English

BS2000/OSD

C Library Functions V3.1A

C Library Functions

Reference Manual

Edition June 2012

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

Certified documentation according to DIN EN ISO 9001:2008

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 9001:2008.

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

Copyright and Trademarks

Contents

1 Preface

The C runtime system is part of the Common Runtime Environment CRTE. When CRTE is used in BS2000/OSD operating systems without POSIX, the C runtime system offers more than 300 predefined functions, including all the functions defined in the ANSI/ISO standard (as well as ISO-C Amendment 1 to ISO/IEC 9899:1990), and around 50 BS2000/OSD-specific extensions. These functions serve as a convenient aid in programming many tasks for which no higher-level language facilities are provided in C itself. Some examples of such programming tasks include:

– processing of files (open, close, position, read, write, etc.)

– processing of individual characters or strings (search, change, copy, delete etc.)

– processing of multibyte and wide characters

– processing of type `long long integer`

– dynamic memory management (allocation and deallocation of storage areas, etc.)

– access to operating system functions (system commands, utility routines, etc.)

– mathematical functions (trigonometric, logarithmic etc.)

The functions are available either as source program sections (macros) or in the form of precompiled program segments (modules). In this manual the term “function” is used to include both types, unless it is necessary to make a distinction between the two.

The required function declarations, definitions of constants, data types and macros, as well as the function macros themselves are incorporated in “include files” (often referred to in C literature as “standard include files”, “include headers” etc.). These include files are source program sections which can be addressed in the C program via `#include` directives and are temporarily copied to the program during each compilation.

All the functions and header files are stored in CRTE libraries as library elements.

A detailed description of how to access the CRTE libraries during the compilation, linkage and execution of a C or C++ program is provided in the C and C++ User Guides.

1.1 Summary of contents and target group

This manual describes all the C functions and macros of the C runtime system that can be used in the BS2000/OSD operating system without POSIX. It addresses users of C who employ CRTE V2.9A in BS2000/OSD operating systems in which no POSIX subsystem is available, and also developers of C applications that are designed to run without POSIX.

Familiarity with the C programming language and the BS2000 operating system is a prerequisite to using this manual effectively.

Chapter 2 contains notes and points to be generally observed when using the library functions, e.g. differences between functions and macros, insertion of include files, error handling etc.

Chapter 3 provides an overview of the library functions from the viewpoint of their content.

Chapters 4 to 6 contain basic information, programming notes and examples concerning file processing, STXIT/Contingency routines and locales.

Chapter 7 is a reference section that contains the descriptions of the individual library functions, arranged in alphabetical order.

The Appendix contains tables that show which of the library functions described here are defined in the ANSI standard.
In addition, it also includes a section on selecting the KR functionality supported by C/C++ compiler versions prior to V3.0.

In the body of the text, references to other publications are made using abbreviated titles; the full titles are listed in the “Related publications” section at the back of the manual. Notes on how to order manuals are given at the end of the same section.

The POSIX library functions of the C runtime system (approx. 300 functions that are defined in the XPG4 standard plus some UNIX/SINIX-specific extensions) are described in the manual “C Library Functions for POSIX Applications”.

1.2 Changes since the last version of the manual

The differences between this manual and the manual describing the C library functions in V2.6A are associated with the following major innovations in V3.1A of the C runtime system in CRTE V2.9A:

- Introduction of the TIMESHIFT bind option
  This shifts the reference day (epoch) for time functions from 1/1/1950 00:00:00 to 1/1/1970 00:00:00. As a result, the time functions will return correct results up to 1/19/2038 03:14:07 (instead of 1/19/2018 03:14:07) without any intervention being required in the source program.
- New time functions with 64-bit time counter
  The `ctime64`, `difftime64`, `ftime64`, `gmtime64`, `localtime64`, `mktime64` and `time64` functions use a 64-bit time counter and will therefore return correct results up to 3/18/4317 02:44:48.
- New functions `snprintf` and `vsnprintf`
  These functions behave like `sprintf` and `vsprintf`, but abort the output when the buffer limit is reached.
- New function `_cstxit`
  `_cstxit` corresponds to the `cstxit` function.
  However, the `stxitpar` structure itself is not transferred, but a pointer to this. This enables the `_cstxit` function to set the `err_set` structure element in such a way that it can be evaluated by the calling program.
- The `bs2cmd` function can be used with a variable parameter list. This allows command output to be sent to a memory buffer area specified by the user instead of to stdout/stderr.

1.3 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 online

Readme files are available to you online in addition to the product manuals under the various products at *http://manuals.ts.fujitsu.com*.

Readme files under BS2000/OSD

On your BS2000 system you will find Readme files for the installed products under the file name:

```
SYSRME.CRTE.029.E
```

Please refer to your system administrator for the user ID under which the required Readme file can be found. You can also obtain the path name of the Readme file directly by entering the following command:

```
/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>,LOGICAL-ID=SYSRME.E
```

You can view the Readme file on screen with `/SHOW-FILE` or by opening it in an editor, or print it on a standard printer using the following command:

```
/PRINT-DOCUMENT <filename>, LINE-SPACING=*BY-EBCDIC-CONTROL
```

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 at *http://manuals.ts.fujitsu.com*.

2 Use of the library functions

This chapter contains general information on what should be taken into account when using the library functions.

2.1 Functions and macros

Most of the library functions are implemented as C functions, a few as macros. Some library functions are implemented both as functions and as macros (see list below).

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 in parentheses () or canceled with the `#undef` directive.
Which version you select depends on whether and which aspects (performance, program size, restrictions) are relevant to the particular program.

A **function** is a compiled program section (module) that is available only once and is treated as an external subroutine at execution time. An organizational overhead is required for each function call during program execution; for example, management of the local, dynamic data of a function in the runtime stack, saving of register contents, return addresses, etc.

Some library functions can be generated inline, controlled via the OPTIMIZATION compiler option. In this case the function code is included directly in the call and the above-mentioned administrative activities are not needed.
At present, the following functions can be generated inline: `strcpy`, `strcmp`, `strlen`, `strcat`, `memcpy`, `memcmp`, `memset`, `abs`, `fabs`, `labs`.

The C, C++ and C/C++ User Guides provide more detailed information on this topic.

A **macro** is a source program segment defined by the `#define` directive. During compilation, the macro name in the program is replaced by the contents of the called macro each time the macro is called.

The use of a macro may result in better performance during program execution, since the runtime system is not required to perform administrative functions (see function). However, the compiled program increases in size due to macro expansions.

When using a macro, the following should also be taken into account:

– The names of macros cannot be passed to other functions as arguments if they require a pointer to a function as an argument.

– Using increment/decrement or compound assignment operators for macro arguments may produce unwanted side effects.

– The include file which contains the macro definition must always be incorporated in the program.

List of library functions implemented as macros or as both macros and functions:

Macro:	**Function:**	**Macro:**	**Function:**
clearerr	(clearerr)	iswdigit	(iswdigit)
clock	(clock)	iswgraph	(iswgraph)
feof	(feof)	iswlower	(iswlower)
ferror	(ferror)	iswprint	(iswprint)
getc	(getc)	iswpunct	(iswpunct)
getchar	(getchar)	iswspace	(iswspace)
getwc	(getwc)	iswupper	(iswupper)
isalnum	(isalnum)	iswxdigit	(iswxdigit)
isalpha	(isalpha)	putc	(putc)
iscntrl	(iscntrl)	putchar	(putchar)
isdigit	(isdigit)	toebcdic	(toebcdic)
isebcdic	(isebcdic)	tolower	(tolower)
isgraph	(isgraph)	toupper	(toupper)
islower	(islower)	towlower	(towlower)
ispunct	(ispunct)	towupper	(towupper)
isspace	(isspace)	assert	
isupper	(isupper)	offsetof	
isxdigit	(isxdigit)	va_arg	
iswalnum	(iswalnum)	va_end	
iswalpha	(iswalpha)	va_start	
iswcntrl	(iswcntrl)		

2.2 Include files

Every library function is declared in an include file. Many library functions use symbolic constants and data types, which are defined in include files. The library functions that are implemented as macros are also located in include files.

The include files required for the use of library functions are components of the CRTE library SYSLIB.CRTE. They are stored there as source program elements (type S) and are copied to the program during compilation on the basis of the preprocessor `#include` directive. The C and C++ User Guides provide a detailed description of how this has to be done.

The following include files are available:

<ascii_ebcdic.h>	<assert.h>	<bs2cmd.h>	<cglobals.h>	<cont.h>
<crduc.h>	<ctype.h>	<errno.h>	<float.h>	<ieee_390.h>
<ilcs.h>	<inttypes.h>	<iso646.h>	<limits.h>	<locale.h>
<math.h>	<setjmp.h>	<signal.h>	<stdarg.h>	<stddef.h>
<stdio.h>	<stdlib.h>	<string.h>	<strings.h>	<stxit.h>
<sys.timeb.h>	<sys.types.h>	<time.h>	<timeb.h>	<wchar.h>
<wctype.h>				

The following items, among others, may be contained in include files:

– Definitions of symbolic constants with the values required for proper execution of the functions, e.g.:

BUFSIZ	standard size of the I/O buffer (8192 bytes), as defined by the operating system
EOF	end-of-file criterion (-1)
WEOF	end-of-file criterion for wide character files (L“-1“)
_NFILE	maximum permissible number of concurrently opened files, including the standard files `stdin`, `stdout` and `stderr` (2048).
NULL	null pointer (0)

- Definitions of data types and structures that are used by the functions, e.g.:

FILE	Most I/O functions use a pointer to a structure of type FILE (see also section “FILE structure” on page 62”)
mbstate_t	This data type is used by many of the multibyte functions and corresponds to the type `char` in this implementation.
size_t	This data type is used by many of the string functions and corresponds to the type `unsigned` in this implementation.
ptrdiff_t	This data type is used for the result of a subtraction of pointers and corresponds to the type `int` in this implementation.
wint_t	This data type includes values corresponding to elements of the extended character set and the value WEOF (end of input) and corresponds to the `type integer` in this implementation.
wchar_t	This data type is used by the multibyte functions and corresponds to the type `long` in this implementation.
wctrans_t	A scalar data type representing locale-specific character translations.
wctype_t	A scalar data type representing locale-specific character classes; corresponds to the type `long` in this implementation.
clock_t	This data type is used by the clock function and corresponds to the type `int` in this implementation.
time_t	This data type is used by many of the time functions and corresponds to the type `long` in this implementation.
va_list	This data type is used by functions that process variable argument lists (e.g. `vprintf`).

- The prototype declaration of all functions

  Before a function is called, the data type must be made known, i.e. declared. This is ensured by incorporating the appropriate include file. In the “ANSI” and “STRICT-ANSI” compilation modes, a warning is issued if the declaration is missing. In the “CPLUSPLUS” compilation mode, an error occurs if the declaration is missing, and no module is created.

  See also section “Preprocessor define _STRICT_STDC” on page 41.

- The definition of all macros

  Some library functions are implemented as macros. In order to be able to use a macro, the appropriate include file must be incorporated into the program.

The include files contain *extern “C”* declarations for all functions and data so that C library functions can be called from within C++ sources.

Include file iso646.h

The include file `iso646.h` contains the following 11 macros, which expand to the adjacent notations and thus serve as alternative notations for the corresponding operators:

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

2.3 Error handling

In order to program effectively, it is advantageous in the case of most function calls to check whether the function performed successfully. This may be done as follows:

```
if(fct(...) == error result)    /* Query error return value */
  {
    perror("fct:");             /* Output error information */
    exit(error code);           /* Response to the error, e.g. */
  }                             /* program termination in this case */
else...
```

Most functions return an error return value when an error occurs.
Furthermore, in many cases, the internal C variable `errno` (type `integer`) is set to an appropriate error code. Information specifying the error in more detail is then edited internally (in a structure) on the basis of this error code. The information output by the `perror` function contains:

– a brief error text explaining the error,
– the name of the function at which the error occurred,
– the DMS error code (hexadecimal), if any, for incorrect file access.

All error codes as well as the associated error information are defined in the include file <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>: ERANGE, for example, indicates an overflow error (value 2).
A query could appear as shown in the following example, which uses the `signal` function:

```
#include <errno.h>
        .
        .
if(signal(sig, fct) == 1) /* Query error result */
{
  if(errno == EFAULT)
        .          /* Responses to EFAULT (invalid address) */
        .
  else if(errno == EINVAL)
        .          /* Responses to EINVAL (invalid argument) */
        .
}
else...
```

In addition to the `errno` variable there are two other variables defined in the include file <errno.h>:

The name of the errored function can be accessed with `_ _errcmd`, and the hexadecimal DMS code with `_ _errhex`. Both variables are of `char[8]` type.

Notes

- The variables `errno`, `_ _errcmd` and `_ _errhex` must not be explicitly defined by the user. The include file <errno.h> must be incorporated in the program in order for these variables to be queried.
- The contents of the area in which the errno error code is internally stored are preserved until they are overwritten with current information when an error occurs again. `perror` calls and queries of the `errno`, `_ _errcmd` and `_ _errhex` variables are therefore only useful immediately after a function has provided an error return value.

In the examples given for the individual function descriptions, error queries have often been omitted so as to keep the examples from swelling unnecessarily.

2.4 Pointer as a return value and result parameter

Return value pointer

<type> *funct(...)

Some functions that return a pointer write their result to an internal C data area that is overwritten each time this function is called. Because this is a common source of errors, it is explicitly mentioned 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. When calling from within C++ sources, explicit type conversion is mandatory.

Example

```
long *long_ptr;

.

.

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

Result parameter pointer

<type1> funct(<type2> *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 `const`.

The address, i.e. a pointer, must always be passed as the argument. In addition, you must explicitly provide memory space for the result 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);
```

2.5 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 26).

– 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 27).

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 28).

– You can use conversion functions to convert floating-point numbers explicitly from /390 format to IEEE format (see page 29).

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.

2.5.1 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 28), 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 27) 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
```

2.5.2 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 29).

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

There is an IEEE variant for each of the following C library functions:

acos	asin	atan	atan2	atof	ceil
cos	cosh	difftime	difftime64	ecvt	erf
erfc	exp	fabs	fcvt	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					

2.5.3 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 26).

_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 26).

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 (see page 42).

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 26):

_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 27) 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
```

2.5.4 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 alphabetical reference section (see page 113).

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 31).

– 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 35).

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 35).

– You can use conversion functions to convert ASCII characters and strings explicitly from EBCDIC format to ASCII format (see page 37).

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 35), 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 33) 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*

With CRTE V2.4A and V2.4B, strings were always transferred to the main function in EBCDIC format in the case of parameter transfer (*argv*), environment variables (*environ* structure) and the global variable *tzname* (also when LITERAL-ENCODING= *ASCII-FULL). In these two CRTE versions, to permit ASCII format to be used for end-to-end working additional interventions were therefore required in the source code to convert the specified strings.

In CRTE V2.4C and higher 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. In CRTE V2.4C and higher you can thus produce applications which have been ported to BS2000/OSD or were originally generated as EBCDIC applications as ASCII applications without any need for intervention in the source code.

Compatibility with CRTE V2.4A and V2.4B

By default CRTE Version V2.4C and higher is incompatible with CRTE Versions V2.4A and V2.4B with respect to the ASCII encoding. To permit ASCII applications which were created for use with one of the CRTE versions mentioned to continue to operate, you can, however, specify the old behavior with the ENVIRONMENT-ENCODING option in the MODIFY-RUNTIME-PROPERTIES statement.

The ENVIRONMENT-ENCODING option is evaluated only for the main function; in other cases it is ignored. As a result you cannot define any other program environment than the main program for subroutines.

```
MODIFY-RUNTIME-PROPERTIES .... ENVIRONMENT-ENCODING=*STD|*EBCDIC
```

ENVIRONMENT-ENCODING=*STD
External strings (*argv*, *environ* and *tzname*) are encoded in the manner specified in the MODIFY-SOURCE-PROPERTIES LITERAL-ENCODING option (default). When LITERAL-ENCODING=*ASCII-FULL these transferred strings are therefore also available in ASCII encoding.

Caution!

The behavior activated by *STD is incompatible with the behavior of CRTE V2.4A and V2.4B when LITERAL-ENCODING=*ASCII-FULL.

ENVIRONMENT-ENCODING=*EBCDIC
External strings (*argv*, *environ* and *tzname*) are provided in EBCDIC format irrespective of the encoding of the main function.

Specifying *EBCDIC guarantees compatibility with the behavior of CRTE Versions V2.4A and V2.4B. This enables you to continue to use programs which you have created in accordance with this behavior without having to intervene in the programs' source code.

In POSIX you define this behavior using the following option:

`-K environment_encoding_ebdic`

Detailed information on the ENVIRONMENT-ENCODING option is provided in the "C/C++ Compiler" manual.

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 37).

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

There is an ASCII variant for each of the following C library functions:

asctime	assert	atof	atoi	atol
atoll	basename	bs2exit	bs2fstat	creat
creat64	ctime	ctime64	ecvt	fdopen
fgetc	fgets	fopen	fopen64	fprintf
fputc	fputs	fread	freopen	freopen64
fscanf	fwrite	gcvt	getc_unlocked	getenv
getpgmname	gets	gettsn	isalnum	isalpha
isascii	iscntrl	isdigit	isgraph	islower
isprint	ispunct	isspace	isupper	localeconv
mknod	mktemp	open	open64	perror
printf	remove	rename	scanf	setlocale
snprintf	sprintf	sscanf	strerror	strlower
strptime	strtod	strtol	strtoll	strtoul
strtoull	strupper	tmpnam	tolower	toupper
ungetc	vfprintf	vsnprintf	vsprintf	

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 31).

_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 31).

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 (see page 42).

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 31):

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 33) 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
```

2.6.4 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 alphabetical reference section (see page 113).

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 28) and `_ASCII_SOURCE` (see page 35) 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

There is an ASCII/IEEE variant for each of the following C library functions:

atof	ecvt	fcvt	fprintf	fscanf	gcvt
fprintf	fscanf	gcvt	printf	scanf	snprintf
sprintf	sscanf	srtod	vfprintf	vsnprintf	vsprintf

2.8 Multibyte and wide characters

Wide characters and multibyte characters were defined to expand on the original “character” concept of computer languages, which was based on assigning each character one byte of memory. This assignment proved insufficient for languages such as Japanese, for example, since the representation of a character in such languages requires more than one byte of storage. For this reason, the character concept has now been expanded to include multibyte characters and wide characters.

Multibyte characters represent characters of the extended character set in one, two, three or more bytes.
Multibyte strings may include “shift sequences“, which change the meaning of the following multibyte codes. Shift sequences can thus be typically used to switch between different interpretation modes. For example, the one-byte shift sequence `0200` may define that the following byte pairs are to be interpreted as Japanese characters, whereas the shift sequence `0201` may define that the following byte pairs are to be interpreted as characters of the ISO Latin 1 character set.

Programming model

Due to the new functions added in Amendment 1, programs that work with multibyte characters can now be implemented just as easily as programs which use the traditional character concept.

When multibyte characters or strings are read from an external file, they are internally converted to a `wchar_t` object or an array of type `wchar_t`. During the read operation, the multibyte characters are converted to the corresponding wide character codes.
These `wchar_t` objects can then be processed with `isw`*xxx* functions, `wcstod`, `wmemcmp`, etc., and the resulting `wchar_t` objects can subsequently be output with the wide character output functions such as `putwchar`, `fputws`, and so on.
During the write operation, the wide character codes are converted to the corresponding multibyte characters.

Notes on wide characters

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

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

Wide character constants are written in the form L“*widecharstring*“.

Notes on this implementation

This version of the C runtime system supports only 1-byte characters as wide character codes. These characters are of type `wchar_t`, which is internally mapped to the type `long`.
Consequently, multibyte characters always have a length of 1 byte in this implementation.

2.9 Time functions

For conversion to seconds, the time functions described in this manual by default use 1/1/1950 00:00:00 as the reference date (epoch).

As a result, the time functions `ctime`, `difftime`, `ftime`, `gmtime`, `localtime`, `mktime` and `time` will no longer function from the year 2018. For C applications in BS2000 the year 2018 is thus a greater problem than the millennium change.

The TIMESHIFT bind option shifts the reference date (epoch) to 1/1/1970 00:00:00. As a result, the specified time functions supply correct results up to 1/19/2038 03:14:07 without requiring any intervention in the source program, but in all other ways they behave in exactly the same way as when no use is made of the bind option.

The bind option is contained in the following libraries:

– SYSLNK.CRTE.TIMESHIFT,
– SKULNK.CRTE.TIMESHIFT and
– SPULNK.CRTE.TIMESHIFT

To use the bind option, enter the following statement when binding:

