imbalance.
`REG_ENOSYS`	The function is not supported.
`REG_EEPAREN`	\(\) or ( ) imbalance.
`REG_EBRACE`	{ \} imbalance.
`REG_BADBR`	Content of \{ \} invalid: not a number, number too large, more than two numbers, first larger than second.
`REG_ERANGE`	Invalid endpoint in range expression.
`REG_ESPACE`	Out of memory.
`REG_BADRPT`	?, * or + not preceded by valid regular expression.

The `regerror()` function provides a mapping from error codes returned by `regcomp()` and `regexec()` to unspecified printable strings. The generated string corresponds to the value of the *errcode* argument, which must be the last non-zero value returned by `regcomp()` or `regexec()` with the given value of *preg*. If `errcode` is not such a value, the content of the generated string is unspecified.

If *preg* is a null pointer, but `errcode` is a value returned by a previous call to `regexec()` or `regcomp()`, `regerror()` still generates an error string corresponding to the value of `errcode`, but it might not be as detailed.

If the *errbuf_size* argument is not 0, `regerror()` will place the generated string into the buffer with the size of *errbuf_size* bytes pointed to by *errbuf*. If the string including the terminating null cannot fit in the buffer, `regerror()` will truncate the string and terminate the result by null.

If *errbuf_size* is 0, `regerror()` ignores the *errbuf* argument, and returns the size of the buffer needed to hold the generated string.

If the *preg* argument to `regexec()` or `regfree()` is not a compiled regular expression returned by `regcomp( )`, the result is undefined. A *preg* is no longer treated as a compiled regular expression after it is given to `regfree()`.

Return val. for `regcomp()`:

0	if successful.

Integer value indicating an error as described in `regex.h`, and the content of *preg* is undefined.

for `regexec()`:

0	if successful.
`REG_NOMATCH`	if no match has been found
`REG_ENOSYS`	if the function is not implemented.

for `regerror()`:

Number of bytes needed to hold the entire generated string
if successful.

0	if the function is not implemented.

for `regfree()`:

The function returns no value.

Errors No errors are defined.

Example 1

```
#include <regex.h>

/*
* Match string against the extended regular expression in
* pattern, treating errors as no match.
*
* return 1 for match, 0 for no match
*/

int
match(const char *string, char *pattern)
{
int status;
regex_t re;
if (regcomp(&re, pattern, REG_EXTENDED | REG_NOSUB) != 0) {
return(0); /* report error */
}
status = regexec(&re, string, (size_t) 0, NULL, 0);
regfree(&re);
if (status != 0) {
return(0); /* report error */
}
return(1);
}
```

Example 2 The following demonstrates how the `REG_NOTBOL` flag could be used with `regexec()` to find all substrings in a line that match a *pattern* supplied by a user.

For simplicity of the example, very little error checking is done.

```
(void) regcom (&re, pattern, 0);
/* Dieser Aufruf von regexec() findet die erste Uebereinstimmung in der
* Zeile.
*/
error = regexec (&re, &buffer[0], 1, pm, 0);
while (error == 0) { /* Solange eine Uebereinstimmung gefunden wird */
/* Eine Teilzeichenkette wurde gefunden zwischen pm.rm_so und
* pm.rem_eo.
* Dieser Aufruf von regexec() findet die naechste
* Uebereinstimmung.
*/
error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
}
```

Notes An application could use

```
regerror(code,preg,(char *)NULL,(size_t)0)
```

to find out how big a buffer is needed for the generated string, `malloc()` a buffer to hold the string, and then call `regerror()` again to get the string.
Alternatively, it could allocate a fixed, static buffer that is big enough to hold most strings, and then use `malloc()` to allocate a larger buffer if it finds that this is too small.

See also `fnmatch()`, `glob()`, `regex.h`, `sys/types.h`

regexp: 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 <regexp.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 `regexp.h`.

Programs must have the following five macros declared before the `#include <regexp.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 `regexp.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
r+	One or more occurrences of the regular expression *r*. *r* must be in one of the following forms: *r*, *r*, any character, `[`*r*`]`, `[`*r1*−*r2*`]`, `[^`*s*`]`, `[^`*r1*−*r2*`]`, `(`*r*`)`, `(`*r1*`|`*r2*`)`	`u+`	`u, uu, uuu, ...`
r?	Zero or one occurrence of the redular expression *r*. *r* must be in one of the following forms: *r*, *r*, any character, `[`*r*`]`, `[`*r1*−*r2*`]`, `[^`*s*`]`, `[^`*r1*−*r2*`]`, `(`*r*`)`, `(`*r1*`|`*r2*`)`	`u?`	none or `u`
`(`*r*`)`	Strings matching regular expression *r*. *r* can be any expression.	`(ok(abc))` `(au)*`	okabc none or aus, auau, ...
`(`*r1*`|`*r2*`)`	Strings matching regular expression *r1* or *r2*.	`(ok|ko)`	`ok` or `ko`

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 `regexp` functions are constructed as follows:

Expression	Meaning
c	The character *c*, where *c* must not be a special character.
c	The character *c*, where *c* is any character other than a digit in the range `1-9`.
^	The beginning of the line being compared.
$	The end of the line being compared.
.	Any character in the input.
[*s*]	Any character in the set *s*, where *s* is a sequence of characters. Ranges may be specified as [*c*-*c*]. The character] may be included in the set only in the first position; the character - may be included only in the first or last position, and the character ^ may be included by placing it anywhere other than first position in the set. Ranges in SREs are only valid if the `LC_COLLATE` category is set to the C locale.
[^*s*]	Any character not in the set *s* , where *s* is defined as above.
*r**	Zero or more successive occurrences of the regular expression *r*. The longest leftmost matching string is used.
rx	The occurrence of regular expression *r* followed by the occurrence of regular expression *x* (concatenation).
r\\{*m*,*n*\\}	Any number of *m* through *n* successive occurrences of the regular expression *r*. The regular expression *r*\\{*m*\\} matches exactly *m* occurrences; *r*\\{*m*,\\} matches at least *m* occurrences. The maximum number of occurrences is matched.
\\(*r*\\)	The regular expression *r*. The \\(and \\) sequences are ignored.
n	When *n* is a number in the range 1-9 and appears in a concatenated regular expression, it stands for the regular expression *x*, where *x* is the *n*-th regular expression enclosed in \\(and \\) sequences that appeared earlier in the concatenated regular expression. For example, in the pattern \\(*r*\\)*x*\\(*y* the *2* matches the regular expression *y*, giving *rxyzy*.

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" [2])

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

punct	a punctuation character
print	a printing character
graph	a character with a visible representation
cntrl	a control character
[[=*c*=]]	An equivalence class. Any collation element defined as having the same relative order in the current collation sequence as *c*. As an example, if `A` and `a` belong to the same equivalence class, then both `[[=`*A*`=]`*b*`]` and `[[=`*a*`=]`*b*`]` are equivalent to `[`*Aab*`]`.
[[.*cc*.]]	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 *ch* is a valid collating element, then `[[.`*ch*`.]]` will be treated as an element matching the same string of characters, while *ch* will be treated as a simple list of *c* and *h*. If the string *ch* is not a valid collating element in the current collating sequence definition, the symbol will be treated as an invalid expression.
[*c*-*c*]	Any collation element in the character expression range *c*-*c*, where *c* can identify a collating symbol or an equivalence class. If the character − appears immediately after an opening square bracket, for example, `[-`*c*`]`, or immediately prior to a closing square bracket, for example, `[`*c*`-]`, it has no special meaning.
^	Immediately following an opening square bracket, means the complement of, for example, `[^`*c*`]`. 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

`ab.d`	*ab* any character *d*
`ab.*d`	*ab* any sequence of characters (including none) *d*
`ab[xyz]d`	*ab* one of the characters *x y* or *z d*
`ab[^c]d`	*ab* any character, except *c d*
`^abcd$`	a line containing only *abcd*
`a-d`	any one of the characters *a b c* or *d*

Return val.	`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`, `regexp.h`, and the manual "POSIX Commands" [2].

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 ≠ 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 val. Floating-point remainder = x-ny if y ≠ 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 val. 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, renameat - rename file

Syntax #include <stdio.h>

int rename(const char *old, const char *new);
int renameat(int *oldfd*, const char **old*, int *newfd*, 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. ❑

The `renameat()` function is equivalent to the `rename()` function except when the *old* or *new* parameter specifies a relative path. If *old* specifies a relative pathname, the file which is to be renamed is searched for not in the current directory, but in that connected with the file descriptor *oldfd*. If *new* specifies a relative pathname, the same happens relative to the directory connected with the file descriptor *newfd*. If a file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

When the value `AT_FDCWD` is transferred to the `renameat()` function for the *oldfd* or *newfd* parameter, the current directory for determining the file of the corresponding path is used.

Return val.

0 if successful

-1 if an error occurs; `errno` is set to indicate the error. Neither the file named by *old* nor the file named by *new* 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()` and `renameat()` 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 *old* or *new* exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOENT — The link named by *old* does not name an existing file, or either *old* or *new* points to an empty string.

ENOSPC — The directory that would contain *new* cannot be extended.

ENOTDIR — A component of either path is not a directory, or the *old* argument names a directory, and the *new* 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 *new* and *old* are on different file systems.

In addition, `renameat()` fails if the following applies:

EACCES — The file descriptor *oldfd* or *newfd* was not opened with O_SEARCH, and the authorizations applicable for the directory do not permit the directory to be searched.

EBADF — The *old* parameter does not specify an absolute pathname, and the *oldfd* parameter does not have the value AT_FDCWD, nor does it contain a valid file descriptor opened for reading or searching,
or
the *new* parameter does not specify an absolute pathname, and the *newfd* parameter does not have the value AT_FDCWD, nor does it contain a valid file descriptor for reading or searching.

ENOTDIR — The *old* or *new* parameter does not specify an absolute pathname, and the corresponding file descriptor *oldfd* / *newfd* is not connected with a directory.

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()`, `fcntl.h`, `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 val. 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 `strrchr()` 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 val.

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 val. 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 val.

Integer value	represented as a number of type `double`, `float` or `long double` if successful.
undefined	if an overflow occurs. `errno` is set to `ERANGE` 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 val.

$x * r^n$	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  *execptfds, 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 *execptfds* 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 *execptfds* can be specified as null pointers if none of the descriptors is of interest.

Return val.	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: *t_sec* must be between 0 and 10 inclusive. *t_usec* 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 147). 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 147).

`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 val. 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 147).

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 *cmd* 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 sem_perm.cuid or sem_perm.uid in the data structure associated with *semid*.