```
INCLUDE-MODULE LIB=<library>,ELEMENT=*ALL
```

Time functions with 64-bit time counter

The `ctime64`, `difftime64`, `ftime64`, `gmtime64`, `localtime64`, `mktime64` and `time64` functions use a 64-bit time counter. In contrast to the corresponding functions with a 32-bit counter (`ctime`, `difftime`, `ftime`, `gmtime`, `localtime`, `mktime` and `time`) they will therefore supply correct results up to 3/18/4317 02:44:48. In the case of the time functions with a 64-bit time counter, the reference date is always 1/1/1970 00:00:00 irrespective of whether the TIMESHIFT bind option is used.

2.10 Preprocessor define _STRICT_STDC

The standard includes in the library SYSLNK.CRTE contain the prototype declarations for all C library functions which the C runtime system provides. Approx. 50 of these library functions are not defined in the ANSI standard, but are BS2000-specific (e.g. `bs2fstat`, `_edt`) or UNIX-specific extensions (e.g. `open`, `gamma`).

The define `_STRICT_STDC` is provided to permit applications that conform to the ANSI standard to be programmed.
This define can be set with the following option at compilation time:

– For version V2.2 of the C and C++ compilers:

  `SOURCE-PROPERTIES = PAR(LANGUAGE-STANDARD = STRICT-ANSI)`

– For the C/C++ compiler as of V3.0:

  `MODIFY-SOURCE-PROPERTIES LANGUAGE=*C(MODE=*STRICT-ANSI)`

If the define `_STRICT_STDC` is set, the prototype declarations for all functions in the standard includes that are not defined in the ANSI standard are deactivated or bypassed. The names of these functions are then freely available as user-defined names.

The define `_STRICT_STDC` relates only to prototype declarations within ANSI-defined standard includes. The BS2000-specific include headers do not contain a query for this define.

All the functions provided by the C runtime system are listed in the appendix (page 517 ff). Information on whether a function is defined in the ANSI standard or is an extension is provided for each function.

2.11 Preprocessor defines for function prototypes according to XPG4

The following functions are defined differently in the XPG4 Standard and in Amendment 1:

`fputwc`, `putwc`, `putwchar`, `wcschr`, `wcsrchr`, `wcstok`

The preprocessor defines `_XOPEN_SOURCE_EXTENDED` and `_XOPEN_SOURCE` can be used to control whether the prototype of the function compliant with XPG4 or Amendment 1 is to be made available.

If you do not set `_XOPEN_SOURCE_EXTENDED` and `_XOPEN_SOURCE`, the prototype that complies with Amendment 1 is offered.

If you set `_XOPEN_SOURCE_EXTENDED` or `_XOPEN_SOURCE`, the prototype that complies with XPG4 is offered.

2.12 Preprocessor define _MAP_NAME

When IEEE floating-point arithmetic, ASCII encoding and 64-bit interfaces for large files (> 2 GB), some C library functions can be replaced with the corresponding IEEE, ASCII or 64-bit variants of these functions.

You can use the `_MAP_NAME` preprocessor define to specify whether they are to be replaced by means of the name define method or the macro define method:

- If you define `_MAP_NAME`, the name define method is used.
- If you do not define `_MAP_NAME`, the macro define method is used.

The name define method defines a macro without arguments by means of a `#define` statement, whereas the macro define method defines a macro with an argument list by means of a `#define` statement.

Which solution is to be preferred depends on the specific application program. Pointers to functions are not registered in the macro define method, for example, while variables are also renamed (incorrectly) in name defines.

3 Overview of the functions

This chapter contains overviews of the functions, grouped according to content. Each function appears exactly once.

3.1 File processing

The term "elementary", as used below, refers to all I/O functions that work on the basis of file descriptors, as opposed to the standard I/O functions, which work on the basis of file pointers. In addition, functions that can also be used on files with record-oriented input/output (record I/O) are indicated as such.

File access (open, close, position)

Name	Brief description
close	Close file and flush buffer (elementary)
creat	Create file or overwrite existing file (elementary)
creat64	64-bit variant of `creat` for processing large files (> 2 GB)
fclose	Close file and flush buffer (also record I/O)
fdelrec	Delete record in indexed-sequential file (record I/O only)
fdopen	Assign a file pointer to a file descriptor
fflush	Flush file buffer
fgetpos	Return current position of the read/write pointer (also record I/O)
fgetpos64	64-bit variant of `fgetpos` for processing large files (> 2 GB)
flocate	Explicitly position indexed-sequential file (record I/O only)
fopen	Open file (also record I/O)
fopen64	64-bit variant of `fopen` for processing large files (> 2 GB)
freopen	Reassign file pointer (also record I/O)
freopen64	64-bit variant of `freopen` for processing large files (> 2 GB)
fseek	Position read/write pointer (also record I/O)
fseek64	64-bit variant of `fseek` for processing large files (> 2 GB)
fsetpos	Position read/write pointer (also record I/O)
fsetpos64	64-bit variant of `fsetpos` for processing large files (> 2 GB)
ftell	Determine current position of the read/write pointer
ftell64	64-bit variant of `ftell` for processing large files (> 2 GB)
lseek	Position read/write pointer (elementary)
lseek64	64-bit variant of `lseek` for processing large files (> 2 GB)
fwide	Query/define orientation of a file
open	Open file (elementary)
open64	64-bit variant of `open` for processing large files (> 2 GB)
rewind	Position read/write pointer to beginning of file (also record I/O)
setbuf	Set up input/output buffer
setvbuf	Set up input/output buffer
tell	Query current position of the read/write pointer (elementary)

File management

Name	Brief description
bs2fstat	Access file name from catalog
mktemp	Generate unique temporary file name
remove	Delete file (also record I/O)
rename	Rename file (also record I/O)
tmpfile	Open temporary binary file
tmpfile64	64-bit variant of tmpfile for processing large files (>2 GB)
tmpnam	Generate temporary file name
unlink	Erase file (also record I/O)

File and error information

Name	Brief description
clearerr	Delete end-of-file and error flag (also record I/O)
feof	Test for end of file (also record I/O)
ferror	Test for file error (also record I/O)

Input/output

Name	Brief description
fgetc	Read a character from a file
fgets	Read a string from a file
fgetwc	Read a wide character from input file
fgetws	Read a wide character string from a file
fprintf	Formatted output to a file
fputc	Write a character to a file
fputs	Write a string to a file
fputwc	Write a wide character to a file
fputws	Write a wide character string to a file
fread	Read blockwise from a file (also record I/O)
fscanf	Formatted input from a file
fwprintf	Write a formatted character to an output file (wide character format)
fwrite	Write blockwise to a file (also record I/O)
fwscanf	Formatted input from a file (wide character format)
getc	Read a character from a file
getchar	Read a character from standard input
gets	Read a string from standard input
getw	Read wordwise from a file
getwc	Read a wide character from a file
getwchar	Read a wide character from standard input

Name	Brief description
printf	Formatted output to standard output
putc	Write a character to a file
putchar	Output a character to standard output
puts	Output a string to standard output
putw	Write wordwise to a file
putwc	Write a wide character to a file
putwchar	Write wide characters to standard output
read	Read from a file (elementary)
scanf	Formatted input from standard input
snprintf	Formatted output to a string
sprintf	Formatted output to a string
sscanf	Formatted input from a string
swprintf	Formatted output to a wide character string
swscanf	Formatted input from a wide character string
ungetc	Put a character back in the buffer
ungetwc	Put a wide character back in the buffer
vfprintf	Formatted output to a file
vfwprintf	Write a formatted character to a file (wide character format)
vprintf	Formatted output to standard output
vsnprintf	Formatted output to character string
vsprintf	Formatted output to character string
vswprintf	Write a formatted character to a wide character string
vwprintf	Formatted output to standard output (wide character format)
wprintf	Formatted output to standard output (wide character format)
write	Write to a file (elementary)
wscanf	Formatted input from standard input (wide character format)

3.2 Communication with the system environment

Name	Brief description
cputime	CPU time used for the current task
getenv	Query system information
getlogin	Return user ID
gettsn	Return TSN (task sequence number)
system	Execute system command
_edt	Call file editor EDT

3.3 Program information and execution control

Program information

Name	Brief description
getpgmname	Return name of the program
_ _DATE_ _	Output compilation date (macro)
_ _FILE_ _	Output source file name (macro)
_ _LINE_ _	Output current source program line number (macro)
_ _TIME_ _	Output compilation time (macro)
_ _STDC_ _	ANSI language standard (macro)
_ _STDC_ VERSION_ _	Check compliance with Amendment 1

Program termination

Name	Brief description
abort	Abnormal program termination
atexit	Register termination routines
bs2exit	Program termination (with MONJV)
exit and _exit	Program termination

Handling exception conditions, eventing

Name	Brief description
alarm	Set alarm clock
cdisco	Deactivate a contingency routine
cenaco	Definition of a contingency routine
cstxit and _cstxit	Definition of a STXIT routine
kill	Send signal to own program
raise	Send signal to own program
signal	Control signal processing
sleep	Suspend program for fixed period of time

Non-local jumps

Name	Brief description
setjmp	Set marker for non-local jumps
longjmp	Non-local jump

Program diagnostics

Name	Brief description
assert	Macro for error diagnosis

3.4 Memory management

Name	Brief description
calloc	Reserve memory space for an array
free	Release memory space
garbcoll	Release memory space to system (garbage collection)
malloc	Reserve memory space
memalloc	Reserve memory space (more than 2 Kbytes)
memfree	Release memory space (more than 2 Kbytes)
realloc	Reallocate memory space

3.5 Character processing

Character test

Name	Brief description
isalnum	Test for alphanumeric character
isalpha	Test for letter
isascii	Test for EBCDIC character
iscntrl	Test for control character
isdigit	Test for digit
isebcdic	Test for EBCDIC character
isgraph	Test for printable character except space
islower	Test for lowercase letter
isprint	Test for printable character including space
ispunct	Test for special character
isspace	Test for white-space character
isupper	Test for uppercase letter
isxdigit	Test for hexadecimal digit

Wide character test

Name	Brief description
iswalnum	Test for alphanumeric wide character
iswcntrl	Test for wide control character
iswctype	Test for wide character in character class *chartype*
iswdigit	Test for decimal-digit wide character
iswgraph	Test for visible wide character (excluding space)
islower	Test for lowercase wide character
iswprint	Test for printing wide character (including space)
iswpunct	Test for punctuation wide character
iswspace	Test for white-space wide character
iswupper	Test for uppercase wide character
iswxdigit	Test for hexadecimal wide-character digit

Character conversion

Name	Brief description
toascii	Conversion to EBCDIC
toebcdic	Conversion to EBCDIC
tolower	Conversion to lowercase
toupper	Conversion to uppercase

Wide character conversion

Name	Brief description
wcrtomb	Convert wide character to multibyte character
wctob	Convert wide character to (one-byte) multibyte character
towlower	Convert wide character to lowercase
towupper	Convert wide character to uppercase

3.6 Processing strings and character arrays (memory areas)

Strings

Name	Brief description
index	First occurrence of a character in a string
rindex	Last occurrence of a character in a string
strcat	Concatenate two strings
strchr	First occurrence of a character in a string
strcmp	Compare two strings
strcoll	Compare two strings
strcpy	Copy one string to another
strcspn	Calculate the length of a string segment that does not contain any character from a second string
strfill	Copy one string to another up to length *n* and fill with blanks if required
strlen	Calculate the current length of a string
strlower	Copy one string to another with conversion of uppercase to lowercase
strncat	Concatenate two strings up to length *n*
strncmp	Compare two strings up to length *n*
strncpy	Copy one string to another up to length *n*
strpbrk	First occurrence in a string of a character that matches a character in a second string
strrchr	Last occurrence of a character in a string
strspn	Calculate the length of a string segment containing only characters from a second string
strstr	First occurrence of a string in another string
strtok	Split string into several partial strings
strtok_r	Thread-safe variant of `strtok`
strupper	Copy one string to another with conversion of lowercase to uppercase
strxfrm	Transform a string

Character arrays (memory areas)

Name	Brief description
memchr	First occurrence of a character in a memory area
memcmp	Compare two memory areas
memcpy	Copy one memory area to another
memmove	Copy one memory area to another
memset	Initialize a memory area with a character

Wide character strings

Name	Brief description
towctrans	Map wide character
wcsrtombs	Convert wide character string to multibyte character
wcsstr	Find first occurrence of wide character string
wctrans	Define mapping between wide characters
wcscat	Concatenate two wide character strings
wcschr	Scan wide character string for wide characters
wcscmp	Compare two wide character strings
wcscoll	Compare two wide character strings according to LC_COLLATE
wcscpy	Copy wide character string
wcscspn	Get length of complementary wide character substring
wcsftime	Convert date and time to wide character string
wcslen	Get length of wide character string
wcsncat	Concatenate two wide character substrings
wcsncmp	Compare two wide character substrings
wcsncpy	Copy wide character substring
wcspbrk	Get first occurrence of wide character in wide character string
wcsrchr	Get last occurrence of wide character in wide character string
wcsspn	Get length of wide character substring
wcstod	Convert wide character string to floating-point number (double)
wcstok	Split wide character string into tokens
wcstol	Convert wide character string to long integer
wcstoul	Convert wide character string to unsigned long
wcsxfrm	Wide character sting transformation
wctype	Define wide character class

Wide characters arrays (memory areas)

Name	Brief description
wmemchr	First occurrence of wide character in wide character string
wmemcmp	Compare two wide character strings (memory areas)
wmemcpy	Copy wide character string (without overlapping of memory areas)
wmemmove	Copy wide character string to memory with overlapping areas
wmemset	Set n wide characters in wide character string

Multibyte functions

Name	Brief description
btowc	Convert (one-byte) multibyte character to wide character
mblen	Return number of bytes of a multibyte character
mbrlen	Determine remaining length of a multibyte character
mbsinit	Test for initial conversion state
mbrtowcs	Complete multibyte character and convert to wide character
mbstowcs	Convert a multibyte string to wide character string
mbtowc	Convert a multibyte character to a wide character
wcstombs	Convert wide character string to a multibyte string
wctomb	Convert a wide character to a multibyte character

3.7 Error messages

Name	Brief description
perror	Output standard error message
strerror	Return error message text

3.8 Time functions

Name	Brief description
asctime	Date and time
asctime_r	Thread-safe variant of asctime
clock	CPU time used since program call
ctime	Date and time (CET)
ctime64	Date and time (CET) (variant with 64 bit time counter)
difftime	Calculate time difference
difftime64	Calculate time difference (variant with 64 bit time counter)
ftime	Current time (GMT) as a structure
gmtime	Date and current local time (CET) as a structure
gmtime64	Date and current local time (CET) as a structure (variant with 64 bit time counter)
localtime	Date and current local time (CET) as a structure
localtime64	Date and current local time (CET) as a structure (variant with 64 bit time counter)
mktime	Convert date and time (calendar function)
mktime64	Convert date and time (calendar function) (variant with 64 bit time counter)
strftime	Locale-specific formatting of date and time
time	Current time (GMT) in seconds
time64	Current time (GMT) in seconds (variant with 64 bit time counter)

3.9 Mathematical functions

Integer arithmetic

Name	Brief description
abs	Absolute value (integer)
div	Division (integer)
labs	Absolute value (long integer)
llabs	Absolute value (long long integer)
ldiv	Division (long integer)
lldiv	Division (long long integer)

Floating-point numbers

Name	Brief description
acos	Arc cosine
asin	Arc sine
atan	Arc tangent x
atan2	Arc tangent x/y
cabs	Absolute value of a complex number
ceil	Round up to integer
cos	Cosine
cosh	Hyperbolic cosine
erf	Error function
erfc	Complement of the error function
exp	Exponential function
fabs	Absolute value of a floating-point number
floor	Round down to an integer
fmod	Remainder of a division
frexp	Normalized representation in base 2
gamma	Logarithmic gamma function
hypot	Euclidean distance
j0, j1, jn	Bessel functions of the first kind
ldexp	Calculate binary value
log	Natural log
log10	Base 10 log
modf	Split into integer part and fractional part
pow	General exponential function
sin	Sine
sinh	Hyperbolic sine
sqrt	Square root
tan	Tangent
tanh	Hyperbolic tangent
y0, y1, yn	Bessel functions of the second kind

Functions for rounding (independent of rounding mode)

Name	Brief description
llrint	Rounds type double to nearest integer of type long long int
llrintf	Rounds type float to nearest integer of type long long int
llrintl	Rounds type long double to nearest integer of long long int
lrint	Rounds type double to nearest integer of type long int
lrintf	Rounds type float to nearest integer of type long int
lrintl	Rounds type long double to nearest integer of long int
rint	Rounds type double to nearest integer of type double
rintf	Rounds type float to nearest integer of type float
rintl	Rounds type long double to nearest integer of type long double

Functions for rounding (dependent on rounding mode)

Name	Brief description
llround	Rounds type double to nearest integer of type long long int
llroundf	Rounds type float to nearest integer of type long long int
llroundl	Rounds type long double to nearest integer of long long int
lround	Rounds type double to nearest integer of type long int
lroundf	Rounds type float to nearest integer of type long int
lroundl	Rounds type long double to nearest integer of long int
round	Rounds type double to nearest integer of type double
roundf	Rounds type float to nearest integer of type float
roundl	Rounds type long double to nearest integer of type long double

3.10 Conversion of objects

Name	Brief description
atof	String to floating-point number
atoi	String to integer
atol	String to long integer
atoll	String to long long integer
ecvt	Floating-point value to string
fcvt	Floating-point value to string
gcvt	Floating-point value to string
mbstowcs	Multibyte string to long string
strtod	String to double (floating-point number)
strtol	String to long integer
strtoll	String to long long integer
strtoul	String to unsigned long integer
strtoull	String to unsigned long long integer
strptime	String to date and time (as structure)
wcsrtombs	Long string to multibyte string
wcsftime	Date and time to long string
wcstod	Long string to floating-point value (double)
wcstol	Long string to integer (long)
wcstoll	Long string to integer (long long)
wcstoul	Long string to integer (unsigned long)
wcstoull	Long string to integer (unsigned long long)

3.11 Other functions

Search and sort

Name	Brief description
bsearch qsort	Binary search algorithm Quick sort

Random number generator

Name	Brief description
rand rand_r srand	Random number generator Thread-safe variant of `rand` Initialize random number generator

Locales

Name	Brief description
localeconv setlocale	Query locale-specific data Select locale

Variable argument lists

Name	Brief description
va_arg va_end va_start	Process variable argument list (macro) Close variable argument list (macro) Initialize variable argument list (macro)

Offset of a structure component

Name	Brief description
offsetof	Offset of a structure component from the start of the structure in bytes (macro)

4 File processing

The following types of files can be processed with the input/output functions of the C runtime system:

– 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 the one hand and between stream and record-oriented input/output on the other (see also section “Basic terms” on page 60).

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

Up to 2048 files (including `stdin`, `stdout` and `stderr`) can be open at the same time.

4.1 Basic terms

This section explains file processing terms that are often used in the description of the C input/output functions (in chapter 7).

Binary file

A binary file is an ordered sequence of bytes. Data written with the aid of C output functions is transferred to the file on a 1:1 basis. In contrast to text files, control characters for line feed and tabs are not rendered effective (see section “Text file” on page 65) 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 are binary files with stream I/O:

- cataloged PAM files
- temporary PAM files (INCORE)
- cataloged SAM files opened with `fopen`/`fopen64` or `freopen`/`freopen64` in binary mode.

The following are binary files with record I/O:

- cataloged ISAM files
- cataloged SAM files
- cataloged PAM files

opened with the `fopen`/`fopen64` or `freopen`/`freopen64` functions in binary mode and with the suffix "type=record".

Binary mode may be specified only with the `fopen`/`fopen64` or `freopen`/`freopen64` functions.
If the elementary functions `open`/`open64` and `creat`/`creat64` are used, SAM and ISAM files are always opened as text files.

i When you work with the ASCII variants of the input/output functions and binary files, you have to take the following into account:

Since the data is written to a binary file by means of C input functions and read out again in the same format by means of C output functions, changes may have to be made in the case of programs that work with binary files. This is true, for example, when it comes to the processing of text components. In the case of an ISAM file, for example, if the key was stored as an EBCDIC string, you have to ensure that EBCDIC code is not compared with ASCII code in a string comparison.

File descriptor

A file descriptor is a positive integer that is used to identify a file when elementary access operations are performed on it. It is assigned to a file when the file is opened (with `open`/`open64`, `creat`/`creat64`). Once assigned, the file descriptor is used as the file argument for all further access operations (`read`, `write`, `close`, `tell`, etc.).

When a program is started, the standard I/O files are automatically opened with the following file descriptors:

0	Standard input
1	Standard output
2	Standard error output

File pointer

A file pointer is a pointer to a structure of type FILE. It is used when processing a file by means of standard access functions (see <stdio.h>). A file pointer is provided for a file when it is opened (with `fopen`/`fopen64`, `fdopen`, `freopen`/`freopen64`). This pointer serves as the file argument for all further access operations (`fprintf`, `fwprintf`, `fscanf`, `fclose`, etc.) on the file.

When a program is started, the standard I/O files are automatically opened with the following file pointers:

`stdin`	(standard input)
`stdout`	(standard output)
`stderr`	(standard error output)

Elementary

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.

A number of other implementations (e.g. UNIX, SINIX) provide elementary functions 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.

FILE structure

As soon as a file is opened with `fopen`/`fopen64`, `fdopen` or `freopen`/`freopen64`, 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
- position of the read/write pointer
- size of the file.

The file pointer returned by `fopen`/`fopen64`, `fdopen` or `freopen`/`freopen64` points to this FILE structure.

Read/write pointer

The read/write pointer contains information on the current file position. Data is respectively read or written from this current position onwards.
The structure of information in the read/write pointer varies in accordance with the type of file:

- For binary files with stream I/O it corresponds to the number of bytes, calculated from the beginning of the file.
- For text files it contains information on the current record and the position within this record. The structure differs for SAM and ISAM files. The information is used internally by the runtime system.
- For binary files with record I/O it corresponds to the position after the last record to be read, written or deleted, or to the position reached by an immediately preceding positioning operation.
  For ISAM files with duplicate keys, the read/write pointer is positioned after the last record of a group having identical keys if one of these records has previously been read, written or deleted.

Buffering

For all output functions which write data to text files and binary files with stream I/O (`printf`, `putc`, `fwrite` etc.) the data is stored in an internal C buffer and only written to the external file when a specific event occurs. This event is different for text and binary files.

Text file:

a) A newline character (\n) is detected.

b) The maximum record length for a disk file is reached.

c) For data display terminals: output to the terminal is followed by input from the terminal.

d) The positioning functions `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind` or `lseek`/`lseek64` are called.

e) The `fflush` function is called; `fflush` is automatically executed internally when a file is closed (`fclose`, `close`) or when a program is terminated normally or with `exit`.

f) The file is closed.

g) Also, for ANSI functionality: If reading from any text file makes a data transfer necessary from the external file to the internal C buffer, the data of all the ISAM files still stored in buffers is automatically written out to the files.

Even if the data in the buffer does not terminate with a newline character, writing to the external file causes a change of line. Subsequent data is written to a new line (or to a new record).

Exception for ANSI functionality:
If the data of an ISAM file in the buffer does not terminate in a newline character, writing to the external file does not produce a change of line (or change of record).

Subsequent data lengthens the record in the file. When an ISAM file is read, therefore, only newline characters explicitly written by the program are read in.

Binary file:

a) The buffer is full

b) The positioning functions `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind` or `lseek`/`lseek64` are called

c) The `fflush` function is called (see text file above)

d) The file is closed.

No buffering is performed for INCORE files and or for files with record I/O.

Record-oriented input/output

Record-oriented input/output means that the read/write pointer of the file can only be positioned at the start of a record (or block). Using record-oriented input/output makes efficient file processing adapted to the structure of the BS2000 system possible. The unit for an input/output function call is a record (or block). Additional functions are available which can be used, e.g., to delete or insert records or to access the key in an ISAM file.

Record-oriented processing can be used for cataloged SAM, ISAM and PAM files. The files must be opened with the functions `fopen`/`fopen64` or `freopen`/`freopen64`, qualified with "type=record" in the *type* parameter and always in binary mode.
Among other things, input/output functions which read in and output characters or character strings (up to \n) cannot be used on files with record I/O.

The following functions are used for processing files with record I/O:

`fopen`/`fopen64`, `freopen`/`freopen64`, `fclose`	Open, close
`fread`, `fwrite`	Read, write
`fsetpos`/`fsetpo64`, `fgetpos`/`fgetpos64`, `flocate`, `fseek`/`fseek64`, `rewind`	Position
`fdelrec`	Delete record

The following functions for file management and error handling can be used unchanged:

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

In contrast to stream I/O, no data is buffered in the case of record I/O (see section “Buffering” on page 63).

Stream-oriented input/output

Stream-oriented input/output means that the read/write pointer 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 I/O, the data for output to files with stream I/O is first stored in an internal C buffer and only later written to the external file (see section “Buffering” on page 63).

Text file

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 space are converted to their appropriate effect depending on the type of text file (see section “White space” on page 65). This means that data read from a text file does not correspond precisely to the data that was originally written to it. For a written tab (\t) an appropriate number of blanks is read.

The following points also apply to text files:

– Newline characters not originally written to the file may be read in (see `fflush`, `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `lseek`/`lseek64`, `rewind`).

– Output to SYSOUT and SYSTERM (for writing)
  Each line is started with a blank as a print control character. This causes 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 is specified in a line.

– The contents of a text file are always interpreted as a sequence of EBCDIC characters. When text files are processed using the ASCII variant of an I/O function (see page 33), the data is therefore converted internally as follows:

  – When data is written to the file, it is converted from ASCII to EBCDIC.
  – When data is read from the file, it is converted from EBCDIC to ASCII.

White space

The control characters for space and the backspace control character ’\b’ (cf. table below) are evaluated by all output functions which write to text files and receive as the argument the control character either as a character constant (starting with \) or as a numerical EBCDIC value.
The decimal or hexadecimal values of the control characters are given in the C and C++ User Guides (EBCDIC table).

Key to 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)

	\n	\t	\f	\v	\r	\b
SAM/ISAM	X	X				
SYSOUT/SYSTERM	X	X	X			
SYSLST	X	X	X	X	X	X

Tab (\t)
: The tab character is converted to the corresponding number of spaces. Tab positions are spaced 8 columns apart (1, 9, 17, ...). Spaces are substituted for the tab character when read in.

With SAM and ISAM files, the tab character is converted to spaces by default only when KR functionality has been selected, not for ANSI functionality (see additional specification option "tabexp" for the `fopen`/`fopen64`, `freopen`/`freopen64` functions).
KR functionality applies to C/C++ versions prior to V3.0 only.

Line feed (\n)
: The newline character is converted to a change of line (change of record). Subsequent read functions then supply 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 at the data display 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)
: There is no line feed and the cursor is returned to the start of the current line, 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 (cf. `isspace`) but a control character (cf. `iscntrl`).

\r and \b are effective only in conjunction with printers equipped with the overwrite function.

4.2 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 (see section “Preprocessor define _MAP_NAME” on page 42).

*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 (see page 42).

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).

4.3 System files (SYSDTA, SYSOUT, SYSLST)

SYSDTA

A C program can use SYSDTA as follows:

– An open function (`fopen`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`) 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 input is to be from a cataloged file instead of the data terminal, there are two ways of doing this:

1. If a parameter line was requested with PARAMETER-PROMPTING=YES (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 cataloged file. Please refer to your C and C++ User Guides.

   This reassignment does **not** have an effect on files that were opened with the names "(SYSDTA)" or "(SYSTERM)". Input from files with this name is still expected from the data terminal.

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

   For all input functions, input data is then expected from the assigned file.

   The following must be observed when an assignment is made with the ASSIGN-SYSDTA command:

   – After the program is executed, the internal record pointer is 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 has been declared in the C program, the EOF or WEOF condition for inputs at the data terminal can be provoked 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`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`) 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 input 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);
```

– In this case, the file pointer `stderr` or the file descriptor 2 is 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 (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 cataloged file. Please refer to your C and C++ User Guides.

This reassignment has **no** effect on files that were opened with the name "(SYSOUT)" or "(SYSTERM)".

SYSLST

A C program can use SYSLST as follows:

– An open function (`fopen`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`) is used to open a file with the name "(SYSLST) for writing. The file pointer returned by the open function 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 (in the RUNTIME-OPTIONS compiler option), this parameter line can be used to redirect the standard output or standard error output to SYSLST (please refer to your C and C++ User Guides).

  This reassignment has **no** effect on 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 not to be automatically output to a printer but sent to a cataloged file instead, SYSLST must be reassigned before the program is executed. This is effected with the command:
ASSIGN-SYSLST *filename*.

4.4 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 system are assigned according to the type of C file (binary file, text file, stream-oriented or record-oriented input/output). 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 for the open functions (`open`/`open64`, `creat`/`creat64`, `fopen`/`fopen64`, `freopen`/`freopen64`) and this link name must be linked with the name of the cataloged file in the ADD-FILE-LINK command.

Not all possible file attributes can be combined. Combinations which are not necessary either for functional or performance reasons are not supported by the input/output functions of the C runtime system.

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.

4.4.1 Default values and permissible modifications of the file attributes

The input/output functions of the C runtime system can process disk files with the file attributes listed in the following tables. The default attributes which the runtime system inserts if the user does not specify any options in the ADD-FILE-LINK command or in the open functions are underlined.

Notes on Tables 1 to 3

– The maximum number of data bytes in the following 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 data volume (see also page 78).
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).

– Please also refer to section “K and NK block formats” on page 78 for information on the block format (BLKCTRL) and the maximum number of data bytes.
In particular, you will learn how to avoid overflow blocks with NK-ISAM files which occur if the full length of a transfer unit is utilized when writing the 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, RECSIZE (RECORD-SIZE parameter in the ADD-FILE-LINK command) determines the register in which the length of a record is passed. This register is predefined (R4) and must not be changed.

Table 1: File attributes of text files for stream-oriented input/output

FCB-TYPE	**REC-FORM**	**BLKCTRL**	**BLKSIZE (STD,n)**	**RECSIZE (r byte)**	**Max. number of data bytes**
SAM [1]	V	PAMKEY	1≤ n ≤16	4≤ r ≤n*2048-4	RECSIZE - 4
		DATA(2K)	1≤ n ≤16	4≤ r ≤n*2048-16	RECSIZE - 4
		DATA(4K)	2≤ n ≤16		
	U	PAMKEY	1≤ n ≤16		BLKSIZE
		DATA(2K)	1≤ n ≤16		BLKSIZE - 16
		DATA(4K)	2≤ n ≤16		
ISAM [2]	V	PAMKEY	1≤ n ≤16	12≤ r ≤n*2048	RECSIZE - 12
		DATA(2K)	1≤ n ≤16	12≤ r ≤n*2048	RECSIZE - 12
		DATA(4K)	2≤ n ≤16		

1) In KR mode SAM files are created by default (KR mode applies to C/C++ versions prior to V3.0 only).
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 system: when a new ISAM file is created the first record is assigned the key "00010000" and the key is incremented in steps of 100 for each further record.

Table 2: File attributes of binary files for stream-oriented input/output

FCB-TYPE	**REC-FORM**	**BLKCTRL**	**BLKSIZE (STD,n)**	**RECSIZE (r byte)**	**Max. number of data bytes**
SAM	F	PAMKEY	1≤ n ≤16	1≤ r ≤n*2048	RECSIZE
		DATA(2K)	1≤ n ≤16	1≤ r ≤n*2048-16	RECSIZE
		DATA(4K)	2≤ n ≤16		
	V	PAMKEY	1≤ n ≤16	4≤ r ≤n*2048-4	RECSIZE - 4
		DATA(2K)	1≤ n ≤16	4≤ r ≤n*2048-16	RECSIZE - 4
		DATA(4K)	2≤ n ≤16		
	U	PAMKEY	1≤ n ≤16		BLKSIZE
		DATA(2K)	1≤ n ≤16		BLKSIZE - 16
		DATA(4K)	2≤ n ≤16		
PAM		PAMKEY	1≤ n ≤16		BLKSIZE
		DATA(2K)	1≤ n ≤16		BLKSIZE - 12
		DATA(4K)	2≤ n ≤16		
		NO(2K)	1≤ n ≤16		BLKSIZE
		NO(4K)	2≤ n ≤16		

Table 3: File attributes of binary files for record-oriented input/output

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

1) The default attributes for key position (for record format V = 5, F = 1) and key length (8) can be modified up to 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 input/output, the ISAM keys belong to the record data which is written from the C program or read into the C program.