ERANGE *cmd* is equal to SETVAL or SETALL and the value to which semval 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;
```

See also semget(), semop(), sys/sem.h, section “Interprocess communication” on page 147.

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 val.

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 *nsems* 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 *key*, but the corresponding semaphore set contains less than *nsems* semaphores and *nsems* is not equal to 0.

`ENOENT`	No semaphore ID exists for *key* and `(`*semflg* `& 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.

See also `semctl()`, `semop()`, `sys/sem.h`, section “Interprocess communication” on page 147.

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`	`sem_num`	Semaphore number
`short`	`sem_op`	Semaphore operation
`short`	`sem_flg`	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 i`s 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 val. 0 if successful.

-1 if unsuccessful. `errno` is set to indicate the error.

Errors `semop()` will fail if:

`E2BIG` The value of *nsops* 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 161).

`EAGAIN` The operation would result in suspension of the calling process, but (`sem_flg & IPC_NOWAIT`) is non-zero.

EFBIG	The value of sem_num is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.
EIDRM	The semaphore identifier *semid* was removed from the system.
EINTR	semop() was interrupted by a signal.
EINVAL	The value of *semid* is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the system-imposed limit.
ENOSPC	The system-specific limit on the maximum number of individual processes requesting a SEM_UNDO would be exceeded.
ERANGE	An operation would cause a semval or a semadj to exceed the maximum value for the system.

See also exec, exit(), fork(), semctl(), semget(), sys/sem.h, section “Interprocess communication” on page 147.

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 110.

setcontext - modify user context

Syntax #include <ucontext.h>

int setcontext(const ucontext_t *ucp);

Description See `getcontext()`

setenv - add or change environment variable

Syntax #include <stdlib.h>

int setenv (const char **envname*, const char **envval*, int *overwrite*);

The `setenv()` function updates or adds a variable in the environment of the calling process.

The *envname* argument points to a string containing the name of an environment variable to be added or altered. If the environment variable already exists, two cases must be distinguished: If the value of *overwrite* is not zero, the environment is changed; if the value is zero, the environment remains unchanged. In both cases the function is terminated successfully.

If the application modifies *environ* or the pointers to which it points, the behavior of `setenv` is undefined. The `setenv` function updates the list of pointers to which *environ* points.

The strings described by *envname* and *envval* are copied by this function.

`setenv()` is not thread-safe.

Return val.

0	if successful.
-1	otherwise. `errno` is set to indicate the error. The environment remains unchanged.

Errors `setenv()` will fail if:

`EINVAL`	The *envname* argument is a null pointer, points to an empty string, or points to a string containing an '=' character.
`ENOMEM`	Insufficient memory was available to add a variable or its value to the environment.

See also `environ`, `exec`, `getenv()`, `malloc()`, `putenv()`, `unsetenv()`, `stdlib.h`, section “Environment variables” on page 104.

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 val.

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()`.

setgroups - write group numbers

Syntax #include <unistd.h>

int setgroups(int *ngroups*, const gid_t *grouplist*[]);

Description The `setgroups()` function can only be called by the system administrator. The `setgroups()` function sets the group access list of the calling process from the group numbers field. The number of entries is specified by the *ngroups* parameter and must not exceed `NGROUPS_MAX`.

Return val.

0	if successful.
-1	if unsuccessful. `errno` indicates the cause of the error.

Errors `setgroups()` will fail if:

`EINVAL`	The value *ngroups* exceeds `NGROUPS_MAX`.
`EFAULT`	A referenced part of the grouplist array is outside the address range assigned to the process.
`EPERM`	The effective user number is not the user number of the system administrator.

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 val.	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 86).

BS2000
The locale component `LC_MESSAGES` is not supported for BS2000 functionality (see section “Scope of the supported C library” on page 49). ❑

`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 49). ❑

`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'C0'` and `X'D0'` 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_V2TYPE	"V2CTYPE"
LC_C_GERMANY	"GERMANY"
LC_C_DeEDF04F	"De.EDF04F"
LC_C_DeEDF04F@euro	"De.EDF04F@euro"

❑

Return val. 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 86).

See also `catopen()`, `ctime()`, `ctype()`, `environ`, `exec`, `getdate()`, `gettxt()`, `isalnum()`, `isalpha()`, `iscntrl()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `localeconv()`, `mblen()`, `mbstowcs()`, `mbtowc()`, `nl_langinfo()`, `printf()`, `scanf()`, `strcoll()`, `strerror()`, `strfmon()`, `strtime()`, `strtod()`, `strxrfm()`, `tolower()`, `toupper()`, `towlower()`, `towupper()`, `wcscoll()`, `wcstod()`, `wcstombs()`, `wcsxfrm()`, `wctomb()`, `langinfo.h`, `locale.h`, section “Locale” on page 86.

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 i*s equal to *pid,* the process becomes a process group leader. If *pgid i*s not equal *to p*id, 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 val.

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 val. `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*, grid_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 val.

0	if executed successfully
-1	if an error occurs. `errno` is set to indicate the error

Errors `setregid()` will fail if:

`EINVAL`	The value of *rgid* or *egid* 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 val.

0	if executed successfully
-1	if an error occurs. `errno` is set to indicate the error

Errors

`setreuid()` will fail if:

`EINVAL`	The value of the *ruid* or *euid* 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 val.

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" [1] and the manual "POSIX Commands" [2]).

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 val.

0	if successful.
–1	if unsuccessful. `errno` is set to indicate the error.

Errors `setuid()` will fail if:

`EPERM`	The process does not have appropriate privileges and *uid* 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:

`_IOFBF`	Full buffering of input and output
`_IOLBF`	Line buffering
`_IONBF`	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 val.

0	if successful.
≠ 0	if an invalid value was specified for *type* or if the request cannot be satisfied. `errno` is set to indicate the error.

Errors

`setvbuf()` will fail if:

`EBADF`	The file descriptor underlying *stream* 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 110.

shmat - shared memory attach operation

Syntax #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 val. 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 *shmid* is not a valid shared memory ID, or
the value of *shmaddr* 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 *shmaddr* is not equal to 0, (*shmflg* `& SHM_RND`) is equal to 0 and the value of *shmaddr* 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 “Interprocess communication” on page 147.

shmctl - shared memory control operations

Syntax #include <sys/shm.h>

int shmctl(int *shmid*, int *cmd*, struct shmid_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 `shmid_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 `shmid_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 `shmid_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 `shmid_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 `shmid_ds` data structure associated with *shmid*.

Return val. 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*.

See also

`shmat()`, `shmdt()`, `shmget()`, `sys/shm.h`, section “Interprocess communication” on page 147.

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 val.

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.

See also

`exec`, `exit()`, `fork()`, `shmat()`, `shmctl()`, `shmget()`, `sys/shm.h`, section “Inter-process communication” on page 147.

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 val.

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:

`EACCES` A shared memory ID for the argument *key* exists, but the permissions specified in the 9 low-order bits of *shmflg* were not granted.

`EEXIST` A shared memory ID exists for the argument *key*, but ((*shmflg* `&` `IPC_CREAT`) `&&` (*shmflg* `&` `IPC_EXCL`)) is not equal to 0.

`EINVAL` The value of *size* is less than the system-imposed minimum or greater than the system-imposed maximum, or
a shared memory identifier exists for the argument *key*, but the size of the segment associated with it is less than *size* and *size* is not 0.

`ENOENT` A shared memory identifier does not exist for *key* 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.

Note *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 147.

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**
void(*)(int)	sa_handler	SIG_DFL, SIG_IGN or a pointer to a signal-handling function.
sigset_t	sa_mask	Additional set of signals to be blocked during execution of the signal-handling function.
int	sa_flags	Special flags that can be used to affect the behavior of *sig*.

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 the 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 823). 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:

`access()`	`free()`	`raise()`	`sysconf()`
`alarm()`	`fstat()`	`read()`	`tcdrain()`
`calloc()`	`getegid()`	`rename()`	`tcflow()`
`cfgetispeed()`	`geteuid()`	`rmdir()`	`tcflush()`
`cfgetospeed()`	`getgid()`	`setgid()`	`tcgetattr()`
`cfsetispeed()`	`getgroups()`	`setpgid()`	`tcgetpgrp()`
`cfsetospeed()`	`getpgrp()`	`setsid()`	`tcsendbreak()`
`chdir()`	`getpid()`	`setuid()`	`tcsetattr()`
`chmod()`	`getppid()`	`sigaction()`	`tcsetpgrp()`
`chown()`	`getuid()`	`sigaddset()`	`time()`
`close()`	`kill()`	`sigdelset()`	`times()`
`creat()`	`link()`	`sigemptyset()`	`umask()`
`dup2()`	`lseek()`	`sigfillset()`	`uname()`
`dup()`	`malloc()`	`sigismember()`	`unlink()`
`execle()`	`mkdir()`	`signal()`	`utime()`
`execve()`	`mkfifo()`	`sigpending()`	`wait()`
`_exit()`	`open()`	`sigprocmask()`	`waitpid()`
`fcntl()`	`pathconf()`	`sigsuspend()`	`write()`
`fork()`	`pause()`	`sleep()`	
`fpathconf()`	`pipe()`	`stat()`	

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:

`catclose()`	`fgetwc()`	`getgrnam()`	`tcdrain()`
`catgets()`	`fopen()`	`getpass()`	`tcsetattr()`
`close()`	`fputc()`	`getpwnam()`	`tmpfile()`
`dup()`	`fputwc()`	`getpwuid()`	`wait()`
`fclose()`	`freopen()`	`open()`	`write()`
`fcntl()`	`fseek()`	`pause()`	
`fflush()`	`fsync()`	`read()`	
`fgetc()`	`getgrgid()`	`sigsuspend()`	

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 val. 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 `sig longjmp()` 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 146.

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 val.

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 val.

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 st*ack (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 val.

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

<signal.h> is included as:

```
#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 val.

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 val. 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. ❑

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 val.

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 val.

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 val.

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 sighold(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 val. 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.

signgam - variable for sign of lgamma

Syntax #include <math.h>

extern int *signgam*;

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 val. 0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

Errors `sigpending()` will fail if:

Extension
`EFAULT` *set* 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 of 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 val.
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 val.

0	on successful return from a direct invocation of `sigsetjmp()`.
≠ 0	if the return is from a call to `siglongjmp()`.

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 146.

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 val.

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 val.

-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 val. 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 val. 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 val. 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`.

snprintf - formatted output to a string

Syntax
#include <stdio.h>

int snprintf(char **s*, size_t *n*, const char **format*, ...);

Description

`snprintf()` edits data (characters, strings, numerical values) according to specifications in the string *format* and writes this data to the area pointed to by *s*.

`snprintf()` only outputs up to the buffer limit specified by the *n* parameter. This prevents buffer overrun. Apart from that the functionality of `snprintf()` is the same as that of `sprintf()`.

`snprintf()` exists, analogous to `sprintf()`, as an ASCII, IEEE and ASCII/IEEE function (cf. sections "IEEE floating-point arithmetic" on page 37 und "ASCII encoding" on page 42).

Parameters

See `fprintf()`.

Return val.

< 0	*n* > `INT_MAX` or output error.
= 0 .. n-1	was possible to edit the output completely. The return value specifies the length of the output without the terminating NULL character.
> n	It was not possible to edit the output completely. The return value specifies the length of the output without the terminating NULL character which a complete output would require.

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 val.