4.4.2 K and NK block formats

BS2000 supports data volumes with different formats:

– **K**ey data volumes for storing files with the block control information in a separate field ("PAMKEY") per 2Kbyte data block. These files have the block format PAMKEY.

– **N**on-**K**ey data 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).

With BS2000 V11.0, NK volumes are distinguished by the minimum size of the transfer unit. NK2 volumes have the old transfer unit (2Kbyte). NK4 volumes have a transfer unit of 4Kbyte.

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

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

With BS2000 V11.0, there are two more operand values for NK-ISAM files:

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

Please refer to the "DMS Introductory Guide and Command Interface" 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.

If the ADD-FILE-LINK command is not used when a new file is created or BLOCK-CONTROL-INFO=BY-PROGRAM is specified, the default values of the C runtime system are used. These values depend on the disk type, on the CLASS2-OPTION that can be specified by the system administrator, and on the access method:

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

K and NK-ISAM files

ISAM files in K format which make 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 since the DMS forms extensions of 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 consequently the number of input/output operations during file processing

– under no circumstances may the ISAM key be in an overflow block.

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 can be used for NK-ISAM files.

Usable area for records (NK-ISAM files)

For ISAM files the following table can be used to calculate the space available for records per logical block.

File format	RECORD-FORMAT	max. 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 lower 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 lower 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 2-byte long record pointer for each record.
For RECORD-FORMAT=FIXED there is a 4-byte record length field for each record but this is not included in 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 to the next lower number
divisible by 4: 4044 (bytes).
```

4.4.3 Support of the DIV access method

As of BS2000/OSD V1.0, DMS offers the new access method DIV (DATA IN VIRTUAL). This access method is particularly suitable for processing the unstructured streams which often occur in C programs (e.g. those ported from UNIX).

DIV enables NK-PAM files to be processed which contain no data management information (BLOCK-CONTROL-INFO=NO) and are located on public volume.

If data which has already been read into a “window” as the result of a previous access is accessed frequently, the performance can be considerably enhanced.

Further background information on DIV is provided in the “DMS Assembler Interface” manual.

In BS2000 versions > V10, the C runtime system always uses the DIV access method for stream-oriented input/output to NK-PAM files without data management information. DIV cannot be used for NK-PAM files opened for record-oriented input/output.

Notes on stream-oriented input/output

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 this file is opened again and data is written at the end of the file a new record is started. New data is therefore written after the binary zeros.

If variable record length is used (V or U), new data can be written on a byte-specific basis. Variable record lengths do, however, tend certain loss of performance with positioning operations (e.g. `fseek`/`fseek64`, `ftell`/`ftell64`).

Binary files (PAM)

In order to permit byte-specific updating of PAM files (after closing and reopening), 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. For this reason, concurrent processing of a PAM file by different tasks is not possible if the file is extended by one of the participating tasks.
The C runtime system does not set any locks. If data is modified by several users, inconsistent states might result.

Text files (SAM, ISAM)

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

Notes on record-oriented input/output

Record-oriented input/output, which is possible for SAM, ISAM and PAM files, is always binary input/output. In the case of record-oriented input/output using the ASCII variants of the input/output functions (see page 33), data is therefore not converted either when it is written or when it is read.

With the `fopen`/`fopen64` or `freopen`/`freopen64` functions, the file must always be opened in binary mode and with the parameter option "type=record".

Input/output functions which read or write characters or strings (up to \n) cannot be used for record-oriented input/output.

Available input/output functions

The following functions are available for processing files with stream input/output:

`fopen`/`fopen64`, `freopen`/`freopen64`	Open
`fclose`	Close
`fread`	Read
`fwrite`	Write
`fsetpos`/`fsetpos64`	Positioning in the data stream
`fgetpos`/`fgetpos64`	Position in the data stream
`fseek`/`fseek64`	Position at start/end of file
`rewind`	Position at start of file
`flocate`	Explicitly position 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`

Any input/output functions not listed here are not available for record-oriented input/output and are rejected with an error return value.
For performance reasons, however, no checks are carried out for the two macros `getc` and `putc`. The behavior is undefined if these macros are used on files with record-oriented input/output.

Processing a file for record-oriented and stream-oriented input/output

Files which have been created for record-oriented input/output can be opened for stream-oriented input/output and vice versa. However, stream-oriented input/output does not support all the file attributes which are possible for record-oriented input/output.

FCB type of a new file to be created

The FCB type (FCBTYPE) of a new file to be created can be defined as follows:

– Specification in a ADD-FILE-LINK command and use of the LINK name in the `fopen`/`fopen64` and `freopen`/`freopen64` function

– Specification of the `forg` parameter in the `fopen`/`fopen64` and `freopen`/`freopen64` functions:

  "forg=seq": a SAM file is created
  "forg=key": an ISAM file is created.

The following restrictions apply to the FCBTYP of a file and the entries for `fopen`/`fopen64` and `freopen`/`freopen64`:

– For "type=record" the file must have FCBTYP SAM, PAM or ISAM.

– For "forg=seq" the file must have FCBTYP SAM or PAM.

– For "forg=key" the file must have FCBTYP ISAM.

– Specifying the append mode "a" is not allowed for ISAM files. The position is determined by the key in the record.

Multiple keys for ISAM files

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

Example of record-oriented processing of an ISAM file

The following program creates and processes an ISAM file using record-oriented input/output.

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

main()
{
  FILE * isamfp;
  size_t ret=0;
  int    i,intret;
  char   buffer[200];
  char   buffer2[200];
  static char * texts[3] = {"1  Ritchie***, 9999, ZZ",
                            "2  Kernighan*, 8765",
                            "3  Stroustrup, 1234, C++"};

  static char isamlink[] = "ADD-FILE-LINK LINK=ISAMFILE,F-NAME=FILE.ISAM,"
                           "ACCESS-METHOD=ISAM(KEY-LEN=10,KEY-POS=4),"
                           "REC-FORM=FIXED(REC-SIZE=50)";

  static int maxtext = 3;
  fpos_t isampos;

  ret = system(isamlink);
  if (ret != 0)
  {
     printf("system(isamlink) error\n");
     exit(1);
  }

  isamfp = fopen("link=isamfile", "wb+,type=record,forg=key");
  if (isamfp == 0)
  {
     printf("Attempt to open isamfp has failed\n");
     exit(2);
  }

  /* Write 3 records to ISAM file */

  for (i=0; i<maxtext; i++)
  {
     ret = fwrite(texts[i], 1, strlen(texts[i]), isamfp);
     if (ret == 0)
     {
```

```
         printf("Error on writing to ISAM file\n");
         exit(3);
      }
   }

   /* Read records from beginning of file and write on standard output */

   rewind(isamfp);
   for (i=0; i<maxtext; i++)
   {
      ret = fread(buffer, 1, 100, isamfp);
      fwrite(buffer, 1, ret, stdout);
   }

   /* Position explicitly on basis of key value and start processing */

   flocate(isamfp, "Ritch", strlen("Ritch"), _KEY_GE);

   ret = fread(buffer, 1, 100, isamfp);    /*"Ritchie" has been read        */
   *(buffer+ret) = '\0';                   /* EOS at end of record          */
   printf("\nRecord read: %s\n", buffer);
   fgetpos(isamfp, &isampos);              /* Record position after         */
                                           /* record just read              */

   ret = fread(buffer, 1, 100, isamfp);    /*"Stroustrup" has been read     */
   *(buffer+ret) = '\0';                   /* EOS at end of record          */
   printf("Record read: %s\n", buffer);

   fsetpos(isamfp, &isampos);              /* Reset file position indicator */
   ret = fread(buffer2, 1, 100, isamfp);   /*"Stroustrup" is read again     */
   *(buffer2+ret) = '\0';                  /* EOS at end of record          */
   printf("Record read: %s\n", buffer2);

   intret = fdelrec(isamfp, "Kernighan*"); /* Delete a record               */
   if (intret == 0)
      printf("Kernighan deleted\n");

   intret = fdelrec(isamfp, "Kernighan*"); /* Attempt to delete a record    */
   if (intret > 0)                         /* that has already been deleted */
      printf("OK, this record no longer exists\n");
   else
      printf("Error, \"Kernighan*\" could be deleted twice\n");

   printf("******* END OF PROGRAM *******\n");
}
```

4.5 Temporary PAM files in virtual memory (INCORE files)

If the file name "(INCORE)" is specified with the functions `fopen`/`fopen64`, `freopen`/`freopen64` or `open`/`open64`, a temporary PAM file is created in virtual memory. This file "lives" only for the duration of a program run.

INCORE files must be opened for writing before they can be accessed for reading (cf. `fopen`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`).

INCORE files are processed as binary files.

4.6 Standard input/output files stdin, stdout and stderr

In contrast to other implementations, the `stdin`, `stdout` and `stderr` macros are not constant expressions and therefore cannot be used in static initializations.

Example

The following construction is not permitted:

```
FILE *fp = stdin;
int main(void)
{
   .
   .
}
```

5 Contingency and STXIT routines

This chapter provides information on how contingency or STXIT routines can be implemented in C.

Familiarity with the concept of contingency and STXIT routines is important to the understanding of this chapter. These concepts as well as the corresponding BS2000 system macros are described in detail in the "Executive Macros" manual.

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

Caution

Using some of the 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) which has been 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:

- memory management routines: `malloc`, `calloc`, `realloc`, `free`
- file access functions for opening and closing files:
- `fopen/fopen64`, `freopen/freopen64`, `open/open64`, `creat/creat64`, `fclose`, `close`
- all file access, file management and input/output functions used on the same file
- random number generator functions: `rand`, `srand`
- time functions: `localtime/localtime_r`, `gmtime/gmtime_r`
- functions for enabling and disabling contingency routines: `cenaco`, `cdisco`
- `atexit`
- `strtok`
- `setlocale`

The "critical" functions also include the input/output functions in the C++ standard library.

5.1 C library functions (alarm, raise, signal)

The concept of contingency routines or STXIT contingency routines is primarily handled 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

STXIT contingency routines

The following STXIT event classes can be processed by means of the `alarm`, `raise` and `signal` functions:

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

The SVC interrupt event class is not supported at present.

Event-driven routines

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

Eventing via C library functions only operates within a task, i.e. intercommunication between different tasks is not possible.
These event-driven routines are therefore not implemented internally as contingency routines but via a CALL interface.

5.2 Free use of contingency routines

For special requirements that are not covered by the `signal` and `raise` functions (see section “C library functions (alarm, raise, signal)” on page 88), 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 opening 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` or `cdisco` must be invoked. In addition to enabling and disabling contingency routines, `cenaco` and `cdisco` 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 include 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 _abnorm1    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 */
```

Structure of the contingency routine:

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 built something like this:

```
#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; the program is continued at the point of interruption or

– by calling the `lonjmp` function; the program is resumed at the position defined with 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.

5.3 Free use of STXIT contingency routines

For special requirements that are not covered by the `signal` function (see section “C library functions (alarm, raise, signal)” on page 88), STXIT contingency routines can be freely programmed in C. Such requirements may include, for example, 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` or `_cstxit`.

The SVC interrupt event class cannot be implemented even if using the `cstxit` interface.

When the STXIT contingency routine is started, it is supplied with a structure which is declared in the include 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; the program is continued at the point of interruption or

– by calling the `lonjmp` function with a `jmp_buf` variable supplied by a `setjmp` call; the program is resumed at the position defined with a `setjmp` call or

– by calling the `longjmp` function with the termination label passed in the `stxcontp` structure (see above).

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

6 Locale

6.1 The locale concept

The principle underlying the concept of “locale” is to enable the behavior of C programs to be modified to take account of national conventions, standards and languages.

The locale directly affects the execution of certain C library functions. The `localeconv` function makes locale-specific information available in a structure which can be used for formatted output (`printf`, `fprintf` etc.).

The locale comprises the following categories:

LC_COLLATE	The sort sequence of the character set affects the behavior of the `strcoll`, `strxfrm`, `wcscoll` and `wcsxfrm` functions.
LC_CTYPE	Classification of the characters affects the behavior of the character handling macros `is...` (not `isdigit`, `iswdigit`, `isxdigit` or `iswdigit`), `tolower`, `toupper`, `towctrans`, `towlower`, `towupper`, `strlower`, `strupper` and `wctrans`.
LC_MONETARY	The conventions for representing monetary values (e.g. currency) affect the values supplied by `localeconv`.
LC_NUMERIC	The conventions for representing non-monetary numerical values (e.g. decimal point, sign) affect the type of decimal point for formatted input/output and for the conversion of strings (`atof`, `strtod`, `wcstod`) and also the values supplied by `localeconv`.
LC_TIME	The conventions for representing the date and time affect the behavior of `strftime` and `wcsftime`.

The C runtime system provides some predefined locales (see section “Predefined locale C” on page 94). Users can also define their own locales (see section “Compatible locales V1CTYPE and V2CTYPE” on page 97).

CRTE provides the predefined locales De.EDF04F and De.EDF04F@euro to support the euro. The only difference between these two locales lies in the category LC_MONETARY, which represents the German Mark (DM) for the locale De.EDF04F, and the euro for the locale De.EDF04F@euro.

The locale under which the C program is to run is selected with the `setlocale` function. Detailed descriptions of the C library functions mentioned in this section can be found in the reference section of this manual.
The locale C is preset by default, provided the `main` routine is not a C V1.0 object; in this case, the locale “V1CTYPE” or LC_C_V1CTYPE is set automatically when the program starts.

6.2 Predefined locale C

The C runtime system provides a number of predefined locales. When the program starts, the “C” locale is set.

Default locale

This locale is designated as "" or LC_C_DEFAULT. In this version it corresponds to the C locale.

C locale

This locale is designated as “C” or LC_C_C. It is the default locale when the program starts (with one exception: if the `main` routine is a C V1.0 object, then “V1CTYPE” applies, see page 97).

The C locale has the following effects in the various categories:

LC_CTYPE
: The classification corresponds to the EBCDIC definition of the individual characters (EBCDIC.DF.03, international version).

LC_NUMERIC
: The information defined in `localeconv` has the following values:

decimal_point	’.’
thousands_sep	""
grouping	""

LC_MONETARY

The information defined in `localeconv` has the following values:

int_curr_symbol	""
currency_symbol	""
mon_decimal_point	""
mon_thousands_sep	""
mon_grouping	""
positive_sign	""
negative_sign	""
int_frac_digits	CHAR_MAX (= 255)
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

English is used for the days of the week and the months of the year. The formats for date and time comply with the standard conventions for English-speaking countries.

LC_COLLATE

The sort sequence for the characters complies with the definition in the XPG4 standard, in which the sequence depends on the ASCII value of the characters (see table on next page). In all other predefined categories the sort sequence is determined by the EBCDIC value of each character as shown by the table on page 99).

Sort sequence in accordance with the XPG4 standard (ASCII)

\0	/	D	Y	n
\t	0	E	Z	o
\n	1	F	[	p
\v	2	G	\	q
\f	3	H	]	r
\r	4	I	^	s
␣	5	J	_	t
!	6	K	`	u
"	7	L	a	v
#	8	M	b	w
$	9	N	c	x
%	:	O	d	y
&	;	P	e	z
'	<	Q	f	{
(	=	R	g	|
)	>	S	h	}
*	?	T	i	~
+	@	U	j	
,	A	V	k	
-	B	W	l	
.	C	X	m	

6.3 Compatible locales V1CTYPE and V2CTYPE

V1CTYPE

This locale is designated as “V1CTYPE” or LC_C_V1CTYPE. It is set automatically when the program starts if the `main` routine is a C V1.0 object.

Differences from the C locale:

LC_CTYPE
: The characters X’8B’, X’8C’and X’8D’ are lowercase characters, X’AB’, X’AC’ and X’AD’ are uppercase characters, and X’C0’ and X’D0’ are special characters. In the “C” locale all these characters are control characters.

LC-COLLATE
: The sort sequence corresponds to the EBCDIC value of each character (see table on page 99).

V2CTYPE

This locale is designated as “V2CTYPE” or LC_C_V2CTYPE. It is compatible with locale “C” in versions 2.0 and 2.1 of the C runtime system.

Differences from the C locale:

LC-COLLATE
: The sort sequence corresponds to the EBCDIC value of each character (see table on page 99).

6.4 Country-specific locale GERMANY

A country-specific locale is available for the German-speaking area. This locale is designated as follows:

“GERMANY” LC_C_GERMANY

This locale differs from the C locale in the following ways:

LC_CTYPE

Characters X’FB’, X’4F’, X’FD’ and X’FF’ are classified as lowercase characters (ä, ö, ü, ß).
Characters X’BB’, X’BC’ and X’BD’ are classified as uppercase characters (Ä, Ö, Ü).

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

LC_MONETARY

International currency symbol (int_curr_symbol): “DEM”

Local currency symbol (currency_symbol): “DM”

Decimal point (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 complies with the standard conventions for German-speaking countries:

`<weekday name>, <day of month>.<name of month> <year>`

e.g. `Donnerstag, 25.Juli 1991`

LC_COLLATE

The sort sequence of the character set affects the behavior of the `strcoll`,`strxfrm`, `wcscoll` and `wcsxfrm` functions. For the “GERMANY” locale the sort sequence corresponds to the EBCDIC value of each characters shown in the following table.

Sort sequence in accordance with the EBCDIC value

\0	^	j	B	W
\t	,	k	C	X
\v	%	l	D	Y
\f	_	m	E	Z
\r	>	n	F	0
\n	?	o	G	1
␣	:	p	H	2
`	#	q	I	3
.	@	r	J	4
<	'	s	K	5
(	=	t	L	6
+	"	u	M	7
|	a	v	N	8
&	b	w	O	9
!	c	x	P	{
$	d	y	Q	}
*	e	z	R	~
)	f	[	S	
;	g	\	T	
-	h	]	U	
/	i	A	V	

6.5 The locales De.EDF04F and De.EDF04F@euro

Both of these locales support the processing of files and texts that contain the euro sign.

For compatibility reasons, the underlying conversion tables have been expanded in both locales to 8-bit code, which also contains the euro sign. The conversion tables are based on the ASCII code ISO 8859-15 or the EBCDIC code EDF04F.

The only difference between the two locales lies in the category LC_MONETARY.

LC_CTYPE
The following table indicates the base class to which each character belongs:

Symbolic name	Glyphe	class (n)	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

Symbolic name	Glyphe	class (n)	ASCII	EBCDIC
<CAN>		control	18	18
<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

Symbolic name	**Glyphe**	**class (n)**	**ASCII**	**EBCDIC**
<nine>	9	digit xdigit	39	F9
<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

Symbolic name	Glyphe	class (n)	ASCII	EBCDIC
<Z>	Z	upper	5A	E9
<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

Symbolic name	**Glyphe**	**class (n)**	**ASCII**	**EBCDIC**
<left-curly-bracket>	{	punct	7B	FB
<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

Symbolic name	Glyphe	class (n)	ASCII	EBCDIC
<sc1c>			9C	04
<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

Symbolic name	Glyphe	class (n)	ASCII	EBCDIC
<oe>	œ	lower	BD	B8
<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

Symbolic name	Glyphe	class (n)	ASCII	EBCDIC
<THORN>	Þ	upper	DE	8E
<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

Symbolic name	Glyphe	class (n)	ASCII	EBCDIC
<dia-y>	ÿ	lower	FF	DF

The remaining classes are defined as follows:

alpha — The character belongs to the class `upper` or `lower`.

alnum — The character belongs to the class `alpha` or `digit`.

print — The character belongs to the class `alnum` or `punct` or is the character <space>.

graph — The character belongs to the class `alnum` or `punct`.

The diagrams `toupper` and `tolower` illustrate the usual behavior:
<XYZ> becomes <xyz> and <xyz> becomes <XYZ>.

LC_COLLATE

As under UNIX, only the characters of the 7-bit code and the umlauts used in German are taken into account for the sort sequence. The umlauts are treated as equal to their base vowel; the umlauts follow their respective base vowel in their secondary weighting. The character ‘ß’ has the ASCII value X’DF’ (EBCDIC: X’59’).
Apart from this, the sequence corresponds to that of the ASCII character set.

LC_NUMERIC

decimal_point: ","
thousands_sep: "."
grouping: 0;0

LC_TIME

The German language is used for the names of days and months.
The abbreviated weekday names are: So, Mo, Di, Mi, Do, Fr, Sa.
The abbreviated month names are: Jan, Feb, Mär, Apr, Mai, Jun, Jul, Aug, Sep, Okt, Nov, Dez.

am_pm: “AM”, “PM”

Date and time 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 (%r) `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	"yes"
nostr	"no"
quitstr	"quit"
noexpr	"^[nN]"
yesexpr	"^[yY]"
quitexprr	"^[qQ]"

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

6.6 User-specific locales

Users can 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. The structure of this data is described below. The data can be changed to the desired values.

The following modification 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. It must be a valid entry name.

In the C source program, only the name USERLOC need be modified 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 is used when the `setlocale` library function is called to identify the user-specific locale (as a string in the second parameter).

The modified source programs can be compiled or assembled with the C/C++ compiler or with the Assembler (also ASSGEN).
If the module is not stored in the library SYSLNK.CRTE but in another PLAM library, this library must be assigned with the following SET-FILE-LINK command before the C program is started:

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

Structure of the data for the various locale categories

LC_COLLATE

The sort sequence is determined by a table (COLL/uscol) which defines the sort rating of each character by means of a weighting. The initial values are the characters' own hexadecimal values, i.e. the sort sequence corresponds to the EBCDIC sequence.

LC_CTYPE

There are three tables which define the classification and the conversion from uppercase to lowercase and vice versa 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:

```
                    Assembler program    C program

Uppercase letter          X'01'             _U
Lowercase letter          X'02'             _L
Decimal digit             X'04'             _N
Space                     X'08'             _S
Special character         X'10'             _P
Control character         X'20'             _C
Hexadecimal character       X'40'               _X
```

The C values are defined in the include file <ctype.h>.

The tables for converting from uppercase to lowercase letters (LOWER/uslow) and from lowercase to uppercase letters (UPPER/usupp) indicate the character resulting from conversion for each character from X'00' to X'FF'. These tables are used by the `toupper` and `tolower` macros for converting to uppercase and lowercase letters. The table needs to be filled only for characters which are classified as uppercase or lowercase letters in the classification table.

LC_NUMERIC, LC_MONETARY

A string with a maximum of 8 characters is provided for all information of type char *. These strings must always be terminated with a null byte.

LC_TIME

Strings with a maximum of 12 characters are provided for the days of the week and the months of the year.

7 Alphabetical reference

Here you will find, in alphabetical order, descriptions of all C functions and macros that are made available to you by the C runtime system.

Explanation of the function descriptions

All the function descriptions are based on a uniform principle, which is explained below.

The description of a function is divided into the following information categories:

- Function name and brief description
- Definition and general description
- Parameters
- Return value
- Notes
- Record I/O
- Example
- See also

Some of the above-mentioned sections may be omitted if they are not relevant to the function concerned or if the pertinent information (e.g. the data type of a parameter) is already evident from the syntax of the function call.

Definition and general description

The function definition includes the following information:

– the name of the include file required for the function

– the function header (data type and name of the function, list of formal parameters).

Below this syntax you will find a general description of how the function works.

Parameters

In the case of complex functions, the function definition is followed by a detailed description of the parameters. This includes their meanings, possible values, associated effects, etc.

Parameters are differentiated into input parameters and result parameters. In the case of result parameters, as opposed to input parameters, the contents of variables transferred during the call are modified by the function. One also speaks of “implicit” function results in this context. Result parameters are defined as pointers to an object without the qualifier “const”. For result parameters you must always specify the variable address, i.e. a pointer argument, when you call the function. In addition, sufficient memory space must be allocated for arrays, string variables, and structures.

Return value

The possible function return values are listed here. If the return value indicates an error, you will find an additional note stating which error code, if any, is stored in the `errno` variable.

Notes

In this section, you will find information on the following:

– possible sources of error (always the first item)
– programming and application tips
– interrelationship with other functions
– technical details regarding how the function works
– special points pertaining to BS2000.

Record I/O

This section is included for all input/output functions which can also be used on files with record-oriented input/output. It supplements the general “Notes” (principally formulated for stream-oriented input/output) with special notes applicable to record I/O (cf. section “Binary file” on page 60, “Stream-oriented input/output” on page 64, “Record-oriented input/output” on page 64).

Example

Short example illustrating the application of the described function.

See also

References to the names of related functions.

_a2e, _e2a - Convert from ASCII to EBCDIC and EBCDIC to ASCII

Definition

```
#include <ascii_ebcdic.h>

char*_a2e (char* z);

char*_e2a (char* z);
```

The functions `_a2e` and `_e2a` convert the (null-terminated) string *z* passed as a parameter from ASCII to EBCDIC and vice versa. The conversion takes place on the spot with the help of conversion tables. The corresponding data areas therefore have to be writable.

The conversion tables are declared as follows:

```
unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];
```

Parameters

char* z
String in ASCII or EBCDIC encoding to be converted

Return val.

The string *z* passed as a parameter, after its conversion to EBCDIC or ASCII code

See also

_a2e_n, _e2a_n, _a2e_max, _e2a_max, _a2e_dup, _e2a_dup, _a2e_dup_n, _e2a_dup_n

_a2e_dup, _e2a_dup - Convert from ASCII to EBCDIC and EBCDIC to ASCII

Definition

#include <ascii_ebcdic.h>

char*_a2e_dup (const char* z);

char*_e2a_dup (const char* z);

The functions `_a2e_dup` and `_e2a_dup` create a new string by taking the string *z* passed as a parameter and converting it from ASCII to EBCDIC or vice versa. The memory for the new string is allocated by means of `malloc()`, and it is up to the user to release it. If the available memory is insufficient, NULL is returned as the result. Otherwise, the new string is returned.

The conversion tables are declared as follows:

unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];

Parameters

char* z
String in ASCII or EBCDIC encoding to be converted

Return val.

New EBCDIC or ASCII string (if successful)

NULL, if there is insufficient memory

See also

_a2e, _e2a, _a2e_n, _e2a_n, _a2e_max, _e2a_max, _a2e_dup_n, _e2a_dup_n

_a2e_dup_n, _e2a_dup_n - Convert from ASCII to EBCDIC and EBCDIC to ASCII

Definition

#include <ascii_ebcdic.h>

char*_a2e_dup_n (const char* z, size_t n);

char*_e2a_dup_n (const char* z, size_t n);

The functions `_a2e_dup_n` and `_e2a_dup_n` create a new string by taking *z* and converting precisely *n* characters from ASCII to EBCDIC and vice versa. The memory for the new string is allocated by means of `malloc()`, and it is up to the user to release it. If the available memory is insufficient, NULL is returned as the result. Otherwise, the new, null-terminated string is returned.

The conversion tables are declared as follows:

unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];

Parameters

const char* z
String in ASCII or EBCDIC encoding to be converted

size_t n
Number of characters to be converted in the string *z*

Return val.

New EBCDIC or ASCII string (if successful)

NULL, if there is insufficient memory

See also

_a2e, _e2a, _a2e_max, _e2a_max, _a2e_n, _e2a_n, _a2e_dup; _e2a_dup

_a2e_max, _e2a_max, - Convert from ASCII to EBCDIC and EBCDIC to ASCII

Definition

#include <ascii_ebcdic.h>

char*_a2e_max (char* z, size_t n);

char*_e2a_max (char* z, size_t n);

The functions `_a2e_max` and `_e2a_max` convert the string *z* passed as a parameter with a maximum length of *n* from ASCII to EBCDIC or vice versa. If *z* contains a NULL character at a position < *n*, the conversion is terminated. The conversion takes place on the spot with the help of conversion tables. The corresponding data areas thus have to be writable.

The conversion tables are declared as follows:

unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];

Parameters

char* z
String in ASCII or EBCDIC encoding to be converted

size_t n
Maximum number of characters (left-aligned) to be converted in *z*

Return val.

The string *z* passed as a parameter, after its conversion to EBCDIC or ASCII code

See also

_a2e, _e2a, _a2e_n, _e2a_n, _a2e_dup, _e2a_dup, _a2e_dup_n, _e2a_dup_n

_a2e_n, _e2a_n - Convert from ASCII to EBCDIC and EBCDIC to ASCII

Definition

```
#include <ascii_ebcdic.h>

char*_a2e_n (char* z, size_t n);

char*_e2a_n (char* z, size_t n);
```

The functions `_a2e_` and `_e2a_n` convert the (null-terminated) string *z* passed as a parameter with a length of *n* from ASCII to EBCDIC or vice versa. Conversion takes place on the spot. The corresponding data areas thus have to be writable.

The conversion tables are declared as follows:

```
unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];
```

Parameters

char* z
String in ASCII or EBCDIC encoding to be converted

size_t n
Number of characters to be converted in the string *z*

Return val.

The string *z* passed as a parameter, after its conversion to EBCDIC or ASCII

See also

_a2e, _e2a, _a2e_max, _e2a_max, _a2e_dup, _e2a_dup, _a2e_dup_n, _e2a_dup_n

abort - Abnormal program termination

Definition

```
#include <stdlib.h>

void abort(void);
```

`abort` triggers the SIGABRT signal. If the program does not provide a routine for signal handling or if such a routine returns to the point of the interrupt, the program is aborted with `_exit(-1)`.
Any termination routines registered with `atexit` are not called and open files are not closed.

See also atexit, exit, _exit, raise, signal

abs - Absolute value of a whole number

Definition

#include <stdlib.h>

int abs(int i);

`abs` calculates the absolute value of the integer *i*.

Return val.

`|i|` for any given integer value *i*.

Note

The absolute value of the highest presentable negative number cannot be presented. If the highest negative number (-2^{31}) is specified as argument *i*, the program is terminated with an error.

Example

The following program outputs the absolute value corresponding to an input value.

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

int main(void)
{
  int i;
  printf("Please enter int value: \n");
  if (scanf("%d", &i) == 1)          /* Checks number of entries */
     printf("i = %d; •i• = %d\n", i, abs(i));
  else
     printf("Input error! \n");
  return 0;
}
```

See also

cabs, fabs, labs, llabs

acos - Arc cosine

Definition

#include <math.h>

double acos(double x);

`acos` is the inverse function of `cos` and calculates the corresponding angle in radians for a number in the interval [-1.0, +1.0].

Return val.

arc cosine(x)	a floating-point number of type `double` from [0, pi] for values *x* in the interval [-1.0, +1.0].
0	for values outside the interval [-1.0, +1.0]. In addition, `errno` is set to EDOM (domain error, i.e. argument too large).

Example

The following program prints the corresponding arc cosine values for input values in the interval [0.0, 1.0]:

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

int main(void)
{
  double x;
  for(x = 0.0; x < 1.0; x = x + 0.1)
  printf("x = %g : acos(%g) = %g\n", x, x, acos(x));
  return 0;
}
```

See also

cos, sin, tan, asin, atan, atan2

alarm - Set alarm clock

Definition #include <signal.h>

unsigned int alarm(unsigned int sec);

`alarm` triggers the signal SIGALRM for the calling program when the specified time span *sec* (passed as an argument) has elapsed. SIGALRM corresponds to the STXIT event class RTIMER (real-time interval timer). The program is terminated with `exit(-1)` if the signal is not intercepted (see also `signal`).
`alarm` calls with the value 0 - `alarm(0)` - do not trigger an alarm but set the alarm clock to 0 and cancel any pending alarms.

Return val. Time remaining in the alarm clock before execution of the `alarm` call.

Notes A number of `alarm` calls in succession resets the alarm clock with each call.

Since the alarm clock has a 1-second pulse, there may be time shifts of up to a second when the signal is triggered.

If the signal is intercepted (see `signal`), the restart of the interrupted program (i.e. the base process) may be delayed on priority grounds.

With the assignment: `i = alarm(0)` you can turn off the alarm clock and additionally ascertain how much time would have remained since the last alarm request.

Example The following program sends an asterisk to the standard output approx. every two seconds:

```
#include <stdio.h>
#include <signal.h>
void f(int sig)          /* Signal handling for SIGALRM */
{
  printf("*\n");
  alarm(2);            /* Resetting of alarm clock; all further asterisks */
}
int main(void)
{
  signal(SIGALRM + SIG_PS, f);
  alarm(2);            /* First asterisk */
  for(;;)
    ;
  return 0;
}
```

See also signal, sleep

asctime - Date and time

Definition

#include <time.h>

char *asctime(const struct tm *tm_p);

`asctime` converts a time specification coded in accordance with the structure `tm` (see below) into a 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.

Parameters

const struct tm *tm_p Structure as in the include file <time.h>:

```
struct tm
{
       int   tm_sec;        /* seconds (0-59) */
       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 from start of year (0-11) */
       int   tm_year;       /* years since 1900 */
       int   tm_wday;       /* weekday (0-6, Sunday=0) */
       int   tm_yday;       /* day since January 1 (0-365) */
       int   tm_isdst;      /* daylight saving time flag */
};
```

Return val.

Pointer to the string generated.

The resulting string has a length of 26 (including the null byte) and is formatted as a date and time specification:
Weekday Month Day Hrs:Min:Sec Year,
e.g. `Fri Apr 29 12:01:20 2011\n\0`

NULL In the event of an error

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 as the result by the `gmtime`, `gmtime64`, `localtime` and `localtime64` functions.

The calls `asctime(localtime(sec_p))` and `ctime(sec_p)` are equivalent. In the same way the calls `asctime(localtime64(sec_p))` and `ctime64(sec_p)` are equivalent.

Example

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

struct tm *t;
char *s;
time_t clk;

int main(void)
{
   clk = time((time_t *) 0);
   t = gmtime(&clk);
   printf("Year: %d\n", t->tm_year + 1900);
   printf("Time in hours: %d\n", t->tm_hour);
   printf("Day of the year: %d\n", t->tm_yday);

   s = asctime(t);
   printf("%s", s);
   return 0;
}
```

See also ctime, ctime64, gmtime, gmtime64, localtime, localtime64, mktime, mktime64, time, time64

asin - Arc sine

Definition #include <math.h>

double asin(double x);

`asin` is the inverse function of `sin` and calculates the corresponding angle in radians for a number in the interval [-1.0, +1.0].

Return val.

arc sine(x)	a floating-point number of type `double` within [-pi/2, +pi/2] for values *x* in the interval [-1.0, +1.0].
0	for values outside of [-1.0, +1.0]. In addition, `errno` is set to EDOM (domain error, i.e. argument too large).

Example The following program calculates and prints the corresponding arc sine values for 0.0, 0.1,..., 1.0:

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

int main(void)
{
   double x;
   for(x = 0.0; x < 1.0; x = x + 0.1)
   printf("x = %g : asin(%g) = %g\n", x, x, asin(x));
   return 0;
}
```

See also sin, cos, acos, tan, atan, atan2

assert - Macro for diagnostics

Definition

```
#include <assert.h>
void assert(int expression);
```

The `assert` macro determines whether a given *expression* is false (zero) at a particular point in the program. If this is the case, the program is terminated with `abort`, and the following comment is printed on the standard error output (`stderr`):

```
"CCM0009 Assertion failed: file xyz, line nnn"
```

xyz is the name of the source file; *nnn* is the line number of the line with the assert call.

Note

`assert` calls are ignored in the program (i.e. not executed) if you compile the program with the following compiler option:

SOURCE-PROPERTIES = PARAMETERS(DEFINE = NDEBUG)

See also

abort

atan - Arc tangent

Definition #include <math.h>

double atan(double x);

`atan` is the inverse function of `tan` and calculates the corresponding angle in radians for the floating-point number *x*.

Return val. arc tangent(x) a floating-point number of type `double` from the interval [-pi/2, +pi/2].

Example The following program calculates and prints the arc tangent of an input value:

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

int main(void)
{
  double x;
  printf("For which number do you want ATAN computed?: \n");
  if( scanf("%lf", &x) == 1)   /* Verifies the input of a number */
    printf("x = %g : atan(%g) = %g\n", x, x, atan(x));
  return 0;
}
```

See also atan2, tan, sin, asin, cos, acos

atan2 - Arc tangent of x/y

Definition #include <math.h>

double atan2(double x, double y);

`atan2` calculates the arc tangent of *x/y*. The signs of the two arguments determine the resulting quadrants.

Return val. arc tangent(*x/y*)
a floating-point number of type `double` in the interval [-pi/2, +pi/2].
If the divisor *y* is equal to 0, `atan2` returns either -pi/2 or +pi/2, depending on the sign of the dividend.

0 if the dividend *x* is equal to 0.

pi/2 if both arguments are equal to 0. `errno` is set to EDOM (domain error).

Example The following program reads in the arguments *x* and *y* and prints the computed arc tangent of *x/y*.

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

int main(void)
{
  double x, y;
  printf("Example of ATAN2(x/y)\n");
  printf("Enter x and y please:\n");
  if (scanf("%lf %lf", &x, &y) == 2)
       printf("ATAN2 (%g / %g) = %g\n", x, y, atan2(x, y));
  return 0;
}
```

See also atan, tan, sin, asin, cos, acos

atexit - Register termination routines

Definition

#include <stdlib.h>

int atexit(void (*funct) (void));

atexit is used to register a function *funct* that is to be executed when the program terminates normally.

Return val.

0	on successful registration of the function.
≠ 0	on error.

Notes

Up to 40 functions can be registered. The functions are called in the reverse order of their registration. If a function is registered more than once it is also called more than once.

The functions registered with atexit are only called if the program is terminated “normally” in one of the following ways:

- by explicitly calling the exit function
- on termination of the main function without an explicit exit call
- on termination of the program by the C runtime system with exit(-1), in other words: on the occurrence of a raise signal (not SIGABRT) which is either not processed or is processed by the signal default function SIG_DFL (see signal).

Only when all the termination routines have been processed are any files still open automatically closed.

Example

The termination routines *end1* and *end2* are registered with `atexit` and executed in the order *end2*, *end1* when the `main` function terminates.

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

void end1(void);
void end2(void);

int main(void)
{
   atexit(end1);
   atexit(end2);
   printf("main function\n");
   return 0;
}

void end1(void)
{
   printf("end1 routine\n");
}

void end2(void)
{
   printf("end2 routine\n");
}
```

See also

exit, raise, signal

atof - Convert a string into a floating-point number (double)

Definition #include <stdlib.h>

double atof(const char *s);

atof converts a string to which *s* points into a floating-point number of type double. The string to be converted may be formatted as follows:

```
 ⎧tab⎫     ⎧+⎫                              ⎧E⎫ ⎧+⎫
[⎨   ⎬...][⎨ ⎬][digit...][.][digit...][⎨ ⎬[⎨ ⎬]digit...]
 ⎩ ␣ ⎭     ⎩-⎭                              ⎩e⎭ ⎩-⎭
```

All control characters for white space are legal for *tab* (see definition of white space under isspace).

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.

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. In addition, errno is set to ERANGE (result too large).

Notes The decimal point (or comma) 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: it cuts off the numeric part, converts it according to the above description, and ignores the rest.

Example The following program converts a string passed in the call (Enter Options) into the corresponding floating-point number.

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

int main(int argc, char *argv[])

               /* Numbers are passed as strings!! A conversion is */
               /* required if the numeric value is needed */
{
  printf("floating : %f\n", atof(argv[1]));
  return 0;
}
```

See also atoi, atol, strtod, strtol, strtoul

atoi - Convert a string into a whole number (int)

Definition

#include <stdlib.h>

int atoi(const char *s);

`atoi` converts a string to which *s* points into an integer. The string to be converted may be formatted as follows:

```
 ⎧tab⎫     ⎧+⎫
[⎨   ⎬...][⎨ ⎬]digit...
 ⎩ ␣ ⎭     ⎩-⎭
```

All control characters for white space are legal for *tab* (see definition of white space under `isspace`).

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.

Note

`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.

Example

The following program converts a string passed in the call (Enter Options) into the corresponding integer value.

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

int main(int argc, char *argv[])

               /* Numbers are passed as a string!! A conversion is */
                  required if the numeric value is needed. */
                 {
   printf("integer : %d\n", atoi(argv[1]));
   return 0;
}
```

See also

atof, atol, strtod, strtol, strtoul

atol - Convert a string into a whole number (long)

Definition

#include <stdlib.h>

long int atol(const char *s);

`atol` converts a string to which *s* points into an integer of type `long`. The string to be converted may be formatted as follows:

```
 ⎧tab⎫      ⎧+⎫
[⎨   ⎬...][⎨ ⎬]digit...
 ⎩ ␣ ⎭      ⎩-⎭
```

All control characters for white space are legal for *tab* (see definition of white space under `isspace`).

Return val.

Integer value of type `long int`
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.

Note

`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.

Example

The following program converts a string passed in the call (Enter Options) into the corresponding integer value.

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

int main(int argc, char *argv[])

               /* Numbers are passed as a string!!
                  A conversion is required if the
                  numeric value is needed. */
{
   printf("long integer : %ld\n", atol(argv[1]));
   return 0;
}
```

See also

atof, atoi, atoll, strtod, strtol, strtoll, strtoul, strtoull

atoll - Convert a string into a whole number (long long int)

Definition

#include <stdlib.h>

long long int atoll(const char *s);

`atoll` converts a string, to which *s* points, into a whole number of type `long long int`. The string to be converted may be formatted as follows:

```
  ⎧tab⎫      ⎧+⎫
[{     }...][{   }]digit...
  ⎩ ␣ ⎭      ⎩-⎭
```

All control characters for white space are permitted for *tab* (see definition of white space under `isspace`).

Return val.

Integer value of type `long long int`
for strings formatted as described above and representing a numeric value.

0 for strings that do not correspond to the syntax described above.

LLONG_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 *p* is a NULL pointer and *base* is equal to 10, the only difference between `atoll` and the `strtoll` function lies in the error handling.
`atoll(s)` corresponds to `strtoll(s, (char **)NULL, 10)`.

The C compiler that supports the data type `long long` only creates objects in LLM format. For this reason, the `long long` library functions are also only available as LLMs and are not contained in the prelinked modules. Like data modules, they must either be integrated or reloaded from the library.

See also

atof, atoi, atol, strtod, strtol, stroll, strtoul, stroull

bs2cmd - Execute BS2000/OSD commands by means of the CMD macro

Definition

#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/OSD command by means of the BS2000/OSD 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 K 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_TRACE
Internal debug flag for outputting the internal buffer.

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

Definition

```
#include <stdlib.h>

void bs2exit(int status, const char *monjv_rcode);
```

`bs2exit` terminates the program.
Before this is done, all files opened by the program are closed, and the following messages are output to `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).

The status indicator of the monitoring job variable (1st to 3rd byte) is set to the value "$T " or "$A " in accordance with the first *status* parameter.

The return code of the MONJV (4th - 7th byte) can additionally be supplied with the *monjv_rcode* parameter.

Parameters

int status
see `exit` function.

const char *monjv_rcode
This parameter can be used to specify a pointer to 4 bytes of data (the return code), which is loaded in the MONJV when the program terminates.

Notes

When a program is terminated with `bs2exit` the termination routines registered with `atexit` are not called (cf. `exit`).

In order to set and query monitor job variables, you must start the C program with the following command:

```
/START-PROG program,MONJV=monjvname
```

The content 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.

Example The program is terminated and the return code is set

```
#include <stdio.h>

int main(void)
{
  .
  .
  .
  if(error)
    bs2exit(-1, "ABCD");
}
```

See also exit, _exit

bs2fstat - Access file name from catalog

Definition #include <stdlib.h>

int bs2fstat(const char *pattern, void (*fct)(const char *f_name, int len));

`bs2fstat` returns

– the fully qualified file names (:catid:$userid.filename) of one or more files that satisfy the selection criterion given by *pattern*, and

– the length of the particular file name including the terminating null byte (\0).

For each file found, `bs2fstat` calls a function *fct* (which must be supplied by the user) and passes to it the particular file name *f_name* (string char *) and the name length *len* (integer) as current arguments.

If no file matches the selection criterion *pattern* or if *pattern* is errored the function *fct* is not called and `bs2fstat` returns a DMS error message.

Parameters const char *pattern

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 can also be specified to determine which files are selected, e.g.:

– file and catalog attributes (FCBTYPE, SHARE etc.)

– creation and access date (CREATE, 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 (*fct)(const char *f_name, int len)

A user-supplied function with the parameters *f_name* (file name) and *len* (name length). These parameters are supplied with current values by `bf2stat` on each function call. The function calls are made automatically by `bs2fstat` (in a while loop).

Return val. 0 if the call was successful.

DMS error message code
if the call was not successful.

Note The DMS error message code can be only queried from outside the user-own function *fct*, since the function is not called if the search was unsuccessful (see also example).

Example In the following program, all files matching the name pattern entered by the user are made shareable with the MODIFY-FILE-ATTRIBUTES command.

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

void share(const char *, int);

int main(void)
{
  char name[54];
  int result;
  printf("Which files are to be made shareable?\n");
  gets(name);
  result = bs2fstat(name, share);
  if(result != 0)
     printf("Error code: DMS%x\n", result);
  return 0;
}

void share(const char *nam, int len)
                        /* The formal parameters nam and len are      */
                        /* supplied as current parameters by bs2fstat */

{
  char cmd[200];
  strcpy(cmd, "/MODIFY-FILE-ATTRIBUTES ");
  strcat(cmd, nam);
  strcat(cmd, ",PROTECTION=PAR(USER-ACCESS=ALL-USERS)");
  system("/MODIFY-TERMINAL-OPTIONS OVERFLOW-CONTROL=NO-CONTROL");
  printf("%s\n", cmd);
  system(cmd);
}
```

See also system

bsearch - Binary search algorithm

Definition

```
#include <stdlib.h>

void *bsearch(const void *search, const void *field, size_t n,
              size_t elsize, int (*comp) (const void *, const void *));
```

The `bsearch` function is a binary search function. `bsearch` searches the number *n* of elements of an array *field* for the value in the data item *search*. Each array element is *elsize* bytes long. The array elements must already be sorted in ascending order as expected by the comparison function *cmp*.

cmp is a user-supplied comparison function which is called by `bsearch` with two arguments, a pointer to *search* (argument 1) and a pointer to an array element (argument 2). *cmp* supplies an integer as the result. The result is interpreted as follows:

< 0	argument1 is less than argument2
= 0	argument1 and argument2 are equal
> 0	argument1 is greater than argument2

Return val. Pointer to the array element found.

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.

Note If, for example, the `qsort` function is used for sorting the array, it makes sense to use the same comparison function *cmp* that is used by `bsearch`. The current arguments of `qsort` are then pointers to two array elements to be compared.

See also qsort

btowc - Convert (one-byte) multibyte character to wide character

Definition

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

wint_t btowc(int c);
```

`btowc` converts a multibyte character *c*, which must consist of one byte and be in the initial shift state, to a wide character.

Return val. Wide character, if successful.

WEOF	if *c* has the value EOF or if `(unsigned char)c` does not represent a valid (one-byte) multibyte character in the initial shift state.

Note This version of the C runtime system only supports one-byte characters as wide character codes or multibyte characters.
The shift state of the multibyte character is ignored.

See also mblen, mbtowc, wcstombs, wctomb

cabs - Absolute value of a complex number

Definition #include <math.h>

double cabs(_ _complex z);

`cabs` calculates the absolute value of the 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. `sqrt(z.x * z.x + z.y * z.y)`
i.e. the absolute value of the complex number *z*.

In the case of an overflow, the program aborts (signal SIGFPE)!

Example The following program calculates the absolute value of a complex number.

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

int main(void)
{
 __complex z;
 if (scanf("%f %f", &z.x, &z.y) == 2)
     printf("%f : Absolute value\n", cabs(z));
 return 0;
}
```

See also abs, fabs, labs, llabs, sqrt

calloc - Reserve memory space

Definition

<stdlib.h> is included via:

#include <stdlib.h>

void *calloc(size_t n, size_t elsize);

`calloc` provides contiguous memory space at execution time for an array with *n* elements, where each element requires *elsize* bytes. `calloc` initializes each element of the new array with binary zeros.

`calloc` is part of a C-specific memory management package which internally manages requested and released memory areas. Wherever possible, new requests are met first from areas already being managed and only then by the operating system (cf. `garbcoll` function).

Return val.

Pointer to the new memory space
if sufficient memory space is present.

NULL pointer if memory space does not suffice for the request.

Notes

The new data area begins on a doubleword boundary.

To ensure that you are requesting the correct size for an array element, you should use the `sizeof` operator for the calculation of *elsize*.

A serious disruption in working memory may be expected if the length of the memory area provided is exceeded when writing.

If *n* or *elsize* has the value 0, `calloc` returns an unambiguous address which can also be transferred to `free`.

Example

The following program fragment requests memory space for 20 array elements of type `long integer`.

```
#include <stdlib.h>

long *long_array;
    .
    .
long_array = (long *)calloc(20, sizeof(long));
```

See also

malloc, realloc, free, garbcoll

cdisco - Deactivate a contingency routine

Definition

#include <cont.h>

void cdisco(struct enacop *enacopar);

`cdisco` deactivates a contingency routine (TU or P1) defined with `cenaco`.

For detailed information on contingency routines, refer to chapter “Contingency and STXIT routines” on page 87ff and the "Executive Macros" manual.

Parameters

struct enacop *enacopar

Pointer to a structure which 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)(struct contp);      /* start adr of cont.rout. */
  int  comess;                      /* contingency message */
  int coidret;                      /* 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` entry (identifier of the contingency process) in the structure.

Entries supplied by `cdisco`:

secind "Secondary Indicator", as stored in the most significant byte of register 15 (values X’10’ or X’16’) after execution of the DISCO macro.

rcode1 "Return Code", as stored in the least significant byte of register 15 (values 0 or 4) after execution of the DISCO macro.

Note The Assembler macro DISCO locks the contingency routine only for future event requests. However, if an event that was requested earlier occurs after DISCO, the contingency routine will be called even after DISCO.
Note that calls to the contingency routine `econt` are suppressed even for events that were requested earlier.

See also cenaco

ceil - Round up

Definition #include <math.h>

double ceil(double x);

`ceil` rounds up a floating-point number to the lowest integer of type `double` that is greater than or equal to *x*.

Return val. Lowest integer of the type `double` which is greater than or equal to *x* if successful.

HUGE_VAL in the event of an overflow, `errno` is also set to ERANGE (result too high).

Example

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

int main(void)
{
  double x;
  printf("Please enter the floating-point number to be rounded up:\n");
  if (scanf("%lf", &x) == 1)
      printf("The number %g is being rounded up to %f\n", x, ceil(x));
  return 0;
}
```

See also abs, fabs, floor

cenaco - Definition of a contingency routine

Definition #include <cont.h>

void cenaco(struct enacop *enacopar);

`cenaco` defines a contingency routine (TU or P1). This means that a routine written by the user can be assigned as a contingency routine by means of `cenaco`.

For detailed information on contingency routines, refer to chapter “Contingency and STXIT routines” on page 87ff and the "Executive Macros" manual.

Parameters struct enacop *enacopar

Pointer to a structure that 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)(struct contp);       /* start adr of cont.rout. */
  int  comess;                       /* contingency message */
  int  coidret;                      /* 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 of the entries in the parameter structure can or must be supplied by you prior to the cenaco call; other entries are used by cenaco to store information during the run.

Entries supplied by the user:

coname
: Name of the contingency process. The name is a maximum of 54 bytes long (without null byte), must be in uppercase and must terminate 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* (see also example).
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, 12, 16 or 24) 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.

Note A maximum of 255 contingency routines can be defined.

Example Program fragment for the definition of a contingency routine:

```
#include <cont.h>

/* Contingency routine: controut */

int controut(struct contp contpar)
{
    .
    .
    .
  printf("Contingency message: %d\n", contpar.comess);
    .
    .
    .
}

/* Main routine in which the controut routine is defined as a
   contingency routine. */

int main(void)
{
    .
    .
    .
  struct enacop enacopar;
    .
    .
    .
  enacopar.econt  = controut;
  enacopar.level  = 1;
  enacopar.comess = 100;
  strfill(enacopar.coname, "CONTPROC1 ", sizeof(enacopar.coname));
  cenaco(&enacopar);
    .
    .
    .
}
```

See also cdisco, cstxt, signal, alarm, raise, sleep

clearerr - Clear end-of-file and error flag

Definition #include <stdio.h>

void clearerr(FILE *fp);

`clearerr` clears the end-of-file and error information of the file with the file pointer *fp*.

Note `clearerr` is implemented as a macro and as a function (see section “Functions and macros” on page 17).

Record I/O `clearerr` can also be used on files with record I/O.

See also feof, ferror

clock - CPU time used since the program call

Definition #include <time.h>

clock_t clock(void);

`clock` supplies the CPU time which has elapsed since the program was called.

Return val. The CPU time in ten thousandths of a second since the program was called if successful.

`(clock_t)` -1 if the time cannot be calculated or represented.

Notes `clock` is implemented as a macro and as a function (see section “Functions and macros” on page 17).

To obtain the time in seconds, the result of clock must be divided by the value of the CLOCKS_PER_SEC macro.