`sqrt`(x)	if x > = 0.
0	if x is negative. `errno` 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 srandom(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:

`st_mode`	The mode of the file is defined in the system call `mknod()`.
`st_ino`	Uniquely identifies the file in a given file system. The pair `st_ino` and `st_dev` uniquely identifies regular files.
`st_dev`	Uniquely identifies the file system that contains the file.
`st_rdev`	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.
`st_nlink`	May be used only by administrative commands.
`st_uid`	The user ID of the file's owner.
`st_gid`	Group ID of the group to which the file is assigned.
`st_size`	For regular files, this is the size of the file in bytes. It is undefined for block special or character special files. For PAM files this member contains the file size. Any existing marker is not considered. If the LBP is zero, the entire last block counts to the size.
`st_atime`	Time when file data was last accessed. Modified by the following system calls: `creat()`, `mknod()`, `utime()` and `read()`.
`st_mtime`	Time when data was last updated. Modified by the following system calls: `creat()`, `mknod()`, `utime()` and `write()`.
`st_ctime`	Time when the file status was last changed. Modified by the following system calls: `chmod()`, `chown()`, `creat()`, `link()`, `mknod()`, `unlink()`, `utime()` and `write()`.

Extension

`st_blksize`	A hint as to the 'best' unit size for I/O operations. This field is not defined for block special or character special files.
`st_blocks`	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:

mode_t *st_mode*	File mode containing the access permissions and file type. 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. File type: introduces a new file type S_IFDVSBS2=X'10000000'. This type, however, is not disjoint to S_IFPOSIXBS2. The S_ISDVSBS2(*mode*) macro can be used for querying. Introduces a new file type S_IFDVSNODE=X'20000000'. This type is also not disjoint to S_IFPOSIXBS2. The S_ISDVSNODE(*mode*) macro can be used for querying. A node file is also a BS2000 DVS file. I.e. for node files the bit `S_IFDVSBS2` is always set.
time_t *st_atime*	Last access time, as is usual in BS2000 but in seconds since 1.1.1970 UTC).
time_t *st_mtime*	Last modification time.
time_t *st_ctime*	Creation time.
long *st_blksize*	Block size, 2K (i.e. 1 PAM page).
long *st_blocks*	Number of blocks occupied by the file on the disk.
dev_t *st_dev*	Contains the 4-byte `catid`.

The two consecutive fields

uid_t *st_uid*	and
gid_t *st_uid*	contain the 8-byte BS2000 user ID.

All other fields are set to 0.

Return val.

0	if successful
-1	if an error occurs. `errno` 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 *path*.
`EMULTIHOP`	Components of *path* require hops to several remote computers, but the file system does not permit this.
`ENAMETOOLONG`	The length of *path* exceeds `{PATH_MAX}` or a pathname component is longer than `{NAME_MAX}` and `{_POSIX_NO_TRUNC}` is in effect.
`ENOLINK`	*path* 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.
`EOVERFLOW`	A component is too large to be stored in the structure pointed to by *buf*.

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 <regexp.h>

int step(const char **string*, const char **exbuf*);

Description See `regexp()`.

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 val. 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 val. 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 val. 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 val. 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 val.
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 val.
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 val.
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 val. 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 161). 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 val.

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.

The message text can also contain inserts:

– If the error number passed in the *errnum* parameter matches the current error number, inserts are taken into account and added to the error message text. The current error number is the one stored in the `errno` variable.

– Otherwise, a message text is returned without inserts, that matches the error number passed in *errnum*.

See also

`perror()`, `popen()`, `errno.h`, `string.h`, section “Error handling” on page 161.

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 - `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 val.
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 val. 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`
	`-$123.45`
	`$3,456.78`
		Default formatting

%11n	`$123.45`
	`-$123.45`
	`$3,456.78`
		Right alignment within an 11-character field

%#5n	`$   123.45`
	`-$   123.45`
	`$ 3,456.78`
		Values through 99.999 are aligned in a column

%= *#5n	`$***123.45`
	`-$***123.45`
	`$*3,456.78`
		Specification of a filler character for free positions

%=0#5n	`$000123.45`
	`-$000123.45`
	`$03,456.78`
		Filler characters are not grouped, even if the filler character is a digit

%^#5n	`$   123.45`
	`-$   123.45`
	`$  3456.78`
		Suppress grouping character

%^#5.0n	`$   123`
	`-$   123`
	`$  3456`
		Round to integer

%^#5.4n	`$   123.4500`
	`-$   123.4500`
	`$  3456.7800`
		Increase right-adjusted precision

%(#5n	`$   123.45`
	`($   123.45)`
	`$  3456.78`
		Alternative representation for positive/negative values

%!(#5n	`123.45`
	`(   123.45)`
	`3456.78`
		Suppress currency symbol

See also `localeconv()`, `monetary.h`.

strftime - convert date and time to string

Syntax

#include <time.h>

size_t strftime(char **s*, size_t *maxsize*, const char **format*, const struct tm **timeptr*);

Description

`strftime()` 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 `strftime()` 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 `strftime()`.

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 `strftime()`, by the contents of *timeptr*:

`%%`	The character %
`%a`	Abbreviated weekday name of the locale
`%A`	Full weekday name of the locale
`%b`	Abbreviated month name of the locale
`%B`	Full month name of the locale
`%c`	Appropriate date and time representation of 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`	Day of the month (1-31; single digits are preceded by a space)
`%f`	Date and time represented in accordance with date()
`%h`	Abbreviated month name of the locale
`%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`	Equivalent to \n
`%p`	Locale´s equivalent of either AM or PM
`%r`	Time in the form %I:%M:%S [AMŠPM]
`%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 ccyy (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. 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`	Day of month, using the locale's alternative numeric symbols, padded as needed with leading spaces.
`%OH`	The hour (24-hour clock), using the locale's alternative numeric symbols.
`%OI`	The hour (12-hour clock), using the locale's alternative numeric symbols.
`%Om`	The month, using the locale's alternative numeric symbols.
`%OM`	The minutes, using the locale's alternative numeric symbols.
`%OS`	The seconds, using the locale's alternative numeric symbols.
`%Ou`	The weekday as a number in the locale's alternative representation (Monday = 1).
`%OU`	The week number of the year (Sunday 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.

The default language for the output of `strftime()` is U.S. English. The user can select the output language for `strftime()` by using `setlocale()` to set the `LC_TIME` category for the locale.

The timezone is taken from the environment variable `TZ` (see `ctime()`).

Return val. 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 val. 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 val. 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 val. 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 val. 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 val.

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`.

strnlen - determine length of a string up to a maximum length

Syntax #include <string.h>

size_t strnlen(const char **s*, size_t *maxlen*);

Description The `strnlen()` function calculates the minimum of the two following values:

– Number of bytes of the array to which *s* points, exclusively to the terminating `NULL` byte
– Value of the *maxlen* parameter.

The `strnlen()` function never checks more than *maxlen* bytes.

Errors No errors are defined.

Return val. Length of the string *s* or value of the *maxlen*
parameter when successful. The terminating null byte is not counted.

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 val. 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. With conversion specifications that expect a numeric value, the string to be converted may contain not more digits than specified in the format description. I.e. additional leading zeroes are not allowed. If between two conversion specifications there is neither a white-space character nor an non-alphanumeric character, the numbers of digits even must be the same as in the format description.

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` *ccyy* `(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. 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 val. 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()`.

The structure to which *tm* points is not initialized with zeros when `strptime()` is executed. The values set by the user remain intact as long as they are not modified by conversion statements or implicit calculations. The structure element *tm_isdst* is never changed. Date adjustment may be carried out implicitly, i.e. if the date entry is incomplete, the missing structure elements are added and a plausibility check is made between the structure elements.

However, this is only made if a week number was specified via `%U`, `%W`, `%OU` or `%OW`. In this case, the year entry (*tm_year*) and weekday (*tm_wday*) are used to calculate and reassign the day in the year (*tm_yday*), the day of the month (*tm_mday*) and the month of the year (*tm_mon*). The weekday is assigned the value 0 if it was not explicitly specified with `%w`, `%a`, `%A` or `%Ow`.

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 val. 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 val. 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 val.

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:

$$[\left\{\begin{matrix} tab \\ \sqcup \end{matrix}\right\}\texttt{...}]\,[\left\{\begin{matrix} + \\ - \end{matrix}\right\}]\,[digit\texttt{...}]\,[\texttt{.}]\,[digit\texttt{...}]\,[\left\{\begin{matrix} \mathrm{E} \\ \mathrm{e} \end{matrix}\right\}\,[\left\{\begin{matrix} + \\ - \end{matrix}\right\}]\,digit\texttt{...}]$$

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 val.
Floating-point number of type `double`
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.
`errno` 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 val. 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 val. 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:

$$[\left\{\begin{matrix} \mathit{tab} \\ \text{␣} \end{matrix}\right\}\ldots][\left\{\begin{matrix} + \\ - \end{matrix}\right\}][\left\{\begin{matrix} 0 \\ 0X \end{matrix}\right\}]\mathit{digit}\ldots$$

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 val. 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`.

strtoll - 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:

$$[\begin{Bmatrix} tab \\ \sqcup \end{Bmatrix}\ldots][\begin{Bmatrix} + \\ - \end{Bmatrix}][\begin{Bmatrix} 0 \\ 0X \end{Bmatrix}]\,digit\ldots$$

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 val. 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, `strtoll()` differs from the `atoll()` function only in the error handling:

`atoll(s)` is equivalent to `strtoll(s, (char **)NULL, 10)`.

See also `atol()`, `atoll()`, `atoi()`, `strtol()`, `stroul()`, `stroull()`, `wcstol()`, `wcstoll()`, `wcstoul()`, `wcstoull()`

strtoul - convert string to unsigned long integer

Syntax

#include <stdlib.h>

unsigned long int strtoul(const char *s, char **$endptr$, int $base$);

Description

`strtoul()` 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:

$$[\left\{\begin{matrix} tab \\ \\ \sqcup \end{matrix}\right\}\ldots]\,[\left\{\begin{matrix} 0 \\ \\ 0X \end{matrix}\right\}]\,digit\ldots$$

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*.

`strtoul()` also recognizes strings that begin with convertible digits (including octal and hexadecimal digits) but end with some other characters. In such cases, `strtoul()` truncates the numeric part before converting it.

`strtoul()` 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 val. 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 *entptr*, 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:

```
 ⎡tab⎤     ⎡0 ⎤
[{    }...][{   }]digit...
 ⎣ ␣ ⎦     ⎣0X⎦
```

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.

`strtoul()` 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 val. 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 val. 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 val.

Length	of the transformed string (excluding the terminating null byte) if successful.
Value ≥ *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 `fwscanf()`.

symlink, symlinkat - make symbolic link to file

Syntax #include <unistd.h>

int symlink(const char *path1,, const char *path2);
int symlinkat(const char *path1, int fd, 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.

The `symlinkat()` function is equivalent to the `symlink()` function except when the *path2* parameter specifies a relative path. In this case the symbolic link is not created in the current directory, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

When the value `AT_FDCWD` is transferred to the `symlinkat()` function for the *fd* parameter, the current directory is used.

Return val. 0 if successful.

-1 if an error occurs; `errno` is set to indicate the error.

Errors `symlink()` and `symlinkat()` 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` 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.

The new symbolic link cannot be created because there is no space left on the file system which will contain the link.

There are no free inodes on the file system on which the file is to be created.

`EROFS` The new symbolic link would reside on a read-only file system.

Extension

`EDQUOT` 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.

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.

The user's quota of inodes on the file system on which the file is to be created was exceeded.

`EFAULT` *path1* or *path2* points outside the allocated address space for the process.

`ENOSYS` The file system does not support symbolic links. ❏

In addition, `symlinkat()` fails if the following applies:

`EACCES` The file descriptor *fd* was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path2* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

`ENOTDIR` The *path2* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

Notes `symlink()` and `symlinkat()` are executed for POSIX files only

See also `lchown()`, `link()`, `lstat()`, `open()`, `readlink()`, `fcntl.h`, `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 val. 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 *name* (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

System variable	Value of *name* (constant)
_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
_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

System variable	Value of *name* (constant)
PTHREAD_KEYS_MAX	_SC_THREAD_KEYS_MAX
PTHREAD_STACK_MIN	_SC_THREAD_STACK_MIN
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 getgrgid_r() and getgrnam_r()	_SC_GETGR_R_SIZE_MAX
Maximal size of the data buffer of the functions getpwnam_r() and getpwuid_r()	_SC_GETPW_R_SIZE_MAX

Return val. 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 *fsname*, a null-terminated file-system identifier, into a file-system type index.

`GETFSTYP` Translates *fs_index*, a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by *buf.* This buffer must be at least of size `FSTYPSZ` (see `sys/fstyp.h`).

`GETNFSTYP` Returns the total number of file system types configured in the system.

Return val.
File system type index
if *opcode* is `GETFSIND`; upon successful completion.

0 if *opcode* is `GETFSTYP`; upon successful completion.

Number of file system types configured
if *opcode* is `GETNFSTYP`; upon successful completion.

-1 if unsuccessful. `errno` is set to indicate the error.

Errors
`sysfs()` will fail if:

`EINVAL` *fsname* points to an invalid file-system identifier; *fs_index* is zero or invalid, or *opcode* is invalid, or
an attempt was made to access a BS2000 file.

`EFAULT` *buf* or *fsname* 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 49).

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 val. 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)`.

$\neq 0$ 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 ≠ 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 "Executive Macros" [10]). 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" [2].

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 val.

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 val.

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 val.

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 129.

tcflow - suspend or restart data transmission

Syntax #include <termios.h>

int tcflow(int *fildes*, int *action*);

Description `tcflow()` suspends transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*. The *fildes* 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 *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.

Extension
All values are supported for connections with a remote processor. ❑

Return val.

0	if successful.
-1	if an error occurs. `errno` is set to indicate the error.

Errors `tcflow()` will fail if:

`EBADF`	*fildes* 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 *fildes* 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 129.

tcflush - discard non-transmitted data

Syntax

#include <termios.h>

int tcflush(int *fildes*, int *queue_selector*);

Description

fildes is a file descriptor associated with a terminal. Upon successful completion, `tcflush()` discards data that was written to the object referred to by *fildes* 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 `tcflush()` 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.

Extension
All values are supported for connections with a remote processor. ❑

Return val.

0	if successful.
-1	if an error occurs. `errno` is set to indicate the error.

Errors

`tcflush()` will fail if:

`EBADF`	*fildes* 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 *fildes* is not a terminal.

Notes

`tcflush()` has no effect on block-mode terminals.

See also

`tcdrain()`, `termios.h`, `unistd.h`, section “General terminal interface” on page 129.

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 val.

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 129.

tcgetpgrp - get foreground process group ID

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

pid_t tcgetpgrp(int *fildes*);

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 val. 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` *fildes* 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 val. 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 *fildes*, int *duration*);

Description

Extension
In non-conformance with XPG4, this function has no effect and returns without performing any action. ❑

Return val.

0	if successful.
-1	if an error occurs. `errno` is set to indicate the error.