Example

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

int main(void)
{
  clock_t result;
  result = clock();
  printf("used cputime %f seconds\n", ((float)result / CLOCKS_PER_SEC));
  return 0;
}
```

See also cputime

close - Close file and flush buffer (elementary)

Definition #include <stdio.h>

int close(int fd);

`close` closes a file that was opened by `open`/`open64` or `creat`/`creat64`. Before closing the file, `close` calls the `fflush` function, which flushes the buffer.

Return val. 0 `close` has closed the file with the file descriptor *fd*.

-1 The file descriptor is unknown or no file is open for this file descriptor. In addition, `errno` is set to EBADF (invalid file descriptor).

Notes Upon termination of a program (normal or with `exit`), all open files are automatically closed.

A maximum of _NFILE files may be open simultaneously per program. _NFILE is defined as 2048 in <stdio.h>. Programs that process more files must therefore temporarily close unused ones.

If the file was opened with the standard I/O function `fopen` or `fopen64`, it must be closed with `fclose` instead of `close`.

Example see example under `lseek`/`lseek64`

See also creat, creat64, fclose, fflush, open, open64, exit

cos - Cosine

Definition

#include <math.h>

double cos(double x);

`cos` calculates the trigonometric function cosine for the floating-point number *x*.

Return val. `cos(x)` a floating-point number in the interval [-1.0, +1.0].

Example

The following program lists the cosine values corresponding to input values in the interval [-pi, +pi].

```
#include <math.h>
#include <stdio.h>
#define pi 3.14159265358979

int main(void)
{
   double x;
   for (x = -pi; x <= pi; x = x + pi/4.)
       printf("cos(%f) = %f\n", x, cos(x));
   return 0;
}
```

See also acos, cosh, sin, asin, sinh, tan, atan, atan2, tanh

cosh - Hyperbolic cosine

Definition
#include <math.h>

double cosh(double x);

`cosh` calculates the hyperbolic cosine for the floating-point number *x*.

Return val.

`cosh(x)`	for a floating-point number *x*.
+HUGE_VAL	if the result overflows. In addition, `errno` is set to ERANGE (result too large).

Example
The following program lists the hyperbolic cosine values corresponding to input values in the interval [-1.0, +1.0].

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

int main(void)
{
  double x;
  for (x = -1.0; x < 1.0; x = x + 0.1)
    printf("cosh(%f) = %f\n", x, cosh(x));
  return 0;
}
```

See also
acos, cos, sin, asin, sinh, tan, atan, atan2, tanh

cputime - CPU time used by the current task

Definition #include <stdlib.h>

int cputime(void);

`cputime` returns the CPU time used by the current task (since LOGON).

Return val. Integer indicating the CPU time consumed in ten thousandths of a second.

Example

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

int main(void)
{
float time_f;
time_f = (float)cputime() / 10000;
printf("cputime since logon: %f seconds\n", time_f);
return 0;
}
```

creat, creat64 - Create a new file (elementary)

Definition

#include <stdio.h>

int creat(const char *f_name, int mode);
int creat64(const char *f_name, int mode);

`creat` and `creat64` open a file for writing.

- If the file does not yet exist, it is created.
- Files that already exist are truncated to a length of 0.

`creat` and `creat64` return a file descriptor for subsequent elementary access operations (`write`, `read`).

There is no functional difference between `creat` and `creat64`, except that a large file identifier is stored with the file description that is linked to the file descriptor, i.e. the O_LARGEFILE bit is set. A file descriptor is returned that can be used to extend the file over 2 GB.

To process files > 2 GB, proceed as follows:

- If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `creat`. `creat64` is then used implicitly with the appropriate parameters.
- Otherwise, you have to call `creat64`.

Parameters

const char *f_name
A string specifying the name of the file to be opened. *f_name* can be:

- any valid BS2000 file name
- “link=*linkname*”
  linkname denotes a BS2000 link name

int mode
This parameter is ignored. However, it is required for the creation of portable programs since it controls the protection bit (rwx rwx rwx) assignment in the UNIX operating system.

Return val.

File descriptor	i.e.positive number used later to identify the file in elementary access operations (`read`, `write`).
-1	if the file could not be opened, e.g. because too many files are open or because *f_name* is not a valid file or link name.

Notes

The BS2000 file name or link name may be written in lowercase and uppercase letters. It is automatically converted to uppercase letters.

If a non-existent file is created, the following applies by default:
With KR functionality (applies to C/C++ versions prior to V3.0 only), a SAM file with variable record length and standard block length is created.
With ANSI functionality, an ISAM file with variable record length and standard block length is created.

By using a link name the following file attributes can be changed with the ADD-FILE-LINK command: access method, record length, record format, block length and block format. See also section “System files (SYSDTA, SYSOUT, SYSLST)” on page 70.

If an existing file is truncated to length 0, the catalog attributes of this file are preserved.

A maximum of _NFILE files may be open simultaneously. _NFILE is defined as 2048 in <stdio.h>.

Example The program given below writes the contents of an input file to an output file. The output file is created as a new file with creat. The name of this file as well as the file attributes are defined by means of a ADD-FILE-LINK command (link name=LINK).
The following command, for example, could be used to create an ISAM file named OUT.ISAM:

```
/ADD-FILE-LINK LINK-NAME=LINK,FILE-NAME=OUT.ISAM,ACCESS-METHOD=ISAM
```

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

int main(void)
{
   char name[50];
   char buf;
   int fin, fout;

   printf("Name of the input file?\n");
   gets(name);
   printf("File %s is being copied.\n", name);
   if ((fin = open(name,0)) == -1)
   {
     perror(name);
     exit(-1);
   }
   if ((fout = creat("link=link", 1)) == -1)
   {
     perror("link");
     exit(-1);
   }
   while(read(fin, &buf, 1) > 0)
   {
     putchar(buf);           /* Log to stdout */
     write(fout, &buf, 1);
   }
   close(fin); close(fout);
   return 0;
}
```

See also close, fdopen, open, open64, read, write, perror

cstxit, _cstxit - Definition of an STXIT routine

Definition #include <stxit.h>

void cstxit(struct stxitp stxitpar);

`cstxit` defines an STXIT routine. This means that a routine written by the user can thus be assigned as an STXIT routine.
Detailed information on the programming of STXIT routines is provided in chapter “Contingency and STXIT routines” on page 87ff and the "Executive Macros" manual.

`_cstxit` also defines an STXIT routine.
In contrast to `cstxit` with `_cstxit` a pointer to an `stxitpar` structure is passed instead of the structure itself. This allows `_cstxit` to set the structure element `err_set` such that it can also be evaluated by the calling program.

Parameters struct stxitp stxitpar

Structure in which the information required for the definition of an STXIT routine is to be specified. The structure is defined in <stxit.h> for ANSI-C as follows (`struct cont` depends on the compilation mode):

```
struct stxitp
{
  addr     bufadr;       /* Address of the message for the program */
                         /* (OPINT) */
  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 control */
  struct type typep;     /* Parameter transfer mode */
};

struct cont              /* Address of the STXIT routine for */
{                        /* the particular event class */
  void  (*prchk) (struct stxcontp stxcontpar);
  void  (*timer) (struct stxcontp stxcontpar);
  void  (*opint) (struct stxcontp stxcontpar);
  void  (*error) (struct stxcontp stxcontpar);
  void  (*runout) (struct stxcontp stxcontpar);
  void  (*brkpt) (struct stxcontp stxcontpar);
  void  (*abend) (struct stxcontp stxcontpar);
  void  (*pterm) (struct stxcontp stxcontpar);
  void  (*rtimer) (struct stxcontp stxcontpar);

};
```

```
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 data 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 this event class is not evaluated.

new_stx
: A new STXIT routine is assigned for the corresponding event class. The remaining data for this 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 this event class is not evaluated.

Diagnostic control:

ful_diag	For compatibility reasons the diagnostic control parameters
min_diag	are accepted syntactically but since conversion to ILCS are
no_diag	no longer evaluated. 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 in activating the STXIT routine.
mem_err	Error in the memory space request (when activating the STXIT routine).

Notes

You must supply the parameter structure `stxitpar` yourself.

To standardize initialization, a prototype (`stxit_pr`) has been defined and provided for you in the include file <stxit.h>. If you copy this prototype to one of your own defined structures of type `stxitp`; you will only need to set the fields for those event classes for which the assignment of an STXIT routine is to be changed.

For event class INTR, you must supply the address (`stxitpar.bufadr`) at which the information for the program is to be provided. The STXIT contingency routine can then fetch the message from this address and evaluate it.

See also

alarm, cenaco, raise, signal, sleep

ctime, ctime64 - Date and time (CET)

Definition

```
#include <time.h>

char *ctime(const time_t *sec_ptr);
char *ctime64(const time64_t *sec_ptr);
```

`ctime` and `ctime64` interpret the time specification to which *sec_ptr* points (see return values of `mktime`, `mktime64` and `time`, `time64`) as the number of seconds which have passed since the reference date (epoch). The functions calculate the local time (CET) from this and convert the result to a string. `ctime` and `ctime64` behave analogously to `localtime` and `localtime64`.

Negative values are interpreted as seconds before the reference date. The earliest displayable date is 01/01/1900 00:00:00 local time.

With `ctime` the reference date depends on the use of the TIMESHIFT bind option (see section “Time functions” on page 40):

– without TIMESHIFT bind option (default): 1/1/1950 00:00:00.
– with TIMESHIFT bind option: 1/1/1970 00:00:00.

With `ctime64` the reference date is always 1/1/1970 00:00:00

The latest date which can be displayed with `ctime` is 01/19/2018 03:14:07 (without TIMESHIFT bind option) or 01/19/2038 03:14:07 (with TIMESHIFT bind option).

Irrespective of the use of the TIMESHIFT bind option, `ctime64` can display dates up to 3/18/4317 02:44:48.

Return val. Pointer to the 26-character string generated.

The resulting string has a length of 26 (incl. the terminating null byte \0) and is formatted as a date and time of the form:
Weekday Month Day Hrs:Min:Sec Year,
e.g. `Fri Apr 29 12:01:20 2011\n\0`

NULL In the event of an error

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.

Time specifications are based on the 24-hour clock.

Example The following program converts an input value to local time and outputs the result in the form of a date and time.

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

int main(void)
{
   time_t sec;
   sec = time((time_t *)0);
   printf("%s",ctime(&sec));
   return 0;
}
```

See also asctime, gmtime, gmtime64, localtime, localtime64, mktime, mktime64, time, time64

_ _DATE_ _ - Output date of compilation (macro)

Definition _ _DATE_ _

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

Note This macro need not be defined in an include file. Its name is recognized and replaced by the compiler.

Example

```
#include <stdio.h>

int main(int argc, char *argv[])

{
printf("Program %s was compiled on %s at %s hours\n", argv[0], __DATE__,
__TIME__);
return 0;
}
```

See also asctime, _ _TIME_ _

difftime, difftime64 - Calculate time difference

Definition #include <time.h>

double difftime(time_t time2, time_t time1);
double difftime64(time64_t time2, time64_t time1);

`difftime` and `difftime` calculate the difference between *time2* and *time1*. The time values must be of type `time_t` or `time64_t`. Such time values are supplied for instance by the `mktime` and `time` or `mktime64` and `time64` functions.

Return val. Time difference in seconds in floating point representation.

See also time, time64, mktime, mktime64, ftime, ftime64, localtime, localtime64, asctime, gmtime, gmtime64

div - Division with integers (int)

Definition #include <stdlib.h>

div_t div(int dividend, int divisor);

`div` calculates the quotient and the remainder of the division of *dividend* by *divisor*.
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 equation: Quotient * Divisor + Remainder = Dividend

Return val. Structure of type `div_t`
containing both the quotient *quot* and the remainder *rem* as `integer` values.

Example

```
div_t d;

d = div( 7, 3);        /*  d.quot =  2,  d.rem =  1  */
d = div(-7, 3);        /*  d.quot = -2,  d.rem = -1  */
d = div( 7,-3);        /*  d.quot = -2,  d.rem =  1  */
d = div(-7,-3);        /*  d.quot =  2,  d.rem = -1  */
```

See also ldiv, lldiv

double2ieee - Convert floating-point number from /390 format to IEEE format

Definition

#include <ieee_390.h>

extern double double2ieee (double num);

`double2ieee` converts an 8-byte floating-point number *num* in /390 format to IEEE format and returns it as the result. Neither overflow nor underflow can occur, but up to three bit positions can be lost.

Parameters

double num
    8-byte floating-point number in /390 format

Return val.

8-byte floating-point number in IEEE format (in the event of success)

The global variable *float_exceptions_flag* contains information for the event of unsuccessful conversion and is defined as follows:

```
extern int float_exception_flags;
enum {
   float_flag_inexact   =  1,
   float_flag_divbyzero =  2,
   float_flag_underflow =  4,
   float_flag_overflow  =  8,
   float_flag_invalid   = 16
};
```

If bit positions are lost during conversion and the result is thus inaccurate, *float_flag_inexact* is set.

See also

ieee2double, float2ieee, ieee2float

ecvt - Convert a floating-point number to a string

Definition
#include <stdlib.h>

char *ecvt(double value, int n, int *dec_pt, int *sign);

`ecvt` converts a floating-point number *value* to a string of *n* digits and returns a pointer to this string as its result.

The string begins with the first non-zero digit of the floating-point number, i.e. leading zeros are not included.

The decimal point 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.

Parameters
double value
Floating-point value that is to be edited for output.

int n
Number of digits in the result string (calculated from the first non-zero digit of the floating-point number to be converted).

If *n* is less than the number of digits in *value*, the least significant digit is rounded.
If *n* is greater, zero padding is used for right justification.

int *dec_pt
Pointer to an integer specifying the position of the decimal point in the result string.

Positive number: position relative to the beginning of the result string.
Negative number or 0: the decimal point is to the left of the first digit.

int *sign
Pointer to an integer specifying the sign of the result string.

0: the sign is positive
Not equal to 0: the sign is negative

Return val.
Pointer to the converted string.
`ecvt` terminates the string with the null byte (\0).

Notes
An invalid parameter, such as an `integer` value instead of a `double` value, causes the program to abort!

Note that the arguments *dec_pt* and *sign* must be pointers!

`ecvt` writes its result into an internal C data area that is overwritten with each call! The `fcvt` function also uses the same data area.

Example

The following program reads a floating-point value *x*, converts it as specified in *n*, and outputs it as a string. In addition, the calculated *sign* and the position of the decimal point *dec_pt* are output.

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

int main(void)
{
  double x;
  int n, dec_pt, sign;
  char *s;
  printf("Please enter floating-point number:  \n");
  if (scanf("%lf", &x) == 1)
  {
    printf("How many significant digits?:  \n");
    if (scanf("%d", &n) == 1)
    {

      s = ecvt(x, n, &dec_pt, &sign);
      printf("The string is: %s\n", s);

      printf("The sign is %s \n",
             (sign == 0 ? "positive" : "negative"));

      printf("The position of the decimal point is %d \n", dec_pt);
    }
  }
  return 0;
}
```

See also

fcvt, gcvt

_edt - EDT call

Definition

#include <stdlib.h>

void _edt(void);

_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 function call.

Note

Programs that call the _edt function 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.

Example

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

int main(void)
{
   _edt();
   printf("Return to the C program\n");
   return 0;
}
```

erf - Error function (mathematical)

Definition #include <math.h>

double erf(double x);

`erf` calculates the error function for a floating-point number x as defined below:

$$\frac{2}{\sqrt{\Pi}} \int_{0}^{x} e^{-t^2} \, dt$$

Return val. `erf(x)`

See also erfc

erfc - Complement of error function (mathematical)

Definition #include <math.h>

double erfc(double x);

`erfc` calculates the complement of the error function for a floating-point number x and returns the value `1.0 - erf(x)`.

Return val. `1.0 - erf(x)`

Note The `erfc` function is provided because calculations of the error function with `erf` produce extremely inaccurate results for large values of x.

See also erf

exit, _exit - Program termination

Definition

```
#include <stdlib.h>

void exit(int status);

void _exit(int status);
```

`exit` terminates the program.

First, the termination routines registered with the `atexit` function are called in the reverse order of their registration. If a routine has been registered more than once it is also called more than once.
All files opened by the program are then closed and the following messages are output to `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).

`_exit` also terminates the program.

However, in contrast to `exit`, the termination routines registered with `atexit` are not called and open files are not closed. Only the message
"CCM0999 exit status" is output (if *status* ≠ EXIT_SUCCESS).

Depending on the value of the *status* parameter, the status indicator of the monitoring job variable (1st to 3rd byte) is set to the value "$T " or "$A ".

Parameters int status

This parameter may contain the following values:

– the symbolic constants EXIT_SUCCESS and EXIT_FAILURE defined in the include file <stdlib.h>, or

– any `integer` value.

EXIT_SUCCESS (value 0)
causes a program to terminate normally. The status indicator of the MONJV is set to the value "$T ".

EXIT_FAILURE (value 9990888)
results in a so-called job step termination, i.e.

- the program is terminated
- 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 set to the value "$A ".

`Integer` value
If this value is not equal to the predefined values EXIT_SUCCESS and EXIT_FAILURE (≠ 0 or ≠9990888), a job step termination is performed, and the status indicator of the MONJV is set to the value "$T ".

When this value corresponds to the predefined values EXIT_SUCCESS or EXIT_FAILURE, the actions stated above are performed.

Notes In order to be able to set and query monitoring job variables, you must start the C program with the following command:

`/START-PROG` *`program`*`,MONJV=`*`monjvname`*

The content 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 monitoring job variables can be found in the "Job Variables" manual.

See also abort, atexit, bs2exit, signal

exp - Exponential function

Definition

#include <math.h>

double exp(double x);

`exp` calculates the exponential function for permissible floating-point numbers *x*.

Return val.

e^x	if successful.
HUGE_VAL	if the result overflows. In addition, `errno` is set to ERANGE (result too large).

Example

The following program calculates e^x for an input value *x*.

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

int main(void)
{
  double x;
  printf("Please enter a floating-point number:\n");
  if (scanf("%lf", &x) == 1)
      printf("exp(%g) = %g\n", x, exp(x));
  return 0;
}
```

See also

log, log10, pow

fabs - Absolute value of a floating-point number

Definition

#include <math.h>

double fabs(double x);

`fabs` calculates the absolute value of a floating-point number *x*.

Return val. Absolute value of the argument: `|x|`

Example The following program calculates the absolute value of a floating-point number entered:

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

int main(void)
{
   double x;
   printf("Please enter a floating-point number:\n");
   if (scanf("%lf", &x) == 1)
       printf("•%g• = %g\n", x, fabs(x));
   return 0;
}
```

See also abs, cabs, ceil, floor, labs, llabs

fclose - Close a file and flush buffer

Definition #include <stdio.h>

int fclose(FILE *fp);

`fclose` closes the file to whose FILE structure the file pointer *fp* points and releases *fp*. Memory space that was dynamically allocated for this FILE structure (with `fopen` or `fopen64`) is also freed. `fclose` calls the `fflush` function before the file is closed.

Return val.

0	The file has been closed.
EOF	`fclose` was not successful, because – *fp* is not assigned to a file (file already closed) or – an error occurred when flushing the buffer.

Notes If the file pointer *fp* does not point to a FILE structure, the program aborts!

Whenever a program is terminated normally or by means of `exit`, an `fclose` is automatically executed for each open file. Therefore, you need not call `fclose` explicitly unless you want to close a file prior to program termination, e.g. to ensure that the limit for open files (=2048) is not exceeded.

Record I/O Since data is not buffered in the case of record I/O, there is no internal call to the `fflush` function.

Example The following program fragment closes the file pointed to by file pointer *fp* when the end of the file is reached.

```
FILE *fp;

if(feof(fp))
  fclose(fp);
```

See also fflush, close, fdopen, fopen, fopen64, exit

fcvt - Convert a floating-point number to a string

Definition

#include <stdlib.h>

char *fcvt(double value, int n, int *dec_pt, int *sign);

`fcvt` converts a floating-point *value* to a string of digits and returns a pointer to this string as the result. The output format corresponds to the FORTRAN F format.

The string begins with the first non-zero digit of the floating-point number to be converted and includes *n* decimal places.

The decimal point and a negative sign, if any, do not form a part of the string. However, `fcvt` returns the position of the decimal point and the sign in result parameters.

Parameters

double value
Floating-point value that is to be edited for output.

int n
Number of digits after the decimal point.

If *n* is less than the number of digits after the decimal point in *value*, the least significant digit is rounded (as in the FORTRAN F format).
If *n* is greater, zero padding is used for right justification.

int *dec_pt
Pointer to an integer that specifies the position of the decimal point in the result string.

Positive number: position relative to the beginning of the result string.
Negative number or 0: the decimal point is to the left of the first digit.

int *sign
Pointer to an integer that specifies the sign of the result string.

0: the sign is positive
Not equal to 0: the sign is negative

Return val.

Pointer to the converted string. `fcvt` terminates the string with the null byte (\0).

Notes

Invalid parameters, e.g. an `integer` value instead of a `double` value, cause the program to abort!

Note that the arguments *dec_pt* and *sign* must be pointers!

`fcvt` writes its result into an internal C data area that is overwritten with each call! The `ecvt` function also uses the same data area.

Example The following program reads a floating-point value *x*, converts it as specified in *n* according to the FORTRAN F format, and outputs it as a string. In addition, the calculated *sign* and the position of the decimal point *dec_pt* are output.

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

int main(void)
{
  double x;
  int n, dec_pt, sign;
  printf("Please enter floating-point number:  \n");
  if (scanf("%lf", &x) == 1)
  {
    printf("How many significant digits?:  \n");
    if (scanf("%d", &n) == 1)
    {
      printf("The converted number is :  %s \n",
             fcvt(x, n, &dec_pt, &sign));
      printf("The sign is %s \n",
             (sign == 0 ? "positive" : "negative"));
      printf("The position of the decimal point is: %d \n", dec_pt);
    }
  }
  return 0;
}
```

See also ecvt, gcvt

fdelrec - Delete record in ISAM file (record I/O)

Definition #include <stdio.h>

int fdelrec(FILE *fp, void *key);

`fdelrec` deletes the record with the key value *key* from an ISAM file with record I/O.

Parameters FILE *fp

File pointer of an ISAM file which was opened in the mode “type=record,forg=key” (cf. `fopen`/`fopen64`, `freopen`/`freopen64`).

void *key

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 have been read immediately prior to the `fdelrec` call.

Return val.

0	If the record with the specified key was deleted.
> 0	If the record to be deleted does not exist.
EOF	If an error has occurred.

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 read/write pointer 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, fopen64, freopen, freopen64

fdopen - Assign a file pointer to a file descriptor

Definition #include <stdio.h>

FILE *fdopen(int fd, const char *mode);

`fdopen` assigns a file pointer to a file (with file descriptor *fd*) that has already been opened with `open`/`open64` or `creat`/`creat64`.
Following an `fdopen` call, the file may also be processed with functions from the standard I/O library (`fread`, `fputc`, `fprintf` etc.).

Parameters int fd

File descriptor that was assigned by a `creat`/`creat64` or `open`/`open64` call.

const char *mode

String which specifies the access mode (see description under `fopen/fopen64`). This parameter is not evaluated, i.e. the file retains the original access mode that was specified for `open`/`open64` or `creat`/`creat64`. In other words, the access mode cannot be changed with `fdopen`.

Return val. File pointer to the assigned FILE structure
if successful.

Note If errors occur, e.g. due to an invalid file descriptor, `fdopen` returns neither a defined result nor an error message. The program does not abort either!

Example The following program opens the file *fname* for elementary as well as standard input/output operations.

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

FILE *fp;
int fd;
char buf[10];
int c;

int main(void)
{
   int n;

               /* deal with the file descriptor first */
   if((fd = open("fname",2)) < 0)
     {
        perror("open");
        exit(1);
     }

   if((n = read(fd,buf,10)) > 0)
      write(1,buf,n);

               /* link file pointer with file descriptor */
   fp = fdopen(fd,"w");
   while((c = getchar()) != EOF)
        putc(c,fp);
   fclose(fp);
   return 0;
 }
```

See also creat, creat64, fclose, fseek, fseek64, fopen, fopen64, freopen, freopen64, open, open64

feof - Test for end of file

Definition #include <stdio.h>

int feof(FILE *fp);

`feof` detects the end of the file pointed to by file pointer *fp*.

Return val.

≠ 0	End of file has been reached.
0	Otherwise.

Notes `feof` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

`feof` is normally used following access functions that do not report end of file (`fread`).

If the file has been repositioned (e.g. with `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind`) after EOF has been reached, or if the `clearerr` function has been called, `feof` returns a value of 0.

Record I/O `feof` can also be used unchanged on files with record I/O.

See also clearerr, ferror, fopen, fopen64, fseek, fseek64, fsetpos, fsetpos64

ferror - Test for file error

Definition #include <stdio.h>

int ferror(FILE *fp);

`ferror` checks whether the error flag is set in the FILE structure to which *fp* points.

Return val.

$\neq 0$	An error flag is set.
0	No error flag is set.

Notes `ferror` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

The error flag remains set until the associated file pointer is released (e.g. by `fclose` or program termination) or until the `clearerr` function is called.

You should always use `ferror` when you want to read from a file or write to it.

Record I/O `ferror` can also be used unchanged on files with record I/O.

Example The following program fragment checks before each fread call whether an error has been indicated for the FILE structure pointed to by *fp*.

```
FILE *fp;
char buf[10];
char x[5];

while( !ferror(fp))
     fread(buf,sizeof(x),10,fp);
```

See also clearerr, feof, fopen, fopen64

fflush - Flush file buffers

Definition

```
#include <stdio.h>

int fflush(FILE *fp);
```

`fflush` clears the buffer for the file pointed to by file pointer *fp* and writes the data that was temporarily stored in the buffer to this file. If *fp* is a NULL pointer, `fflush` performs these activities for all open files.

Return val.

0 `fflush` has flushed the buffer, or no buffer needs to be flushed because:
- the buffer does not yet exist (a write function has not yet been executed for the file) or
- the file is an input or INCORE file.

EOF `fflush` has not flushed the buffer because:
- the pointer *fp* is not assigned to a file (e.g. because the file is already closed) or
- the buffered data could not be transferred.

Notes

All standard I/O functions that write data to a file (`printf`, `putc`, `fwrite` etc.) store this data temporarily in an internal C buffer and only write it to the file when one of the following events occurs (See also section “Buffering” on page 63. Buffering does not take place in the case of outputs to strings (`sprintf`) and to INCORE files.):

- a newline character (\n) is detected (only for text files)
- the maximum record length of a disk file is reached
- for data display terminals: output to the terminal is followed by input from the terminal
- the `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind` or `fflush` functions 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 internal C buffer, the data of all ISAM files still stored in buffers is automatically written out to the 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 lengthens the record in the file. When an ISAM file is read, therefore, only those newline characters explicitly written by the program are read in.

Internally, `fflush` is executed automatically when a file is closed (`fclose`, `close`) or when a program ends normally or is terminated by means of `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 (cf. example).

Record I/O A call to the `fflush` function is not rejected with an error, but it has no effect. No data is buffered in the case of files with record I/O.

Example The following program reads alphabetically sorted names from `stdin` and outputs them to a file. Names that begin with the same letter are to be written in the same record of the file, separated from each other by a space. For “ANSI” functionality, the desired result is achieved only if output is to a SAM file. When output is to ISAM files, all names are written to one record, since `fflush` does not cause a change of record.

```
#include<stdio.h>

int main(void)
{
  FILE *fp;
  char name[20];
  char prevname;
  prevname = '%';
  fp = fopen ("link=link", "w");
  while (gets(name))
  {
    if(prevname != name[0])
       fflush(fp);
    else
       fputc(' ', fp);
    fputs(name, fp);
    prevname = name[0];
  };
  fclose(fp);
  return 0;
}
```

See also exit, close, fclose

fgetc - Read a character from a file

Definition

#include <stdio.h>

int fgetc(FILE *fp);

`fgetc` reads a character from the file indicated by file pointer *fp* from the current read/write position.

Return val.

integer	If successful, the character as a positive integer value.
EOF	for end of file or error.

Notes

`fgetc` behaves like `getc` (as a function). If you use a comparison such as

```
while((c = fgetc(fp)) != EOF)
```

in your program, the variable *c* must always be declared as an integer. If you define *c* as a `char`, the EOF condition is never satisfied for the following reason: -1 is converted to `char` ‘`0xFF`’ (i.e. +255); EOF, however, is defined as -1.

If `fgetc` is reading from the standard input `stdin`, and EOF is the end criterion for reading, you can satisfy the EOF condition by means of the following actions at the terminal: pressing the K2 key and entering the system commands EOF and RESUME-PROGRAM.

Example

The following program successively reads one character at a time from a maximum of 10 files passed in the call and outputs the character on the standard output.

```
#include <stdio.h>

FILE *fp[10], **app;

int main(int argc, char *argv[])
{
  int c, i;
  for (i = 1; i < argc && i <= 10; i++)
       fp[i-1] = fopen(argv[i], "r");
  app = fp;
  while(*app != NULL)
       {
         c = fgetc(*app++);
         putchar(c);
       }
  putchar('\n');
  return 0;
}
```

See also

getc, getchar, ungetc, fopen, fopen64

fgetpos, fgetpos64 - Determine current position of the read/write pointer

Definition
#include <stdio.h>

int fgetpos(FILE *fp, fpos_t *pos);
int fgetpos64 (FILE *fp, fpos64_t *pos);

`fgetpos` and `fgetpos64` return the current position of the read/write pointer for the file with the file pointer *fp* in the area to which *pos* points. The information stored in *pos* can be used to position the file with the `fsetpos` or `fsetpos64` function, insofar as **pos* is passed to it as an argument.

There is no functional difference between `fgetpos` and `fgetpos64`, except that `fgetpos64` uses the `fpos64_t` data type.

To process files > 2 GB, proceed as follows:

- If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `fgetpos`. `fgetpos64` is then used implicitly with the appropriate parameters.
- Otherwise, you have to call `fgetpos64`.

Return val.

0	On successful execution of `fgetpos` or `fgetpos64`.
≠ 0	In the event of an error. In addition, `errno` is set to EBADF.

Notes
`fgetpos`/`fgetpos64` can be used on binary files (SAM in binary mode, PAM, INCORE) and text files (SAM in text mode, ISAM).
`fgetpos`/`fgetpos64` cannot be used on system files (SYSDTA, SYSLST, SYSOUT).

For ISAM files the function pair `fgetpos`/`fsetpos` or `fgetpos64`/`fsetpos64` is considerably more efficient than the comparable function pair `ftell`/`fseek` or `ftell64`/`fseek64`.

Record I/O
`fgetpos` and `fgetpos64` return 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` and `fgetpos64` always return 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
fsetpos, fsetpos64, fseek, fseek64, ftell, ftell64

fgets - Read in a string from a file

Definition #include <stdio.h>

char *fgets(char *s, int n, FILE *fp);

`fgets` reads at most *n*-1 characters from the file with file pointer *fp*, stopping at the next newline (and including it) or at the end of the file. The read characters are entered by `fgets` into the area to which *s* points.

Return val. Pointer to the result string
if successful. `fgets` terminates the string with the null byte (\0).

NULL pointer if `fgets` has read nothing, e.g. because end of file was reached immediately or an error occurred when reading.

Notes You must explicitly provide the area in which `fgets` is to store the string read!

In contrast to `gets`, `fgets` also enters a newline character (if read) into the result string.

Example See the example of `fputs`

See also gets, fopen, fopen64, puts, fputs

fgetwc - Read a wide character from input stream

Definition

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

wint_t fgetwc(FILE *fp);
```

`fgetwc` reads the next character from the file with the file pointer *fp*, converts it to the corresponding wide character code, and advances the associated file position indicator for the file (if defined).

If an error occurs, the resulting value of the file position indicator is undefined.

If `fgetwc` is reading from the standard input `stdin`, and WEOF is the end criterion for reading, you can satisfy the WEOF condition by means of the following actions at the terminal: pressing the K2 key and entering the system commands EOF and RESUME-PROGRAM.

Return val.

Value of the read wide character as a `wint_t` value
if successful.

WEOF — if end-of-file is reached. The end-of-file indicator for the file is set;
or
if a read error occurs. The error indicator for the file is set, and `errno` is set to EBADF if *fp* is an invalid file pointer.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

The `ferror` or `feof` functions must be used to distinguish between an error condition and an end-of-file condition.

See also

feof, ferror, fgetc, fopen, fopen64

fgetws - Read a wide character string from a file

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

wchar_t * fgetws(wchar_t *ws, int n, FILE *fp);

`fgetws` reads characters from the file pointed to by *fp*, converts them 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, 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 file is indeterminate

Return val.
Pointer to the resulting wide character string *ws*
if successful.

NULL pointer
if end-of-file is reached. The end-of-file indicator for the file is set;
or
if a read error occurs. The error indicator for the file is set, and `errno` is set to EBADF if *fp* is an invalid file pointer.

Note
This version of the C runtime system only supports one-byte characters as wide character codes.

See also
fgetwc, fopen, fopen64, fread

_ _FILE_ _ - Output a source file name

Definition _ _FILE_ _

This macro generates the file name of the source program as a string in the form:

```
"name\0"
```

Note This macro does not need to be defined in an include file. Its name is recognized and replaced by the compiler.

float2ieee - Convert floating-point number from /390 format to IEEE format

Definition #include <ieee_390.h>

extern float float2ieee (float num);

`float2ieee` converts a 4-byte floating-point number in /390 format to IEEE format and returns it as the result. There is no loss of precision.

Parameters float num

4-byte floating-point number in /390 format

Return val. 4-byte floating-point number in IEEE format (in the event of success)

`+/- Infinity`, if the /390 floating-point number is greater than the largest IEEE floating-point number that can be represented.

`0.0`, if the /390 floating-point number is smaller than the smallest IEEE floating-point number that can be represented.

The global variable *float_exceptions_flag* contains information for the event of unsuccessful conversion and is defined as follows:

```
extern int float_exception_flags;
enum {
   float_flag_inexact   =  1,
   float_flag_divbyzero =  2,
   float_flag_underflow =  4,
   float_flag_overflow  =  8,
   float_flag_invalid   = 16
};
```

If the /390 floating-point number is greater than the largest IEEE floating-point number that can be represented, *float_flag_overflow* is set.

If the /390 floating-point number is smaller than the smallest IEEE floating-point number that can be represented, *float_flag_underflow* is set.

See also ieee2float, double2ieee, ieee2double

flocate - Explicitly position an ISAM file (record I/O)

Definition

```
#include <stdio.h>

int flocate(FILE *fp, void *key, size_t keylen, int option);
```

`flocate` explicitly positions an ISAM file with record I/O. `flocate` changes the current position of the read/write pointer of the file with file pointer *fp* according to the following:
key value *key*,
key length *keylen* and
option *option* (_KEY_FIRST, _KEY_LAST, _KEY_EQ, _KEY_GE).

Parameters

FILE *fp
File pointer of an ISAM file opened in the mode "type=record,forg=key" (cf. `fopen`/`open64`, `freopen`/`freopen64`).

void *key
Pointer to an area containing the key value.

size_t keylen
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 out 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
This parameter may contain the following values defined in <stdio.h>:

_KEY_FIRST	Positions the read/write pointer to beginning of file. The *key* and *keylen* parameters are ignored. Positioning works even if the file is empty.
_KEY_LAST	Positions the read/write pointer to end of file. The *key* and *keylen* parameters are ignored. Positioning works even if the file is empty.
_KEY_EQ	Positions the read/write pointer on the first record with the specified key *key*.
_KEY_GE	Positions the read/write pointer on the first record with a key value greater than or equal to the specified key *key*.

Return val.	0	If the record with the specified key exists.
	> 0	If the record does not exist.
	EOF	If an error has occurred.

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 read/write pointer 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.
With `flocate` it is only possible to position on the first record or after the last record of such a group.

See also

fdelrec, fgetpos, fgetpos64, fsetpos, fsetpos64, fopen, open64, freopen, freopen64

floor - Round down

Definition #include <math.h>

double floor(double x);

`floor` rounds down the floating-point number *x* to an integer.

Return val. Highest integer of the type `double` which is greater than or equal to *x* if successful.

HUGE_VAL in the event of an overflow, `errno` is also set to ERANGE (result too high).

Example

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

int main(void)
{
  double x;
  printf("Please enter the floating-point number to be rounded\n");
  if (scanf("%lf", &x) == 1)
    printf("The number %g is rounded down to %f\n", x, floor(x));
  return 0;
}
```

See also ceil

fmod - Remainder of a division

Definition

```
#include <math.h>

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

`fmod` calculates 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* as a floating-point number of type `double`	if successful.
0	if *y* = 0.

fopen, fopen64 - Open a file

Definition #include <stdio.h>

FILE *fopen(const char *f_name, const char *mode);
FILE *fopen64(const char *f_name, const char *mode);

`fopen` and `fopen64` open the file *f_name* and assign it a FILE structure and a file pointer. The file pointer points to the FILE structure assigned.
The FILE structure is defined in the file <stdio.h>. It contains the required data for most of the functions in the standard I/O library.

There is no functional difference between `fopen` and `fopen64`, except that a large file identifier is stored in the file description that is linked to the file descriptor, i.e. the O_LARGEFILE bit is set. A file descriptor is returned that can be used to extend the file over 2 GB.

To process files > 2 GB, proceed as follows:

– If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `fopen`. `fopen64` is then used implicitly with the appropriate parameters.

– Otherwise, you have to call `fopen64`.

Parameters const char *f_name

String specifying the file to be opened. *f_name* can be:

– any valid BS2000 file name.

– "link=*linkname*"
linkname denotes 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.

const char *mode

String specifying the desired access mode and optionally with an additional specification (tabexp=yes/no) to control how the tab character (\t) is handled.

Access modes:

"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, the old contents are preserved and the new data is appended to the end of the file. An existing file is positioned differently depending on whether KR or ANSI functionality is being used: with KR functionality (applies to C/C++ versions prior to V3.0 only) to the end of the file, with ANSI functionality to the start 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, the old contents are preserved and the new data is appended to the end of the file. An existing file is positioned differently depending on whether KR or ANSI functionality is being used: with KR functionality (applies to C/C++ versions prior to V3.0 only) to the end of the file, with ANSI functionality to the start of the file. If the file does not exist, it is created.

Tab character (\t)

In the *mode* parameter an optional entry controlling how the tab character (\t) is to be handled can be specified in addition to the access 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 with KR functionality (applies to C/C++ versions prior to V3.0 only).

"...,tabexp=no"

The tab character is not expanded.
This is the default setting for ANSI functionality.

Return val. Pointer to the assigned FILE structure
if successful.

NULL pointer if the file could not be opened, e.g. due to the absence of access permission, entry of an incorrect file name or link name etc.

Notes The BS2000 file name or link name may be written in lowercase and uppercase letters. It is automatically converted to uppercase letters.

The inclusion of a "b" as the second or third 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 can be opened as binary files without explicit specification of the binary mode.

When a non-existent file is created it is assigned the following file attributes by default:

	Binary file	Text file
Access method	SAM	SAM (KR functionality, applies to C/C++ versions prior to V3.0 only) ISAM (ANSI functionality)
Record format	F	V

If a link name is used the following file attributes can be changed with the ADD-FILE-LINK command: access method, record length, record format, block length and block format. See also section “Cataloged disk files (SAM, ISAM, PAM)” on page 73.

Whenever the old contents of an already existing file are deleted (opened for writing or for writing and reading) the catalog attributes of this 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`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind`) or a `fflush` call. This also applies to an output that follows an input.

Position of the read/write pointer in append mode:
If you explicitly position the read/write pointer away from the end of a file that was opened in append mode (`rewind`, `fsetpos`/`fsetpos64`, `fseek`/`fseek64`), the way it is handled depends on whether you are using KR or ANSI functionality.
KR functionality (applies to C/C++ versions prior to V3.0 only): The current read/write pointer is ignored only when writing with the elementary function `write` and automatically positioned to the end of the file.
ANSI functionality: The current read/write pointer is ignored for all write functions and automatically positioned to the end of the file.

An attempt to open a non-existent file for reading ends with an error.

(INCORE) files can only be opened for writing ("w"), for writing and reading ("w+r") or for reading ("r"). Data must first be written. To be able to read in the written data, the following options are among those available:
If the file was opened only for writing, it can be opened for reading with the function `freopen` or `freopen64`. If it was opened for writing and reading, the read/write pointer can be set to the beginning of the file with `rewind`.

You may open a file for different access modes simultaneously, provided these modes are compatible with one another within the BS2000 data management system.

When a program begins, three file pointers - for standard input, standard output, and standard error output - are assigned to it automatically. The pointers are named as follows:

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>.

Record I/O 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 indexed-sequentially. Indexed-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 ADD-FILE-LINK command. Sequential organization is assumed for SAM and PAM files, and indexed-sequential organization for ISAM files.

If `forg` is omitted and the FCB type is not defined (file does not exist, no ADD-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 return value is supplied:

a) The file must be opened in binary mode ("b" specified in the access mode).

b) "type=record" is permissible for SAM, PAM and ISAM files.

c) "forg=seq" is permissible for SAM and PAM files, "forg=key" for ISAM files.

d) The append mode (’a’) is not allowed with ISAM files. The position is determined by the key.

Example

```
/* program for copying from
                  file1 and file2 to file3 */

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

FILE *fp_1, *fp_2;
void copy(void);

int main(void)   /* file1 and file2 must exist */
{
   if((fp_1 = fopen("file1","r")) == NULL ||•• (fp_2 = fopen("file3","w"))
                    ==NULL)

      {
               /* program aborts, with return value 1 */
        perror("fopen");
        exit(1);
      }

   copy();
               /* reassign file pointer from file1 to file2 */

   if((freopen("file2","r",fp_1)) == NULL)

               /* program aborts, with return value 2 */
     exit(2);

   copy();
   fclose(fp_1);
   fclose(fp_2);
   return 0;
}

void copy(void)
{
   int c;
   while((c = getc(fp_1)) != EOF)
        putc((char)c,fp_2);
}
```

See also creat, creat64, fdopen, freopen, freopen64, ferror, open, open64, fclose, fseek, fseek64

fprintf - Formatted output to a file

Definition
#include <stdio.h>

int fprintf(FILE *fp, const char *format, argumentlist);

`fprintf` edits data (characters, strings, numeric values) according to the specifications in the *format* string and writes this data to the file with file pointer *fp*.

`fprintf` works like `printf`, except that the edited data is written to a file and not to the standard output.

Parameters
FILE *fp
File pointer to the output file.

const char *format
Format string as described under printf with KR or ANSI functionality (see `printf`).

argumentlist
Variables or constants whose values are to be converted and formatted for output according to the information in the format statements.
If the number of format statements does not match the number of arguments the following applies:
If there are more arguments, the surplus arguments are ignored.
If there are fewer arguments, the results are undefined.

Return val.
number of characters output
if successful.

Negative value if an error occurs.

Notes
`fprintf` rounds to the specified precision when converting floating-point numbers.

`fprintf` does not convert one data type to another. A value must be explicitly converted (e.g. with the `cast` operator) if it is not to be output to conformity with its type.

The characters are not written immediately to the external file but are stored in an internal C buffer (see section “Buffering” on page 63).

Maximum number of characters to be output

With KR functionality (applies to C/C++ versions prior to V3.0 only) a maximum of 1400 characters can be output per `fprintf` call,
with ANSI functionality a maximum of 1400 characters per conversion element (e.g. %s).

Attempts to output non-initialized variables or to output variables in a manner inconsistent with their data type can lead to undefined results.

The behavior is undefined if the percent sign (%) in a format statement is followed by an undefined formatting or conversion character.

Example

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

int main(void)
{
    FILE *fp;
    char c, name[40];
    int i;
    char *string;
    double d;

    printf("Name of the output file: \n");
    gets(name);
    if((fp = fopen(name,"w")) == NULL)
    {
       printf("Can't open %s\n", name);
       exit(1);
    }
    c = 'A';
    i = 999;
    string = "This is a string.";
    d = 123.456;
    fprintf(fp, "%c %d %s %f\n", c, i, string, d);
    fclose(fp);
    puts("Correct output to file:A 999 This is a string. 123.456000");
    return 0;
}
```

See also printf, sprintf, snprintf, putc, putchar, puts, scanf, fscanf

fputc - Write a character to a file

Definition #include <stdio.h>

int fputc(int c, FILE *fp);

`fputc` writes the character *c* to a file (with file pointer *fp*) at the current read/write position.

Return val. Written character *c* as a positive integer value if successful.

EOF otherwise.

Notes The characters are not written immediately to the external file but are stored in an internal C buffer (see section "Buffering" on page 63).

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" on page 65).

Example The following program reads characters from SYSDTA and outputs them to SYSOUT.

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

void copy(void);
FILE *fp_in, *fp_out;
int main(void)
{
   fp_in = fopen("(SYSDTA)","r");
   fp_out = fopen ("(SYSOUT)","w");

   copy();
   fclose(fp_in);
   fclose(fp_out);
   return 0;
}
void copy(void)
{
  int c;
  while((c = fgetc(fp_in)) != EOF)
    fputc((char)c,fp_out);
}
```

See also fopen, fopen64, fputwc, putc, putchar

fputs - Write a string to a file

Definition

#include <stdio.h>

int fputs(const char *s, FILE *fp);

`fputs` writes the string *s* to the file with file pointer *fp*. *s* must be terminated with a null byte (\0).

Return val.

0	if successful.
EOF	otherwise.

Notes

In contrast to `puts`, `fputs` does not end its output with the addition of a newline character.

The terminating null byte of *s* is not output.

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” on page 65).

Example

The following program reads strings from *file* and then outputs them at the display terminal (SYSOUT).

```
#include <stdio.h>

int main(void)
{
   FILE *fp_in, *fp_out;
   char s[BUFSIZ];
   int max = 120;

   fp_in = fopen("file","r");
   fp_out = fopen("(SYSOUT)","w");

   while(fgets(s, max, fp_in) != NULL)
     fputs(s, fp_out);
   return 0;
}
```

See also

fopen, fopen64, puts, fgets

fputwc - Write a wide character to a file

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

wint_t fputwc(wchar_t wc, FILE *fp);

`fputwc` writes the wide character specified by *wc* to the output file pointed to by the file pointer *fp* at the position indicated by the associated file position indicator for the file (if defined), and advances the file position indicator appropriately.
If the file cannot support positioning requests or was opened in append mode, the character is appended to the file.
If an error occurs during the write operation, the “insert” mode of the output file is indeterminate.

Return val. The written wide character *wc* as a `wint_t` value
if successful.

WEOF if an error occurs. The error indicator for the file is set. If *fp* is not a valid file pointer, `errno` is set to EBADF.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

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” on page 65).

See also ferror, fopen, fopen64, setbuf

fputws - Write a wide character string to a file

Definition

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

int fputws(const wchar_t *ws, FILE *fp);
```

`fputws` writes a character string corresponding to the (null-terminated) wide character string pointed to by *ws* to the file pointed to by the file pointer *fp*. No character corresponding to the terminating null wide-character code is written.

Return val.

Non-negative number	upon successful completion.
-1	otherwise.

Notes

`fputws` does not end the output with a newline character.

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” on page 65).

This version of the C runtime system only supports one-byte characters as wide character codes.

See also fopen, fopen64, fputs, fputwc

fread - Read blockwise from a file

Definition

```
#include <stdio.h>

size_t fread(void *p, size_t elsize, size_t n, FILE *fp);
```

`fread` reads *n* elements, each requiring *elsize* bytes, from the file with file pointer *fp* and stores the data elements read in the area to whose beginning *p* points. Following a successful read, the read/write pointer is located after the last byte read.

Return val. Number of elements actually read, if successful.
This number may be less than *n* if an error occurs or end of file is reached.

Notes You must see to it that the area to which *p* points is sufficient for storing the data elements read.

To ensure that *elsize* specifies the correct number of bytes for a data element, you should use the `sizeof` function for the size of the data unit to which *p* points.

`fread` does not distinguish between end of file and error. Therefore, the `feof` and `ferror` functions should be used before or after each `fread` call to check whether a correct read access is possible.

`fread` reads beyond the newline (\n) character and is therefore specially suitable for reading in binary files.

Record I/O `fread` reads a record (or block) from the current file position.

Number of characters to be read in: *n* is taken to be the total number of characters to be read in, i.e.

n = element length * number of elements

If *n* is greater than the current record length, then only this record is read nevertheless.

If *n* is less than the current record length, only the first *n* characters of the record are read. On the next read access the data of the next record is read.

`fread` supplies the same return value as for stream I/O, namely the number of elements read in their entirety. For record I/O it is best to use only element length 1 since in this case the return value corresponds to the length of the record read (without any record length field).

Example The following program transfers two personal data items to a file (`fwrite`) and then reads in this data again (`fread`).

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  size_t result;
  static struct p
  {
    char name[20];
    int a;
  } person[2] =
    {
      ≥"ANNE", 30¸,
      Ã"JOHN", 60Õ,
    };

  fp = fopen("link=link", "w+r");

  result = fwrite(person, sizeof(struct p), 2, fp);
  printf("%d Personal data written\n", result);

  rewind(fp);
  result = fread(person, sizeof(struct p), 2, fp);
  printf("%d Personal data read\n", result);
  printf("Name1: %s, Age1: %d\n", person[0].name, person[0].a);
  printf("Name2: %s, Age2: %d\n", person[1].name, person[1].a);
  return 0;
}
```

See also fwrite, feof, ferror, read, fopen, fopen64, fgetc, fgets, fscanf

free - Free memory space

Definition

#include <stdlib.h>

void free(void *p);

`free` releases the memory space pointed to by *p*. *p* must be the the result of a previous `malloc`, `calloc`, or `realloc` call. Otherwise, the result is undefined!

`free` is part of a C-specific memory management package with its own free memory management facility. Memory released with `free` is not returned to the operating system but is taken by the `free` memory management facility (cf. `garbcoll` function).

Example

The following program fragment deallocates memory space that was previously reserved with `malloc`.

```
#include <stdlib.h>

char *buf;
buf = (char *)malloc(100);
     .
     .
free(buf);
```

See also

malloc, calloc, realloc, garbcoll

freopen, freopen64 - Reassign file pointer

Definition #include <stdio.h>

FILE *freopen(const char *f_name, const char *mode, FILE *fp);
FILE *freopen64(const char *f_name, const char *mode, FILE *fp);

`freopen` and `freopen64` are used to reassign an already defined file pointer to a new file. `freopen` and `freopen64` close the file with file pointer *fp*, open the file *f_name*, and assign to it the FILE structure with file pointer *fp*.

There is no functional difference between `freopen` and `freopen64`, except that a large file identifier is stored in the file description that is linked to the file descriptor, i.e. the O_LARGEFILE bit is set. A file descriptor is returned that can be used to extend the file over 2 GB.

To process files > 2 GB, proceed as follows:

– If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `freopen`. `freopen64` is then used implicitly with the appropriate parameters.

– Otherwise, you have to call `freopen64`.

Parameters const char *f_name
String specifying the new file to be opened. *f_name* can be:

– any valid BS2000 file name

– "link=*linkname*".
linkname identifies a BS2000 link name

– "(SYSDTA)", "(SYSOUT)", "(SYSLST)"
the appropriate system file

– "(SYSTERM)"
terminal I/O

– "(INCORE)"
temporary binary file that is only created in virtual memory.

const char *mode String specifying the desired access mode and optionally with an additional specification (tabexp=yes/no) to control how the tab character (\t) is handled.

Access modes:

"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, the old contents are preserved and the new data is appended to the end of the file. An existing file is positioned differently depending on whether KR or ANSI functionality is being used: with KR functionality (applies to C/C++ versions prior to V3.0 only) to the end of the file, with ANSI functionality to the start 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, the old contents are preserved and the new data is appended to the end of the file. An existing file is positioned differently depending on whether KR or ANSI functionality is being used: with KR functionality (applies to C/C++ versions prior to V3.0 only) to the end of the file, with ANSI functionality to the start of the file. If the file does not exist, it is created.

Tab character (\t)

In the *mode* parameter an optional entry controlling how the tab character (\t) is to be handled can be specified in addition to the access 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 with KR functionality (applies to C/C++ versions prior to V3.0 only).

"...,tabexp=no"

The tab character is not expanded.
This is the default setting with ANSI functionality.

FILE *fp
File pointer to be reassigned.

Return val. Pointer to the original file pointer *fp*
If successful.

NULL pointer if the file could not be opened, e.g. due to the absence of access permission, entry of an incorrect file name or link name etc.

Notes The BS2000 file name or link name may be written in lowercase and uppercase letters. It is automatically converted to uppercase letters.

The file to which the file pointer *fp* was originally assigned is closed even if the new file could not be opened.

Specifying a "b" as the second or third 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 can be opened as binary files without explicitly specifying the binary mode.

If a new file is created it is given the following attributes by default:

	Binary file	Text file
Access method	SAM	SAM (KR functionality, (applies to C/C++ versions prior to V3.0 only) ISAM (ANSI functionality)
Record format	F	V

By using a link name the following file attributes can be changed with the ADD-FILE-LINK command: access method, record length, record format, block length and block format. See also section “System files (SYSDTA, SYSOUT, SYSLST)” on page 70.

Whenever the old contents of an already existing file are deleted (opened for writing or for writing and reading) the catalog attributes of this 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`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind`) or a `fflush` call. This also applies to an output that follows an input.

Position of the read/write pointer in append mode:
If you explicitly position the read/write pointer away from the end of a file that was opened in append mode (`rewind`, `fsetpos`/`fsetpos64`, `fseek`/`fseek64`), the way it is handled depends on whether you are using KR or ANSI functionality.
KR functionality (applies to C/C++ versions prior to V3.0 only): The current read/write pointer is ignored only when writing with the elementary function write and automatically positioned to the end of the file.
ANSI functionality: The current read/write pointer is ignored for all write functions and automatically positioned to the end of the file.

An attempt to open a non-existent file for reading ends with an error.

(INCORE) files can only be opened for writing ("w"), for writing and reading ("w+r") or for reading ("r"). Data must first be written. To be able to read in the written data, the following options are among those available:
If the file was opened only for writing, it can be opened for reading with the function `freopen` or `freopen64`. If it was opened for writing and reading, the read/write pointer can be set to the beginning of the file with `rewind`.

You may open a file for different access modes simultaneously, provided these modes are compatible with one another within the BS2000 data management system.

When a program begins, three file pointers - for standard input, standard output, and standard error output - are assigned to it automatically. The pointers are named as follows:

stdin	file pointer for standard input (terminal)
stdout	file pointer for standard output (terminal)
stderr	file pointer for standard error output (terminal)

`freopen` and `freopen64` are often used to change these standard assignments, i.e. to reassign the pointers to other files. Using it in this way corresponds to the redirection mechanism of the UNIX shell (PARAMETER-PROMPTING in the RUNTIME option) or to the appropriate ASSIGN commands in BS2000 (see also example).

A maximum of _NFILE files may be open simultaneously. _NFILE is defined as 2048 in <stdio.h>.

Record I/O

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 indexed-sequentially. Indexed-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 ADD-FILE-LINK command. Sequential organization is assumed for SAM and PAM files, and indexed-sequential organization for ISAM files.

If `forg` is omitted and the FCB type is not defined (file does not exist, no ADD-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 return value is supplied:

a) The file must be opened in binary mode ("b" specified in the access mode).

b) "type=record" is permissible for SAM, PAM and ISAM files.

c) "forg=seq" is permissible for SAM and PAM files, "forg=key" for ISAM files.

d) With "forg=key" the append mode 'a' is invalid. For ISAM files the position is determined by the key in the record.

Example The following program fragment makes the file *out* the standard output file.