Errors
`tcsendbreak()` will fail if:

`EBADF` *fildes* 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 *fildes* is not a terminal.

See also
`termios.h`, `unistd.h`, section “General terminal interface” on page 129.

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 val. 0 if successful.

-1 if an error occurs. `errno` is set to indicate the error.

Errors `tcsetattr()` will fail if:

`EBADF` *fildes* 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 *fildes* 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 129.

tcsetpgrp - set foreground process group ID

Syntax
#include <unistd.h>

Optional
#include <sys/types.h> ❑

int tcsetpgrp(int *fildes*, 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 *fildes* 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 val.

0	if successful.
-1	if an error occurs. `errno` is set to indicate the error.

Errors
`tcsetpgrp()` will fail if:

`EBADF`	*fildes* 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 *fildes*);

Description `tell()` returns the current value of the file position indicator for the file associated with file descriptor *fildes*. `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.

fildes is the file descriptor of the file for which the current value of the file-position indicator is to be determined.

Return val. 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(`*fildes*`)` and `lseek(`*fildes*, `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 directoy stream, then `telldir()` returns the position specified in the *loc* argument of the `seekdir()` call.

Return val.

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 val.

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, time64 - get time since the Epoch

Syntax

#include <sys/types.h>
#include <time.h>

time_t time(time_t **tloc*);
time64_t time64(time64_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.

As of 19.1.2038 03:14:08 hrs UTC time outputs the message CCM0014 and terminates the program.

The `time64()` function behaves like time with the difference that it also returns correct results after 19.1.2038 03:14:07 hrs.

BS2000
`time()` returns the current time (local time) as the number of seconds that have elapsed since January 1, 1970, 00:00:00 local time. ❑

`(time_t)`-1
`(time64_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 val.

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 val.
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 val.