```
FILE *fp;

fp = freopen("out","w",stdout)
```

Following this assignment, *fp* and `stdout` are both file pointers for the file *out*.

See also fopen, fopen64, fdopen

frexp - Split floating-point number into mantissa and exponent

Definition

#include <math.h>

double frexp(double value, int *e_p);

`frexp` splits a floating-point number *value* into the mantissa *x* and the exponent *n* on the basis of the formula:

$value = x * 2^n$

$\|x\|$	is in the interval [0.5, 1.0[
n	is an integer

The result from `frexp` is the mantissa *x* and an integer value for the exponent *n*. The exponent is returned indirectly via a result parameter *e_p*.

`frexp` is the inverse function of `ldexp`.

Return val.

Mantissa *x*	a floating-point number of type `double` which satisfies the equation $value = x * 2^n$ and lies in the interval [0.5, 1.0[.
0	if *value* is equal to 0 (the exponent is also equal to 0 in this case).

Note

Note that the argument *e_p* must be a pointer!

Example

Normalized representation of the number 5 to base 2:

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

int main(void)
{
   double z;
   int exp;

   z = frexp((double)5, &exp);
   printf("5 = %g * 2 ** %d\n", z, exp);
   return 0;
}
```

See also

ldexp, modf

fscanf - Formatted input from a file

Definition

#include <stdio.h>

int fscanf(FILE *fp, const char *format, argumentlist);

`fscanf` reads data (input fields) from a file with file pointer *fp*, converts this data according to the specifications in the format string *format*, and stores the results in the areas that you specify with the result pointers in the argument list.

`fscanf` works like `scanf`, except that the input fields are read from a file rather than the standard input (`stdin`).

Parameters

FILE *fp
File pointer to the input file.

const char *format
Format string as described under `scanf` with KR or ANSI functionality (see relevant section in `scanf` description)

argumentlist
Pointers to variables in which `fscanf` is to store the converted result.
No pointer arguments may be specified for %* statements (skip assignment) in *format*.
There must be one pointer argument each for all other % statements. The data type of the pointer argument is determined by the type specification of the associated format statement.

Return val.

Number of input fields read and successfully converted.
This does not include input fields for which %* (skip assignment) was specified.

EOF	if an error occurred before the start of the conversions.

Note

You will find detailed information, notes, and examples on formatted input under `scanf`.

See also

scanf, sscanf

fseek, fseeko, fseek64, fseeko64 - Position read/write pointer

Definition #include <stdio.h>

int fseek(FILE *fp, long offset, int loc);
int fseeko(FILE *fp, off_t offset; int loc);
int fseek64(FILE *fp, long long offset, int loc);
int fseeko64(FILE *fp, off64_t offset, int loc);

`fseek` and `fseek64` position the read/write pointer for the file with file pointer *fp* in accordance with the specifications in *offset* and *loc*. It thus becomes possible for you to process a file 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`/`ftello` or `ftell64`/`ftello64`.

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).

To process files > 2 GB, proceed as follows:

– If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `fseeko`. `fseeko64` is then used implicitly with the appropriate parameters. (Automatic conversion is not supported for fseek.)

– Otherwise, you have to call `fseek64` or `fseeko64`.

There is no functional difference between `fseek` and `fseek64` or `fseeko` and `fseeko64`. The functions differ only in terms of the offset type used.

Parameters FILE *fp

File pointer for the file whose read/write pointer is to be positioned.

long offset / off_t offset / long long offset / off64_t offset

Since the meaning, combination options, and effect of these parameters differ for text and binary files, they are individually described in the following:

Text files (SAM in text mode, ISAM)

Possible parameter values:

offset	0L or value determined by a previous `ftell`/`ftello` call.
offset (64-bit interface)	0LL or value determined by a previous `ftell`/`ftello`/`ftell64`/`ftello64` call.
loc	SEEK_SET (beginning of file) SEEK_END (end of file)

Meaningful combinations and their effects:

offset	loc	Effect
`ftell`/`ftello` value or `ftell64`/`ftello64` value	SEEK_SET	Position to the location determined by `ftell`/`ftello` or `ftell64`/`ftello64`.
0L or 0LL	SEEK_SET	Position to the beginning of the file.
0L or 0LL	SEEK_END	Position to the end of the file.

Binary files (SAM in binary mode, PAM, INCORE)

Possible parameter values:

offset
: Number of bytes by which the current read/write pointer 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 position to the beginning or end of the file.

loc
: For absolute positioning to the beginning or end of the file, the position to which the read/write pointer is to be shifted.
  For relative positioning, the position from which the read/write pointer is to be shifted by *offset* bytes:

  SEEK_SET (beginning of file)
  SEEK_CUR (current position)
  SEEK_END (end of file)

Meaningful combinations and their effects:

offset	**loc**	**Effect**
0L or for 64 bit: 0LL	SEEK_SET	Position to the beginning of the file.
0L or for 64 bit: 0LL	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`/`ftello` value or `ftell64`/`ftello64` value	SEEK_SET	Position to the location marked by an `ftell`/`ftello` or `ftell64/ftello64` call.

Return val.

0 if successful.

-1 if an error occurred.
If you position past the end of a binary file opened only for reading, `errno` is set to EMDS.

Notes

The call `fseek(fp,0L,SEEK_SET)` or `fseek64(fp,0LL,SEEK_SET`) is equivalent to the call `rewind(fp)`.

If new records are written to a text file that was opened in the write or append mode and an `fseek/fseeko` or `fseek64/fseeko64` 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/fseeko` or `fseek64/fseeko64` does not cause a change of line (or change of record), i.e. the data is not automatically terminated with a newline character when writing from the buffer. Subsequent data lengthens the record in the file. When an ISAM file is read, therefore, only those newline characters explicitly written by the program are read in.

If you position past the end of binary file opened for writing, a “gap” appears between the last physically stored data and the newly written data. Reading from this “gap” returns binary zeros.
If you position past the end of binary file opened for reading only, an error occurs (EMDS).

It is not possible to position to system files (SYSDTA, SYSLST, SYSOUT).

A successful `fseek`/`fseeko` or `fseek64`/`fseeko64` call deletes the EOF flag of the file and cancels all the effects of the preceding `ungetc` calls for this file.

Record I/O `fseek`/`fseeko` and `fseek64`/`fseeko64` can be used only for positioning to the beginning or end of the file.

`fseek(fp,0L,SEEK_SET)` and `fseek64(fp,0LL,SEEK_SET)` position on the first record of the file.

`fseeko(fp,0L,SEEK_SET)` and `fseeko64(fp,0LL,SEEK_SET)` position on the first record of the file.

`fseek(fp,0L,SEEK_END)` and `fseek64(fp,0LL,SEEK_SET)` position after the last record of the file.

`fseeko(fp,0L,SEEK_END)` and `fseeko64(fp,0LL,SEEK_SET)` position after the last record of the file.

If called with any other arguments, `fseek`/`fseeko` and `fseek64`/`fseeko64` return EOF.

Example 1 The following program reads *file* from the eleventh character to the end of the file (only functions for binary files).

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  int c;

  if((fp = fopen("file","rb")) != NULL)
  {
               /* skip the first 10 characters */
    fseek(fp,10L,SEEK_SET);
    while((c=getc(fp)) != EOF)
          putc((char)c,stdout);
    fclose(fp);
  }
  return 0;
}
```

Example 2 The following program processes a file in the update mode. Lowercase letters are written back as uppercase letters; all other characters remain unchanged.

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

int main(void)
{
   FILE *fp;
   int c;
   long n;
   fp = fopen("link=link","r+w");
     do
     {
       n = ftell(fp);
       c = getc(fp);
       if (islower(c) == 0) continue; /* If character is not in lowercase, */
                                      /* read next character */

      else
       {                              /* If character is in lowercase, */
         fseek(fp, n, SEEK_SET);      /* position to this character and */
         fputc((toupper(c)), fp);     /* write it back in uppercase. */
       }
     }
     while(c != EOF);
     fclose(fp);
     return 0;
}
```

See also ftell, ftello, ftell64, ftello64, fsetpos, fsetpos64, lseek, lseek64, rewind, tell

fsetpos, fsetpos64 - Position read/write pointer

Definition

```
#include <stdio.h>

int fsetpos(FILE *fp, const fpos_t *pos);
int fsetpos64(FILE *fp, const fpos64_t *pos);
```

`fsetpos` and `fsetpos64` set the read/write pointer of the file with file pointer *fp* to a position *pos* previously determined by `fgetpos` or `fgetpos64`.
After positioning, the next operation can be a read or a write function.

To process files > 2 GB, proceed as follows:

– If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `freopen`. `freopen64` is then used implicitly with the appropriate parameters.

– Otherwise, you have to call `freopen64`.

There is no functional difference between `fsetpos` and `fsetpos64`, except that `fsetpos64` uses an `fpos64_t` type.

Return val.

0	On successful execution of `fsetpos`.
≠ 0	In the event of an error. In addition, `errno` is set to EBADF.

Notes

`fsetpos` and `fsetpos64` can be used on binary files (SAM in binary mode, PAM, INCORE) and text files (SAM in text mode, ISAM).
`fsetpos` and `fsetpos64` cannot be used on system files (SYSDTA, SYSLST, SYSOUT).

A successful `fsetpos` or `fsetpos64` call deletes the EOF flag of the file and cancels all the effects of preceding `ungetc` calls for this file.

If new records are written to a text file (opened for creation or in append mode) and a `fsetpos` or `fsetpos64` call is issued any residual data is first written from the buffer to the file and terminated with a newline character (\n).
Exception for ANSI functionality:
If the data of an ISAM file in the buffer does not end in a newline character, `fsetpos` or `fsetpos64` does not cause a change of line (or change of record), i.e. the data is not automatically terminated with a newline character when writing from the buffer. Subsequent data lengthens the record in the file. When an ISAM file is read, therefore, only those newline characters explicitly written by the program are read in.

For ISAM files the function pair `fgetpos`/`fsetpos` or `fgetpos64`/`fsetpos64` is considerably more efficient than the comparable function pair `ftell`/`fseek` or `ftell64`/`fseek64`.

Record I/O In ISAM files with key duplication, `fsetpos` and `fsetpos64` 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.
With `fsetpos` and `fsetpos64` it is only possible to position on the first record or after the last record of such a group.

See also fgetpos, fgetpos64, fseek, fseek64, ftell, ftell64

ftell, ftello, ftell64, ftello64 - Determine current position of read/write pointer

Definition

#include <stdio.h>

long ftell(FILE *fp);
off_t ftello(FILE *fp);
long long ftell64(FILE *fp);
off64_t ftello64(FILE *fp);

`ftell`/`ftello` and `ftell64`/`ftello64` return the current position of the read/write pointer for the file with file pointer *fp*.

`ftell`/`ftello` and `ftell64`/`ftello64` can be used on binary files (SAM in binary mode, PAM, INCORE) as well as text files (SAM in text mode, ISAM).

To process files > 2 GB, proceed as follows:

– If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `ftello`. `ftello64` is then used implicitly with the appropriate parameters.

– Otherwise, you have to call `ftell64` or `ftello64`.

There is no functional difference between `ftell and ftell64` or `ftello and ftello64`. The functions differ only in terms of the offset type used for the return value.

Return val. Position in the file if successful:

for binary files, the number of bytes that offsets the read/write pointer from the beginning of the file,
for text files, the absolute position of the read/write pointer.

-1 if an error occurs. If the value for the file position does not lie within the value range of the return type, `errno` is additionally set to ERANGE.

Notes

The functions `fseek`/`fseeko` and `fseek64`/`fseeko64` can be used to position on the position returned by `ftell`/`ftello` and `ftell64`/`ftello64`.

`ftell`/`ftello` and `ftell64/ftello64` cannot be used for system files (SYSDTA, SYSLST, SYSOUT).

Example In the following program, each character in *file* is output with the position of the read/write pointer, starting with the eleventh character (only functions with binary files).

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   int c;
   if((fp = fopen("file","rb")) != NULL)
     {
              /* the first 10 characters are skipped */
       fseek(fp,10L,SEEK_SET);
       while((c=getc(fp)) != EOF)
            printf("Position : %ld, character : %c\n",ftell(fp),c);
       fclose(fp);
     }
   return 0;
}
```

See also fseek, fseek64, fgetpos, fgetpos64, ftell, ftell64, tell

ftime, ftime64 - Current time

Definition

<sys.timeb.h> is included as: #include <sys.timeb.h>

int ftime(struct timeb *p);
int ftime64(struct timeb64 *p);

In the structure which points to *p*, `ftime` and `ftime64` supply the current time (local time) as the number of seconds and milliseconds which have passed since the reference date (epoch).

With `ftime` the reference date depends on the use of the TIMESHIFT bind option (see section "Time functions" on page 40):

– without TIMESHIFT bind option (default): 1/1/1950 00:00:00.
– with TIMESHIFT bind option: 1/1/1970 00:00:00.

With `ftime64` the reference date is always 1/1/1970 00:00:00

From 01/19/2018 03:14:08 (without TIMESHIFT bind option) or from 01/19/2038 03:14:08 (with TIMESHIFT bind option) `ftime` will issue the message `CCM0014` and terminates the program.

Irrespective of the use of the TIMESHIFT bind option, `ftime64` will supply correct results up to 3/18/4317 02:44:48.

For portability reasons additional options have been included in the `timeb` and `timeb64` structures. However, they are not supported in the BS2000 environment.

Return val. Always 0.

Notes As always in such cases, you must explicitly provide the memory space for the result structure!

See also time, time64, ctime, ctime64

fwide - Define orientation of a file

Definition

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

int fwide(FILE *fp, int mode);
```

`fwide` defines the orientation of the file pointed to by the file pointer *fp*, provided the file has no orientation as yet. If the orientation has already been defined, e.g., by an earlier I/O operation, `fwide` does not alter the orientation of the file.

Depending on the *mode* argument, `fwide` tries to set the orientation as follows:

mode > 0	The file is made wide oriented.
mode < 0	The file is made byte oriented.
mode = 0	The orientation of the file is not altered.

Return val.

> 0	if *fp* is wide oriented after the call to `fwide`.
< 0	if *fp* is byte oriented after the call to `fwide`.
0	if *fp* has no orientation.

Note

This version of the C runtime system only supports one-byte characters as wide character codes.

fwprintf, swprintf, vfwprintf, vswprintf, vwprintf, wprintf - Formatted output of wide characters

Definition

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

int fwprintf(FILE *fp, 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 *fp, const wchar_t *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

These functions are used for formatted output.

`fwprintf` edits the arguments listed in *arglist*, under control of the wide string pointed to by *format*, and writes output to the file pointed to by *fp*.
`fwprintf` returns when the end of *format* is reached.

`vwprintf` corresponds to the function `fwprintf` with *fp* = `stdout`, where the argument list is replaced by an argument of type `va_list`, which must have been initialized with the macro `va_start` (possibly followed by `va_arg` calls). The function does not call the `va_end` macro.

`wprintf` correspond to the function `fwprintf` with *fp* = `stdout`.

`swprintf` writes formatted output to the wide character string *s*. `swprintf` is otherwise equivalent to the `fwprintf` function. A maximum of *n* wide character codes are written, including the terminating null byte, which is automatically appended when *n* > 0.

`vfwprintf` corresponds to the function `fwprintf`, where the argument list is replaced by an argument of type `va_list`, which must have been initialized with the macro `va_start` (possibly followed by `va_arg` calls). The function does not call the `va_end` macro.

`vswprintf` corresponds to the function `swprintf`, where the argument list is replaced by an argument of type `va_list`, which must have been initialized with the macro `va_start` (possibly followed by `va_arg` calls). The function does not call the `va_end` macro.

Parameters *format* is a wide character string which contains none, one or more conversion 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 associated with a conversion specification are converted accordingly and written as formatted output to the output data stream.

– characters of type `wchar_t` (but not %), which are simply copied to the output stream (1: 1).

– white-space characters (see section “White space” on page 65)

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 integer (consisting of decimal digits) or an asterisk (*), which specifies a minimum **field width** for the output of an argument. 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 specifies 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 decimal-point 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 characters to be printed from a string in an *s* conversion. The precision takes the form of a period (.), followed by an integer consisting of decimal digits or an asterisk (*).
If only the period is specified, the precision is set to 0.

– An optional `h`, `l` or `L` before a conversion specifier:
`l` before `c`: means that the c conversion specifier applies to a `wint_t` argument;
`l` before `s`: means that the s conversion specifier applies to a pointer to a `wchar_t` argument (i.e. a pointer to a wide character string);
`h` before d, i, o, u, x, or X: means that the conversion specifier following `h` applies to a `short int` or `unsigned short int` argument (the argument is promoted according to the integral promotions, and its value is converted to `short int` or `unsigned short int` before printing);
`h` before `n`: means that the following `n` conversion specifier applies to a pointer to a `short int` argument;

`l` before `d`, `i`, `o`, `u`, `x` or `X`: means that the following conversion specifier applies to a `long int` or `unsigned long int` argument;
`l` before `n`: means that the following `n` conversion specifier applies to a pointer to a `long int`;
`ll` before `d`, `i`, `o`, `u`, `x` or `X`: means that the following conversion specifier applies to a `long long int` or `unsigned long long int` argument;
`ll` before `n`: means that the following conversion specifier applies to an argument of type pointer to `long long int`;
`L` before `e`, `E`, `f`, `g` or `G`: means that the following `e`, `E`, `f`, `g` or `G` conversion specifier applies to a `long double` argument.

If an `h` , `l` or `L` appears with any other conversion specifier, the behavior is undefined.

– A **conversion character** of type `wchar_t` that indicates the type of conversion to be applied, see the listing below.

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.

Format statements may be structured as follows:

```
                                                ⎧[{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. Formatting characters
3. Field width
4. Precision
5. Characters which specify the actual conversion

Formatting characters

–	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 wide character of a signed conversion is not a sign, or if a signed conversion results in no wide characters, a space is 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 “alternate form”. For o conversion, it increases the precision to force the first digit of the result to be zero. For x (or X) conversions, a non-zero result will have the character string "0x" (or "0X") prefixed to it. For e, E, f, g or G conversions, the result always contains a decimal-point wide character, even if no digits follow it. (Normally, a decimal-point wide 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 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 number in the form [–]*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 a zero value with an explicit precision of zero is no wide character.

`o, u` The `unsigned int` argument is converted to an unsigned octal number (o) or unsigned decimal number (u) in the form *dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1.
The result of converting a zero value with a precision of zero is no wide characters.

`x, X` The `unsigned int` argument is converted to a unsigned hexadecimal number in the form *dddd*; in addition to the numbers, the letters `abcdef` are used for x conversion and the letters ABCDEF for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with an explicit precision of zero is no wide character.

`f` The `double` argument is converted to decimal notation in the form `[–]`*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 form `[-]`*d.ddde+-dd*, where there is exactly 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 form `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. If an precision is 0, it is taken as 1. The form used depends on the value converted; form `e` (or `E`) is 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 decimal-point wide character appears only if it is followed by a digit.
`c`	If an `l` character is present, the argument of type `wint_t` is converted to type `wchar_t` and the resulting character is written. If no `l` character is present, the argument of type `int` is converted to a wide character as if by calling `btowc` and the resulting wide character is written.
`s`	If no `l` character is present, the argument must be a pointer to an array of `char` type. Characters from the array are converted as if by repeated calls to the `mbrtowc` function, with the conversion state described by an object of type `mbstate_t`, initialized to zero before the first multibyte character is converted, and written up to (but not including) the terminating null wide character. If an `l` character is present, the argument should be a pointer to an array of `wchar_t` type. Wide characters from the array are written up to (but not including) a terminating null wide character. If a precision *m* is specified, no more than *m* wide characters are written. If the precision is not specified or is greater than the size of the converted array, the converted array shall contain a null wide character (as end criterion).
`S`	Equivalent to `ls`.
`C`	Equivalent to `lc`.
`p`	The argument must be a pointer to `void`. The output is 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 have the form %%.

If the character that follows `%` is not a valid conversion character, the result of the conversion is undefined.

If any argument is a `UNION` or is a pointer to a `UNION`, the result of the conversion is undefined.
The same applies when the argument is an array or a pointer to an array, except in 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 pointer and uses `%p`.

In no case does a non-existent or a too 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 accomodate the conversion result.

Return val. Number of wide characters printed
if successful.

Negative value if an error occurs.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also btowc, fprintf, mbrtowc, printf

fwscanf, swscanf, wscanf - Read formatted input

Definition

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

int fwscanf(FILE *fp, 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]);

These functions are used for formatted input.
Each of these functions reads input, 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.

`fwscanf` reads formatted input from a file with the file pointer *fp*.

`swscanf` reads formatted input from the wide character string *s*. `swscanf` is otherwise equivalent to the `fwscanf` function. The end of the wide character string is equivalent to EOF.

`wscanf` reads formatted input from the standard input `stdin`. `wscanf` corresponds to the `fwscanf` function with *fp* = `stdin`.

Parameters

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 `wchar_t` (but no white-space character 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 each input character, but do not initially convert it or store it 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 due to an invalid wide character, the character involved remains in the input stream unread.

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 and 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 wide character asterisk (*).

– An optional non-zero decimal integer that specifies the maximum **field width**.

– An optional size modifier `h`,`l` (ell) or `L` indicating the size of the receiving object:
the conversion characters `c`, `s` and `[` preceded by `l` (ell): the corresponding argument is a pointer to `wchar_t`.
`h` or `l` preceded by `d`, `i` and `n` : the corresponding argument is a pointer to `short int` (`h`) or `long int` (`l`).
`h` or `l` (ell) preceded by `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` (ell) or `L` preceded by `e`, `f` and `g` : the corresponding argument is a pointer to `double` (`l`) or `long double (L)`.

  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.

`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 characters) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters is executed by reading input up to the first wide character which is not a white-space character (which remains unread), or until no more wide characters can be read (EOF).

A directive that is an ordinary wide 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 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 characters (up to any specified maximum field width) that is an initial subsequence of a matching sequence. The first wide character after the input item, if any, 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 input error such as EOF, for example, or the occurrence of a read error prevents further input.

Except in the case of a `%` conversion character, the input item (or, in the case of a *%n*, the number of input characters read) is converted to a data 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 as shown below:

```
                    ⎧[{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 the `wcstol` function with the value 10 for *base* (*base* = 10). 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 the `wcstol` function with the value 8 for *base* (*base* = 8). 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* (*base* = 8). 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 the `wcstoul` function with the value 10 for *base* (*base* = 10). 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 the `wcstoul` function with the value 16 for *base* (*base* = 16). 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 non-white-space wide characters. If no `l` (ell) qualifier is specified, characters from the input field are converted as if by repeated calls to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted. The sequence is written up to the terminating null character. The corresponding argument should be a pointer to a `char` array that is large enough to accept the converted sequence and a terminating null character, which will be added automatically. If `l` is specified, the corresponding argument should be a pointer to the initial element of a `wchar_t` array that is large enough to accept the sequence and a terminating null wide character, which will be added automatically
`[`	Matches a non-empty sequence of wide characters from a set of expected wide characters (the scanset). If no `l` (ell) qualifier is specified, characters from the input field are converted as if by repeated calls to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted. The sequence is written up to the terminating null character. The corresponding argument should be a pointer to a `char` array that is large enough to accept the converted sequence and a terminating null character, which will be added automatically.

If `l` is specified, the corresponding argument should be a pointer to the initial element of a `wchar_t` array that is large enough to accept the sequence and a terminating null wide character, which will be added automatically.
The conversion specification includes all subsequent wide characters (i.e. characters after the `[`) 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 wide character 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 second where the first character is a `[` or `[^`, nor the last character, the behavior is undefined.

`c` Matches a sequence of wide characters of the number specified by the field width. if no field width is specified in the conversion specification, 1 wide character is read.
If no `l` (ell) qualifier is specified, characters from the input field are converted as if by repeated calls to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted. The corresponding argument should be a pointer to a `char` array that is large enough to accept the converted sequence. No null character will be added.
If `l` is specified, the corresponding argument should be a pointer to the initial element of a `wchar_t` array that is large enough to accept the sequence. No null character will be added.

The normal skip over white-space characters is suppressed in this case; `%1s` should be used to read the next wide character 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 `int` into which the number of wide characters read thus far by this call are to be entered. Execution of a *%n* conversion specification does not increment the assignment counter returned on completing the 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 error. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, an execution other than *%n* of the following conversion specification (if any) is terminated with an input error.