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 `tempnam()` 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()`, `tempnam()`, `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 val.
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
`isacii()`, `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 val. 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 val. 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 val. 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 val. 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 val. 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"))            /* towlower(wc) */
towctrans(wc, wctrans("toupper"))            /* towupper(wc) */
```

Return val.
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(), towlower(), towupper(), wctrans()`

towlower - convert wide characters to lowercase

Syntax
#include <wchar.h>

wint_t towlower(wint_t *wc*);

Description
`towlower()` converts the wide character *wc* to the corresponding lowercase letter if *wc* is an uppercase wide-character code.

Return val.
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
`towupper()`, `setlocale()`, `wchar.h`.

towupper - convert wide characters to uppercase

Syntax #include <wchar.h>

wint_t towupper(wint_t *wc*);

Description `towupper()` converts the wide character *wc* to the corresponding uppercase letter if *wc* is a lowercase wide-character code.

Return val. 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 `towlower()`, `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

#include <search.h>

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));

Description

`tsearch()`, `tfind()`, `tdelete()` and `twalk()` manipulate binary search trees. Comparisons are made with a user-supplied *compar* 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.

`tsearch()` is used to build and access the tree. The *key* argument is a pointer to an element to be accessed or stored. If there is an entry in the tree that is equal to **key* (the value pointed to by the key), a pointer to this found entry is returned. Otherwise, **key* is inserted, and a pointer to it is returned. Only pointers are copied, so the calling routine must store the data. The *rootp* argument points to a variable that points to the root of the tree. A null pointer value for the variable pointed to by *rootp* 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.

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()`.

`tdelete()` deletes a node from a binary search tree. The arguments are the same as for `tsearch()`. The variable to which *rootp* 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.

`twalk()` traverses a binary search tree. The *root* 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. *action* 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.

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 val. **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 val. 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 *fildes*, char * *name*, size_t *namesize*);

Description The function `ttyname_r()` stores the null-terminated pathname of the terminal associated with file descriptor *fildes* 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 val. 0 if successful.

Otherwise the error number.

Errors `ttyname_r()` fails if:

`EBADF`	*fildes* is not a valid file descriptor.
`ENOTTY`	*fildes* 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 val.

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 ≠ 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 val.

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 val. 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 val. 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 val.

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 val. 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 val.

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 val. 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, unlinkat - remove link

Syntax #include <unistd.h>

int unlink(const char *path);
int unlinkat(int *fd*, const char **path*,int *flag*);

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()`). ❑

The `unlinkat()` function is equivalent to the `unlink()` or `rmdir()` function except when the *path* parameter specifies a relative path. In this case the directory entry to be deleted is not searched for in the current directly, but in the directory connected with the file descriptor *fd*. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

In the *flag* parameter, the value `AT_REMOVEDIR`, which is defined in the `fnctl.h` header, can be transferred. In this case *fd* and *path* should be used to specify a directory and not a normal file.

When the value `AT_FDCWD` is transferred to the `unlinkat()` function for the *fd* parameter, the current directory is used.

Return val. 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()` and `unlinkat()` 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 *path* is not a directory.

`EPERM` The file named by *path* 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.

In addition, `unlinkat()` fails if the following applies:

`EACCES` The *fd* parameter was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

`ENOTDIR` The *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory,
or
the *flag* parameter has the value `AT_REMOVEDIR`, and *path* does not specify a directory.

`EEXIST` or `ENOTEMPTY`
The *flag* parameter has the value `AT_REMOVEDIR` and *path* specifies an unreadable directory, or hard links to the directory which differ from dot exist or more than one entry exists in dot-dot.

`EINVAL` The value of the *flag* parameter is invalid.

Notes `rmdir()` is used to delete a directory.

The program environment determines whether `unlink()` or `unlinkat()` is executed for a BS2000 or POSIX file.

See also `close()`, `link()`, `remove()`, `rmdir()`, `fcntl.h`, `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 val.

0	if successful.
-1	otherwise. `errno` is set to indicate the error.

Errors

`unlockpt()` will fail if:

`EBADF`	The *fildes* argument is not a file descriptor open for writing.
`EINVAL`	The *fildes* argument is not assigned to a master pseudoterminal.

See also

`grantpt()`, `open()`, `ptsname()`, `stdlib.h`.

unsetenv - remove an environment variable

Syntax

#include <stdlib.h>

int unsetenv (const char **name*);

Description

The `unsetenv()` function removes an environment variable from the environment of the calling process.

The *name* argument points to a string, which is the name of the variable to be removed. This string shall not contain an '=' character. If the named variable does not exist in the current environment, the environment remains unchanged and the function is considered to have completed successfully.

If the application modifies *environ* or the pointers to which it points, the behavior of `unsetenv` is undefined. The `unsetenv` function updates the list of pointers to which environ points.

`unsetenv()` is not thread-save.

Return val.

0	if successful.
-1	otherwise. `errno` is set to indicate the error. The environment remains unchanged.

Errors

`unsetenv()` will fail if:

`EINVAL`	The *name* argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

See also

`environ`, `exec`, `getenv()`, `malloc()`, `putenv()`, `setenv()`, `stdlib.h`, section “Environment variables” on page 104.

usleep - suspend process for defined interval

Syntax
#include <unistd.h>

int usleep(useconds_t *useconds*);

Description
Suspends 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 val.

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 val.

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 val. 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 *times* is not zero.
`EROFS`	The file system containing the file is mounted as a read-only file system.

See also `sys/time.h`.

utimensat - Setting file access and update times

Syntax

#include <sys/stat.h>

int utimensat(int *fd*, const char **path*, const struct timespec *times*[2], int *flag*);

Description

The `utimensat()` function sets the access and update times of a file to the values specified in *times*. The times of the file are changed to which the *path* parameter points relative to the directory connected with the file descriptor *fd*. The function permits time specifications which are accurate to the nanosecond.

The *times* parameter is an array consisting of two structures of the type *timespec*. The access time is set to the value of the first element, and the update time to the value of the second element. The times in the *timespec* structure are specified in seconds and nanoseconds since the epoch.

If the *tv_nsec* field of a *timespec* structure has the special value `UTIME_NOW`, the corresponding timestamp of the file is set to the current time. If the *tv_nsec* field of a *timespec* structure has the special value `UTIME_OMIT`, the corresponding timestamp of the file should not be updated. In both cases the content of the *tv_sec* field is ignored.

When *times* is the null pointer, the access and update times are set to the current time. If the file descriptor was opened without `O_SEARCH`, the function checks whether a search is permitted in the connected file descriptor with the authorizations applicable for the directory. If the file descriptor was opened with `O_SEARCH`, the check is not performed.

A process may call `utimensat()` with the null pointer for *times* set or with both *tv_nesc* fields set to `UTIME_NOW` only if it has one of the following properties:

- owner of the file,
- write authorization for the file, or
- special rights.

A process may call `utimensat()` with a pointer other than `NULL` for *times* in which both *tv_nesc* fields are not set to `UTIME_NOW` or `UTIME_OMIT` only if it is the owner of the file or a process with special rights.

When both *tv_nesc* fields are set to `UTIME_OMIT`, the access authorization is not checked. However, other errors can occur.

When the value `AT_FDCWD` is transferred to the `utimensat()` function for the *fd* parameter, the current directory is used.

In the *flag* parameter, the value `AT_SYMLINK_NOFOLLOW`, which is defined in the `fnctl.h` header, can be transferred. If *path* specifies a symbolic link, the timestamps of the symbolic link are updated.

Return val. 0 in the case of success,

-1 in the case of an error `errno` is set to display the error.

Errors `utimensat()` fails when the following applies:

`EACCES` A component of the path may not be searched, or *times* is a null pointer and the effective user number is not that of the system administrator and not that of the owner of the file, and write access is rejected
or
the *fd* parameter was not opened with `O_SEARCH`, and the authorizations applicable for the directory do not permit the directory to be searched.

`EBADF` The *path* parameter does not specify an absolute pathname, and the *fd* parameter does not have the value `AT_FDCWD`, nor does it contain a valid file descriptor opened for reading or searching.

Extension
`EFAULT` *times* is not equal to zero and points beyond the process's assigned address space, or *path* points beyond the process's assigned address space.

`EINTR` A signal was intercepted during the system call `utimensat()`.

`EINVAL` An attempt was made to access a BS2000 file or the value of the *flag* parameter is invalid.

`ELOOP` During the compilation of *path* too many symbolic links occurred to ❑.

`ENAMETOOLONG`
The length of *path* exceeds `{PATH_MAX}` or the length of a component of *path* exceeds `{NAME_MAX}`.

`ENOENT` The specified file does not exist.

`ENOTDIR` A component of the path is not a directory, or the *path* parameter does not specify an absolute pathname, and the file descriptor *fd* is not connected with a directory.

`EPERM` The effective user number if not that of the system administrator and not that of the owner of the file, and *times* is not equal to zero.

`EROFS` The file system containing the file has been mounted write-protected.

See also `fcntl.h`, `sys/stat.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 val.
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 val.

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 val.

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 val.

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. `errno` 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 val.

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).

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records which are longer than the maximum record length are truncated to the maximum record length when they are written. By default or with the specification `split=yes`, these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

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 val.

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.

BS2000

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the specification `split=no` was entered for `fopen()`, records which are longer than the maximum record length are truncated to the maximum record length when they are written. By default or with the specification `split=yes`, these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

See also

`vfprintf()`, `vsprintf()`

vsnprintf - formatted output to a string

Syntax #include <stdarg.h>
#include <stdio.h>

int vsnprintf(char **s*, size_t *n*, const char **format*, va_list *arg*);

`vsnprintf()` formats data (characters, strings, numerical values)according to the specification in the *format* string and writes the data to the area to which *s* points.

`vsnprintf()` is similar to the `vsprintf()` function. In contrast to `vsprintf()`, `vsnprintf()` only outputs up to the buffer limit specified by the *n* parameter. This prevents buffer overrun. n must not exceed `INT_MAX` in size.

`vsnprintf()` outputs a maximum of *n*-1 characters and adds a NULL character (\0) at the end of the output. If *n*=0, nothing is output.

`vsnprintf()` exists, analogous to `vsprintf()` , as an ASCII, IEEE and ASCII/IEEE function (cf. sections “IEEE floating-point arithmetic” on page 37 und “ASCII encoding” on page 42).

Parameters See `fprintf()`.

Return val.

< 0	*n* > `INT_MAX` or output error.
= 0 .. n-1	It was possible to edit the output completely. The return value specifies the length of the output without the terminating `NULL` character
> n	It was not possible to edit the output completely. The return value specifies the length of the output without the terminating `NULL` character which a complete output would require.

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 117).

va_list *arg*
Pointer to the variable argument list that was initialized with `va_start`.

Return val.

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 `FSTOPPED(` **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 val. Process ID of the child process

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 val.

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:

`WEXITED`	waits for processes to exit.
`WTRAPPED`	waits for traced processes to be interrupted or reach a breakpoint (see `ptrace()`).
`WSTOPPED`	waits and returns the process status of a child process which stopped after a signal was received.
`WCONTINUED`	returns the status of a child process that was suspended and then resumed.
`WNOHANG`	returns immediately if there are no child processes to be waited for.
`WNOWAIT`	keeps the process whose status was returned in *infop* 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 val.	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`	*infop* points to an invalid address.

See also `exec`, `exit()`, `wait()`, `sys/wait.h`

wcrtomb - convert wide characters to multi-byte characters

Syntax

<wchar.h> is included via:

```
#include <wchar.h>
```

size_t wcrtomb(char **s*, wchar_t *wc*, mbstate_t **ps*);

Description

If *s* is a null pointer, `wcrtomb()` corresponds to the call
`wcrtomb(buf, L'\0', ps)`
where *buf* designates an internal buffer.

If *s* is not a null pointer, `wcrtomb()` 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 preceded a Shift sequence that restores the initial conversion state.

The final state corresponds to the “initial conversion” state.

Return val.

`(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()`

wcscat - concatenate two wide character strings

Syntax

#include <wchar.h>

wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);

Description

`wcscat()` 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*.
`wcscat()` terminates the wide character string with a null byte (`\0`).

Return val.

Pointer to the resulting wide character string *ws1*.

Notes

Wide character strings terminated with the null wide character (`\0`) are expected as arguments.
`wcscat()` 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`.

wcschr - scan wide character string for wide characters

Syntax

#include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wint_t wc);

Description

`wcschr()` 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 `wchar_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 val.

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 `wchar_t` (see `stddef.h`). ❑

See also

`wcsrchr()`, `wchar.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 val.
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`.

wcscoll - compare two wide character strings according to LC_COLLATE

Syntax
#include <wchar.h>

int wcscoll(const wchar_t **ws1*, const wchar_t **ws2*);

Description
`wcscoll()` lexically compares two wide character strings *ws1* and *ws2*, in accordance with the collation sequence defined for the locale in `LC_COLLATE`.

Return val.
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
`wcscoll()` 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 `wcscoll()` and after the call check `errno`. If `errno` is not 0, assume that an error has occurred.

For sorting long lists, the `wcsxfmr()` and `wcscmp()` 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
`wcsncmp()`, `wcsxfrm()`, `wchar.h`.

wcscpy - copy wide character string

Syntax

#include <wchar.h>

wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);

Description

`wcscpy()` 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 val.

Pointer to the resulting wide character string *ws1*.

Notes

Wide character strings terminated with the null wide character code (`\0`) are expected as arguments.
`wcscpy()` 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 val.

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 *wss* 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 val.

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 val.

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 val.

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

`wcscat()`, `wchar.h`.

wcsncmp - compare two wide character substrings

Syntax

<wchar.h> is included as: #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 val.

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

`wcscmp()`, `wchar.h`.

wcsncpy - copy wide character substring

Syntax

<wchar.h> is included as: #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 val.

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

`wcscpy()`, `wchar.h`.

wcspbrk - get first occurrence of wide character in wide character string

Syntax #include <wchar.h>

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);

Description `wcspbrk()` 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 val.

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 val.

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 `wcrtomb()` 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 val. `(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()`

wcsspn - get length of wide character substring

Syntax

#include <wchar.h>

size_t wcsspn(const wchar_t *_ws1_, const wchar_t *_ws2_);

Description

Starting at the beginning of the wide character string _ws1_, `wcsspn()` 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 val.

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

`wcscspn()`, `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 val. 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 val. 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`	he 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 `iswspace()`, `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 val.

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 val. 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 val.

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

`iswalpha()`, `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 val.

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 val. 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 `iswalpha()`, `scanf()`, `wcstod()`, `wcstol()`, `wchar.h`.

wcstoull - convert wide character string to unsigned long long

Syntax #include <wchar.h>

unsigned long long int wcstoull(const wchar_t *restrict *nptr*, wchar_t **restrict *endptr*, int *base*);

Description `wcstoull()` converts the initial portion of the wide character string pointed to by *nptr* to `unsigned 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.

`wcstoull()` then attempts to convert the subject sequence to an integer of type `unsigned long long int`, 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 val. 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()`

wcswcs - find wide character substring in wide character string

Syntax
#include <wchar.h>

wchar_t *wcswcs(const wchar_t *$ws1$, const wchar_t *$ws2$);

Description
`wcswcs()` locates the first occurrence of the wide character string *ws2* (excluding the terminating null wide character code) in the wide character string *ws1*.

Return val.

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 val.
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 `wcscmp()` function returns the same return value (greater than, equal to or less than zero) for two transformed wide character strings as the `wcscoll()` 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, *wc1* can be a null pointer.

If copying is between overlapping objects, the result is undefined.

Return val.

Integer value < n
indicating the number of wide character codes written to the field (without terminating null).

Integer value ≥ n
in this case the content of the *ws1* field is undefined.

(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 `wcscmp()` 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 `wcscoll()` 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

`wcscmp()`, `wcscoll()`, `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 val.

`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`.
`wcstomb()` 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 val.

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 val.

Value ≠ `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 val.

0	if the character class name is not valid for the current locale (category `LC_CTYPE`).
$\neq 0$	Αν οβφεχτ οφ τψπε `wctype_t` τηατ χαν βε υσεδ ιν χαλλσ το `iswctype()` ισ ρετυρνεδ.

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 val.

-1	if *wc* does not correspond to a representable wide character code.
0	if *wc* is a null wide-character code.
1	if *wc* 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 val.

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 val. < 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 val. 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 `memcmp()`, `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 val. 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 val. 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 *fildes* 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 val. 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.

– *fildes* 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 `write()` fails if the following applies:

`EAGAIN` The `O_NONBLOCK` flag is set for the file descriptor and the process would be delayed in the `write()` operation.

`EBADF` *fildes* 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 `getrlimit()` and `ulimit()`).

Extension
`EAGAIN` The amount of system memory available for raw I/O is temporarily insufficient, or
an attempt was made to write to a stream that cannot accept data with the `O_NDELAY` or `O_NONBLOCK` flag set, or
an attempt was made to write `{PIPE_BUF}`or fewer bytes to a pipe or FIFO and less than *nbytes* of free space was available. ❑

Extension
`EDEADLK` The `write()` function is sleeping and causes a deadlock situation to occur.

`EFAULT` *buf* points outside the allocated address space of the process. ❑

`EINTR` The write operation was terminated by a signal, and no data was transferred.

Extension
`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 `SIGTTIN` signal or the process group of the process is orphaned.

`ENOSPC` There was no free space remaining on the device containing the file.

Extension
`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.
The process gets a `SIGPIPE` signal.

`ERANGE` An attempt was made to write to a stream with an *mbyte* value outside the prescribed minimum and maximum limits, and the minimum value is non-zero.

`EINVAL`	The stream or multiplexer referred to by *fildes* 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 110).

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 117). ❑

The following applies in the case of text files with SAM access mode and variable record length for which a maximum record length is also specified: When the O_NOSPLIT specification was entered for `open`, records which are longer than the maximum record length are truncated to the maximum record length when they are written with `write`. By default (i.e. without the specification O_NOSPLIT), these records are split into multiple records. If a record has precisely the maximum record length, a record of the length zero is written after it. ❑

See also

`creat()`, `dup()`, `fcntl()`, `lseek()`, `open()`, `pipe()`, `ulimit()`, `unistd.h`.

writev - write to file

Syntax

#include <sys/uio.h>

ssize_t writev(int *fildes*, 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 < *iovcnt* ≤ `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 *fildes* 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 val.

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` *fildes* 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

`chmode()`, `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 val.

Value of the Bessel function of *x*, if *x* >0.

-`HUGE_VAL`	for arguments ≤ 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`.

5 Appendix: KR or ANSI functionality

All details presented in this section apply to the functions marked with xx in the table on page 51ff (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 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 local time.

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
%*n*	Job number *n*
%*string*	Job whose command begins with *string*
%?*string*	Job whose command contains *string*

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 pubset 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 UNIX 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()`, `sigfprocmask()` 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 129).

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" [2].

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 129).

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`, `SYSTERM`)

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 `SYSTERM` (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. 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

You will find the manuals on the internet at *http://manuals.ts.fujitsu.com*. You can order printed versions of manuals which are displayed with the order number.

[1] **POSIX** (BS2000)
POSIX Basics for Users and System Administrators
User Guide

[2] **POSIX** (BS2000)
Commands
User Guide

[3] **C** (BS2000)
C Compiler
User Guide

[4] **C/C++** (BS2000)
C/C++ Compiler
User Guide

[5] **C/C++** (BS2000/OSD)
POSIX Commands of the C/C++ Compiler
User Guide

[6] **CRTE**
C Library Functions
Reference manual

[7] **CRTE**
Common RunTime Environment
User Guide

[8] **DCE** (BS2000)
POSIX Program Interface
User Guide

[9] **SDF-P** (BS2000)
Programming in the Command Language
User Guide

[10] **BS2000 OSD/BC**
Executive Macros
User Guide

[11] **BS2000 OSD/BC**
Introductory Guide to DMS
User Guide

[12] **JV** (BS2000)
Job Variables
User Guide

[13] **XHCS** (BS2000)
8-Bit Code and Unicode Processing in BS2000/OSD
User Guide

Other publications

X/Open CAE Specification
System Interfaces and Headers, Issue 4

ISBN: 1-872630-47-2
X/Open Document Number: C202

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

International Standard ISO/IEC 9899 : 1990,
Programming languages - C

International Standard ISO/IEC 9899 : 1990,
Programming languages - C / Amendment 1

Index