Reaching the end of the character string in a `swscanf` call is equivalent to encountering the end-of-file indicator during an `fwscanf` call.

Any trailing white space (including end-of-file 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 successfully read and assigned input items

if no input error occurs before the first assignment.
The number is zero if a format error occurs at the first input item.

EOF if an input error occurs before the first assignment

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also scanf, sscanf, fscanf, wcstod, wcstol, wcstoul, wcrtomb

fwrite - Write blockwise to a file

Definition

```
#include <stdio.h>

size_t fwrite(const void *p, size_t elsize, size_t n, FILE *fp);
```

`fwrite` writes *n* elements (*elsize* bytes in size each) from the area pointed to by *p* to the file with file pointer *fp*.
The position of the read/write pointer is subsequently advanced by the number of bytes written.

Return val.

Number of elements actually written	if successful.
0	for end of file or error.

Notes

To ensure that *elsize* specifies the correct number of bytes for a data element, you should use the `sizeof` function for the size of a data unit to which *p* points.

For output to files with stream I/O the characters are not written immediately to the external file but are stored in an internal C buffer (see section “Buffering” on page 63).

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” on page 65).

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 indexed-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* = element length * number of elements
>
> 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 indexed-sequential file (ISAM), *n* does not need to be equal to the length of the record to be updated. A record can therefore 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, namely the number of elements written in their entirety. For record I/O it is best to use only element length 1 since in this case the return value corresponds 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.

Example

The following program transfers two personal data items to the file with file pointer *p_list*.

```
#include <stdio.h>

int main(void)
{
  FILE *p_list;
  size_t result;
  static struct p
  {
    char name[20];
    int a;
  } person[2] =
    {
      ≥"ANNE", 30¸,
      Ã"JOHN", 60Õ,
    };

  p_list = fopen("link=link", "w");

  result = fwrite(person, sizeof(struct p), 2, p_list);
  printf("%d Personal data written\n", result);
  return 0;
}
```

See also

fread, feof, ferror

gamma - Logarithmic gamma function

Definition

<math.h> is included like this:

```
#include <math.h>
double gamma(double x);
```

`gamma` calculates 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.

`gamma(x)`	if successful.
HUGE_VAL	if the correct value results in an overflow. In addition, `errno` is set to ERANGE (result too large).
HUGE_VAL	if x is a non-positive integer. In addition, `errno` is set to EDOM (illegal argument).

garbcoll - Release memory space to the system

Definition #include <stdlib.h>

void garbcoll(void);

The `calloc`, `malloc`, `realloc` and `free` functions form the C-specific memory management package. This package essentially consists of an internal free memory management facility.
The memory released with `free` is not returned to the system (RELM-SVC) but is taken by the free memory management facility.
The memory request functions (`calloc`, `malloc`, `realloc`) attempt to supply the memory firstly via the free memory management and only as a second option by the operating system (REQM-SVC).

If memory is no longer available even from the system, the memory administered by the free memory management facility is returned (pagewise 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.

Note The `garbcoll` function returns to the system all the memory areas which were previously released with `free` and which can be combined to form free pages.

See also calloc, malloc, realloc, free

gcvt - Convert a floating-point number to a string

Definition
#include <stdlib.h>

char *gcvt(double value, int n, char *buf);

`gcvt` converts a floating-point number *value* to a string of digits and stores the string in the area pointed to by *buf*. A pointer to this area is returned as a result.

Depending on the structure of the floating-point value to be converted, the output format corresponds to

– the FORTRAN F format: *n* significant digits, no leading or trailing zeros from *value*, a negative sign if required, and a decimal point (if there are any non-zero digits after the decimal point)

– or the FORTRAN E format (exponential notation).

Parameters
double value
Floating-point value to be edited for output.

int n
Number of digits in the resulting string (calculated as of the first non-zero digit in the floating-point value to be converted).

If *n* is less than the number of digits in *value*, the least significant digit is rounded.
If *n* is greater, the string ends with the last non-zero digit.

char *buf
Pointer to the converted string.
The memory area to which *buf* points should be at least (*n* + 4) bytes in size!

Return val.
Pointer to the converted string.
`gcvt` closes the string with the null byte (\0).

Notes
Invalid parameters, such as an `integer` instead of a `double` value, cause the program to abort!

It is your responsibility to ensure that the result pointer *buf* points to a memory area of at least (*n* + 4) bytes (see example).

Example The program reads a floating-point value *x*, converts it as specified in *n*, and outputs it as a string to the `char` array *buf*. The malloc function is used to reserve (*n* + 4) bytes.

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

int main(void)
{
  double x;
  int n;
  char *buf;
  printf("Please enter floating-point number: \n");
  if ( scanf("%lf",&x) == 1)
  {
    printf("How many significant digits : \n");
    if ( scanf("%d",&n) == 1)
    {
      buf = (char *)malloc(n + 4);
      printf("After conversion, the number is :  %s \n", gcvt(x, n, buf));

    }
  }
  return 0;
}
```

See also ecvt, gcvt

getc - Read a character from a file

Definition #include <stdio.h>

int getc(FILE *fp);

`getc` reads a character from the file with file pointer *fp* from the current read/write position.

Return val. Character read as a positive `integer` value if successful.

EOF in case of an error or end of file.

Notes getc is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

The call `getc(stdin)` is identical to `getchar()`.

If you use a comparison such as `while((c = getc(fp)) != EOF)` in your program, the variable *c* must always be declared as an `integer`. If you define *c* as a `char`, the EOF condition is never satisfied for the following reason: -1 is converted to `char` ’`0xFF`’ (i.e. +255); EOF, however, is defined as -1.

If `getc` is reading from the standard input `stdin`, and EOF is the end criterion for reading, you can satisfy the EOF condition by means of the following actions at the terminal: pressing the K2 key and entering the system commands EOF and RESUME-PROGRAM.

Example The following program reads a file with file pointer *fp* one character at a time until end of file is reached. The read characters are stored in the area *buf*.

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

int main(void)
{
  int c, i = 0;
  char buf[BUFSIZ];
  FILE *fp;
  char name[40];
  printf("Please enter file to be read\n");
  scanf("%s", name);
  if(( fp = fopen(name, "r")) == NULL)
    {
    perror("fopen");  /* Abort with error message 'fopen' if */
    exit(1);          /* file does not exist                 */
    }
  while (( c = getc(fp)) != EOF )
    buf[i++] = c;
  puts(buf);
  fclose(fp);
  return 0;
}
```

See also fgetc, getchar, getwc, ungetc, fopen, fopen64

getchar - Read a character from standard input

Definition #include <stdio.h>

int getchar(void);

`getchar` reads a character from the standard input (file pointer `stdin`).

Return val. Character read as a positive `integer` value if successful.

EOF for end of file or error.

Notes `getchar` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

If you use a comparison such as `while((c = getchar()) != EOF)` in your program, the variable *c* must always be declared as an `integer`. If you define *c* as a `char`, the EOF condition is never satisfied for the following reason: -1 is converted to `char '0xFF'` (i.e. +255); EOF, however, is defined as -1.

You can satisfy the EOF condition when reading from the terminal by means of the following actions:
pressing the K2 key and entering the system commands EOF and RESUME-PROGRAM.

See also getc, fgetc, getwchar

getenv - Query system information

Definition #include <stdlib.h>

char *getenv(const char *name);

In this implementation, `getenv` is merely a dummy function. The return value is always a NULL pointer. The specified parameter is not evaluated.

Return val. NULL pointer.

getlogin - Query user ID

Definition #include <stdlib.h>

char *getlogin(void);

`getlogin` returns the login name (i.e. userid) under which the calling program is being executed.

Return val. Pointer to the name of the user id.

Note `getlogin` writes its result into an internal C data area that is overwritten with each call!

Example

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

int main(void)
{
  printf("Example showing the use of getlogin():\n");
  printf("Userid = %s\n", getlogin());
  return 0;
}
```

getpgmname - Query program name

Definition #include <stdlib.h>

char *getpgmname(void);

`getpgmname` returns the name of the calling program. The result corresponds to `argv[0]` of the `main` function.

Return val. Pointer to the program name.

Example

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

int main(void)
{
  printf("Example showing the use of getpgmname():");
  printf("Program name = %s\n", getpgmname());
  return 0;
}
```

gets - Read a string from standard input

Definition

#include <stdio.h>

char *gets(char *s);

`gets` reads characters from the standard input `stdin` until the next newline and stores the string in the area pointed to by *s*, replacing the newline with the null byte (\0).

Return val.

Pointer to the result string
if successful. `gets` terminates the string with the null byte (\0).

NULL pointer if end of file is reached or a read error occurs.

Notes

You must explicitly provide the area in which `gets` is to store the string read!

In contrast to `fgets`, `gets` deletes a read newline character, i.e. overwrites it with the null byte.

You can satisfy the EOF condition when reading from the terminal by means of the following actions:
pressing the K2 key and entering the system commands EOF and RESUME-PROGRAM.

Example

The following program reads strings from the standard input and writes them to the standard output. The reading can be terminated with the K2 key and the EOF and RESUME-PROGRAM commands.

```
#include <stdio.h>
int main(void)
{
 char s[BUFSIZ];
 while(gets(s) != NULL)
      puts(s);
 return 0;
}
```

See also

fgets, puts, fputs, getws

gettsn - Query TSN (task sequence number)

Definition #include <stdlib.h>

char *gettsn(void);

`gettsn` returns the task sequence number (TSN) of the calling program.

Return val. Pointer to the task sequence number (TSN).

Note `gettsn` writes its result into an internal C data area that is overwritten with each call!

Example

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

int main(void)
{
  printf("Example showing the use of gettsn():\n");
  printf("The TSN number of the program %s : %s\n", getpgmname(), gettsn());
  return 0;
}
```

getw - Read wordwise from a file

Definition #include <stdio.h>

int getw(FILE *fp);

`getw` reads a machine word from the file with the file pointer *fp* and positions the read/write pointer after the word read.

A machine word may be conceived of as a binary `integer` value.

Return val. word read as an `integer` value
if successful.

EOF or end of file or error.

Notes Since word length and byte arrangement are system-dependent, it is possible that files written with `putw` on a non-BS2000 operating system may not be readable with `getw` in BS2000.

Since EOF represents a valid `integer` value, you should use the functions `feof` and `ferror` to check for end of file or error conditions.

Example The following program fragment reads wordwise from the file with file pointer *fp* until end of file is reached.

```
int buf[MAX];
int i = 0;
FILE *fp;

while(!feof(fp) && !ferror(fp))
     buf[i++] = getw(fp);
```

See also putw

getwc - Read a wide character from a file

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

wint_t getwc(FILE *fp);

`getwc` is equivalent to `fgetwc`, except for the fact that it is implemented as a macro and can evaluate *fp* more than once, so the argument should never be an expression with side effects.

Return val.
Wide character code of type `wint_t`
if successful.

WEOF
if the end-of-file is reached. The end-of-file indicator for the file is set;
or
if a read error occurs. The error indicator for the file is set, and `errno` is set to EBADF if *fp* is an invalid file pointer.

Notes
This version of the C runtime system only supports one-byte characters as wide character codes.

`getwc` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

This interface was provided to support some current implementations and possible future ISO standards.

If `getwc` is used as a macro, an *fp* expression with side effects may be handled incorrectly. In particular, `getwc(*f++)` may not work as expected. For this reason, it is better to use `fgetwc` in such situations instead of `getwc`.

You can satisfy the WEOF condition when reading from the terminal by means of the following actions: pressing the K2 key and entering the system commands EOF and RESUME-PROGRAM.

See also
fgetwc, getc

getwchar - Read a wide character from standard input

Definition #include <wchar.h>

wint_t getwchar(void);

The function call `getwchar(void)` is equivalent to `getwc(stdin)`, i.e. reads a wide character from standard input.

You can satisfy the WEOF condition when reading from the terminal (`stdin`) by means of the following actions: pressing the K2 key and entering the system commands EOF and RESUME-PROGRAM.

Return val. Wide character code of type `wint_t`
if successful.

WEOF if the end-of-file is reached. The end-of-file indicator for the file is set;
or
if a read error occurs. The error indicator for the file is set, and `errno` is set to EBADF if *fp* is an invalid file pointer.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also fgetwc, getwc

gmtime, gmtime64 - Convert date and time to UTC

Definition #include <time.h>

struct tm *gmtime(const time_t *sec_p);
struct tm *gmtime64(const time64_t *sec_p);

`gmtime` and `gmtime64` interpret the time specification to which *sec_p* points as the number of seconds which have passed since the reference date (epoch). The functions calculate the date and time from this and store the result in a structure of the type `tm` in UTC format (Universal Time Coordinated). Negative values are interpreted as seconds before the reference date. The earliest displayable date is 01/01/1900 00:00:00 local time.

`gmtime` and `gmtime64` correspond to the `localtime` and `localtime64` functions, each supplying the local time.

With `gmtime` the reference date depends on the use of the TIMESHIFT bind option (see section “Time functions” on page 40):
– without TIMESHIFT bind option (default): 1/1/1950 00:00:00.
– with TIMESHIFT bind option: 1/1/1970 00:00:00.
With `gmtime64` the reference date is always 1/1/1970 00:00:00

The latest date which can be displayed with `gmtime` is 01/19/2018 03:14:07 (without TIMESHIFT bind option) or 01/19/2038 03:14:07 (with TIMESHIFT bind option).

Irrespective of the use of the TIMESHIFT bind option, `gmtime64` can display dates up to 3/18/4317 02:44:48.

Return val. Pointer to the calculated structure. `gmtime` and `gmtime64` store the result in a structure declared in <time.h> as follows:

```
struct tm
{
  int   tm_sec;        /* seconds (0-59) */
  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 from the start of the year (0-11) */
  int   tm_year;       /* years since 1900 */
  int   tm_wday;       /* weekday (0-6, 0=Sunday) */
  int   tm_yday;       /* days since January 1 (0-365) */
  int   tm_isdst;      /* daylight saving time flag */
};
```

NULL In the event of an error

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.

Example

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

struct tm *t;
char *s;
time_t clk;

int main(void)
{
  clk = time((time_t *)0);
  t = gmtime(&clk);
  printf("Year: %d\n", t->tm_year + 1900);
  printf("Time in hours: %d\n", t->tm_hour);
  printf("Day of the year: %d\n", t->tm_yday);
  s = asctime(t);
  printf("%s", s);
  return 0;
}
```

See also

asctime, ctime, ctime64, localtime, localtime64

hypot - Euclidean distance

Definition #include <math.h>

double hypot(double x, double y);

`hypot` calculates the euclidean distance of the point with the coordinates (x,y).

Return val.

sqrt(x*x + y*y)	square root of the sum of the squared coordinates.
HUGE_VAL	in the event of an overflow. In addition, `errno` is set to ERANGE (result too large).

Example

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

int main(void)
{
  double x, y, alpha, r, pi;

  printf("Enter x and y coordinates:\n");
  scanf("%lf %lf", &x, &y);

  pi = 2.0 * asin(1.0);

  if(x > 0.0)
    alpha = atan(y/x);
  else if (x < 0.0)
          if (y >= 0.0)
             alpha = atan(y/x) + pi;
          else alpha = atan(y/x) - pi;
       else if (y > 0)
               alpha = pi/2.0;
            else if (y < 0)
                    alpha = -pi/2.0;
                 else
                 {
                   printf("Angle not defined!\n");
                   exit(1);
                 }
r = hypot(x, y);
  alpha = alpha * (180.0/pi);
```

```
  printf("The polar coordinates are:\n");
  printf("Distance from zero: %g\n",r);
  printf("Angle to the x axis:\n");
  printf("%g degrees\n",((y < 0.0)? alpha + 360 : alpha) );
  return 0;
}
```

See also cabs, sqrt

ieee2double - Convert floating-point number from IEEE format to /390 format

Definition #include <ieee_390.h>

extern double ieee2double (double num);

`ieee2double` converts an 8-byte floating-point number *num* from IEEE format to /390 format and returns it as the result. There is no loss of precision.

Parameters double num

8-byte floating-point number in IEEE format

Return val. 8-byte floating-point number in /390 format (in the event of success)

`0.0` if the IEEE floating-point number is smaller than the smallest number that can be represented in /390 format or if `NaN` or `inf` is passed as a parameter.

If the IEEE floating-point number is greater than the largest number that can be represented in /390 format, this largest representable number is returned with the corresponding sign.

The global variable *float_exceptions_flag* contains information for the event of unsuccessful conversion and is defined as follows:

```
extern int float_exception_flags;
enum {
   float_flag_inexact   =  1,
   float_flag_divbyzero =  2,
   float_flag_underflow =  4,
   float_flag_overflow  =  8,
   float_flag_invalid   = 16
};
```

If the IEEE floating-point number is greater than the largest number that can be represented in /390 format, *float_flag_overflow* is set.

If the IEEE floating-point number is smaller than the smallest number that can be represented in /390 format, *float_flag_underflow* is set.

If `NaN` or `inf` is passed as a parameter, *float_flag_invalid* is set.

See also float2ieee, float2ieee, double2ieee

ieee2float - Convert floating-point number from IEEE format to /390 format

Definition

#include <ieee_390.h>

extern float ieee2float (float num);

`ieee2float` converts a 4-byte floating-point number *num* in IEEE format to /390 format and returns it as the result. Neither overflow nor underflow can occur, but up to three bit positions can be lost.

Parameters

float num
    4-byte floating-point number in IEEE format

Return val.

4-byte floating-point number in /390 format.

The global variable *float_exceptions_flag* contains information for the event of unsuccessful conversion and is defined as follows:

```
extern int float_exception_flags;
enum {
   float_flag_inexact   =  1,
   float_flag_divbyzero =  2,
   float_flag_underflow =  4,
   float_flag_overflow  =  8,
   float_flag_invalid   = 16
};
```

If bit positions are lost during conversion and the result thus becomes inaccurate, *float_flag_invalid* is set.

See also

float2ieee, double2ieee, ieee2double

index - First occurrence of a character in a string

Definition

#include <string.h>

char *index(const char *s, int c);

`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*.

Note

The `index` and `strchr` functions are equivalent.

Example

Find the first ’f’:

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

int main(void)
{
   char *s = "What ffun in the sun!";
   printf("%s\n", s);
   printf("Where is the error? %s\n", index(s, 'f'));
   return 0;
}
```

See also

rindex, strchr

isalnum - Test for letter or digit

Definition

#include <ctype.h>

int isalnum(int c);

`isalnum` checks whether the character *c* from the EBCDIC character set is alphanumeric, i.e. a letter (A-Z, a-z) or a digit (0-9).

Return val.

≠ 0	*c* is alphanumeric.
0	*c* is not alphanumeric.

Note

`isalnum` is implemented both as a macro and as a function (see section "Functions and macros" on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
    printf("%s : %c\n", ((isalnum(c)) ? "Alphanumeric" : "Other"), c);
  return 0;
}
```

See also

isalpha, isascii, iscntrl, isdigit, isgraph, islower, ispunct, isprint, isspace, isupper, isxdigit, isebcdic, iswalnum

isalpha - Test for letter

Definition

#include <ctype.h>

int isalpha(int c);

`isalpha` checks whether the character *c* is a letter (A-Z, a-z).

Return val.

≠ 0	*c* is a letter.
0	*c* is not a letter.

Note

`isalpha` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n", ((isalpha(c)) ? "Letter" : "Other"), c);
  return 0;
}
```

See also

isalnum, isascii, iscntrl, isdigit, isgraph, islower, ispunct, isprint, isspace, isupper, isxdigit, isebcdic, iswalpha

isascii - Test for ASCII character

Definition

`#include <ctype.h>`

`int isascii(int c);`

`isascii` is a synonym for `isebcdic`. On EBCDIC computers, `isascii` checks whether the value of the character *c* represents an EBCDIC character (values 0 - 255). If portability to ASCII computers is required, `isascii` should be used.

Return val.

≠ 0	the value of *c* represents an EBCDIC character (values 0 - 255),
0	*c* doesn't represent an EBCDIC character (values ≠ 0 - 255).

See also

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, isebcdic

iscntrl - Test for control character

Definition #include <ctype.h>

int iscntrl(int c);

`iscntrl` checks whether the character *c* from is a control character. Control characters are non-printable characters (e.g. for printer control). The non-printable characters for white space are not included (see `isspace`).

Return val.		
	≠ 0	*c* is a control character.
	0	*c* is not a control character.

Note `iscntrl` is implemented both as a macro and as a function (see section "Functions and macros" on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n", ((iscntrl(c)) ? "Control character":"Other"), c);
  return 0;
}
```

See also isalnum, isascii, isalpha, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, isebcdic, iswcntrl

isdigit - Test for digit

Definition #include <ctype.h>

int isdigit(int c);

`isdigit` checks whether the character *c* is a digit (0-9).

Return val.

≠ 0	*c* is a digit.
0	*c* is not a digit.

Note `isdigit` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n", ((isdigit(c)) ? "Digit" : "Other"), c);
  return 0;
}
```

See also isalnum, isascii, iscntrl, isalpha, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, isebcdic, iswdigit

isebcdic - Test for EBCDIC character

Definition

#include <ctype.h>

int isebcdic(int c);

`isebcdic` checks 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	*c* doesn’t represent an EBCDIC character (values ≠ 0 - 255).

Notes

`isebcdic` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

`isebcdic` is a synonym for `isascii`. If portability to ASCII computers is required, `isascii` should be used instead of `isebcdic`.

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
    printf("%s : %c\n", ((isebcdic(c)) ? "EBCDIC character" : "Other"), c);
  return 0;
}
```

See also

isalpha, isalnum, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

isgraph - Test for printable character except space

Definition #include <ctype.h>

int isgraph(int c);

`isgraph` checks whether the character *c* is a printable character, i.e. an alphanumeric or a special character. Spaces are considered to be non-printable.
c is the value of the character to be checked.

Return val.

≠ 0	*c* is printable and not a space.
0	*c* is non-printable or space.

Note `isgraph` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isgraph(c))? "Character" : "Cannot print"), c);
  return 0;
}
```

See also isalnum, isascii, iscntrl, isdigit, islower, isalpha, ispunct, isprint, isspace, isupper, isxdigit, isebcdic, iswgraph

islower - Test for lowercase letter

Definition #include <ctype.h>

int islower(c);

`islower` checks whether the character *c* is a lowercase letter (a-z).
c is the value of the character to be checked.

Return val. $\neq 0$ *c* is a lowercase letter.

0 *c* is not a lowercase letter.

Note `islower` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n", ((islower(c)) ? "Lowercase letter" : "Other"), c);
  return 0;
}
```

See also isalnum, isascii, iscntrl, isdigit, isgraph, isalpha, isprint, ispunct, isspace, isupper, isxdigit, isebcdic, iswlower

isprint - Test for printable character including space

Definition #include <ctype.h>

int isprint(int c);

`isprint` checks whether the character *c* is a printable character, i.e. an alphanumeric character, a special character, or a space.

Return val.

≠ 0	*c* is printable (including space).
0	*c* is non-printable.

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isprint(c))? "Character" : "Cannot print"), c);
  return 0;
}
```

See also isalnum, isascii, iscntrl, isdigit, isgraph, islower, isalpha, ispunct, isspace, isupper, isxdigit, isebcdic, iswprint

ispunct - Test for special character

Definition

#include <ctype.h>

int ispunct(c);

`ispunct` checks whether the character *c* is a special character, i.e. not a control, alphanumeric, or white space character (see `isspace`).

Return val.

≠ 0	*c* is a special character.
0	*c* is not a special character.

Note

`ispunct` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((ispunct(c))? "Special character" : "Other"), c);
  return 0;
}
```

See also

isalnum, isascii, iscntrl, isdigit, isgraph, islower, isalpha, isprint, isspace, isupper, isxdigit, iswpunct

isspace - Test for white space character

Definition

#include <ctype.h>

int isspace(int c);

`isspace` checks whether the character *c* from the EBCDIC character set is a white space character, i.e. a blank, horizontal tab (\t), carriage return (\r), newline (\n), form feed (\f), or vertical tab (\v).

Return val.

≠ 0	*c* is a white space character.
0	*c* is not a white space character.

Notes

`isspace` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

For evaluating control characters for white space (see section “White space” on page 65).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isspace(c))? "White space " : "Other"),c);
  return 0;
}
```

See also

isalnum, isalpha, isascii, iscntrl, isdigit, islower, isprint, isgraph, ispunct, isupper, isxdigit, isebcdic, iswspace

isupper - Test for uppercase letter

Definition

#include <ctype.h>

int isupper(int c);

`isupper` checks whether the character *c* is an uppercase letter (A-Z).

Return val.

≠ 0	*c* is an uppercase letter.
0	*c* is not an uppercase letter.

Note

`isupper` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isupper(c))? "Uppercase letter " : "Other"),c);
  return 0;
}
```

See also

isalnum, isascii, iscntrl, isdigit, islower, isprint, ispunct, isgraph, isspace, isalpha, isxdigit, isebcdic, iswupper

iswalnum - Test for alphanumeric wide character

Definition #include <wctype.h>

int iswalnum(wint_t wc);

`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.

$\neq 0$	*wc* is alphanumeric.
0	*wc* is not alphanumeric.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

`iswalnum` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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`.

See also isalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswalpha - Test for alphabetic wide character

Definition

#include <wctype.h>

int iswalpha(wint_t wc);

`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.

$\neq 0$	*wc* is a letter.
0	*wc* is not a letter.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

`iswalpha` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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`.

See also

isalpha, iswalnum, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswcntrl - Test for control wide character

Definition #include <wctype.h>

int iswcntrl(wint_t wc);

`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.

≠ 0	*wc* is a control character.
0	*wc* is not a control character.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

`iswcntrl` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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`.

See also iscntrl, iswalnum, iswalpha, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswctype - Test wide character for class

Definition

#include <wctype.h>

int iswctype(wint_t wc, wctype_t charclass);

`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.

Return val.

≠ 0	The wide character is in character class *charclass*.
0	The wide character not in the character class *charclass*.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

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

iswdigit - Test for decimal-digit wide character

Definition

```
#include <wctype.h>
int iswdigit(wint_t wc);
```

`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.

Return val.

≠ 0	*wc* is a decimal digit.
0	*wc* is not a decimal digit.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

`iswdigit` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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

isdigit, iswalnum, iswalpha, iswcntrl, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit

iswgraph - Test for visible wide character

Definition #include <wctype.h>

int iswgraph(wint_t wc);

`iswgraph` tests whether the wide character specified by *wc* is a character with a visible representation, i.e. an alphanumeric or 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. $\neq 0$ *wc* is a character with a visible representation.

0 *wc* is not a character with a visible representation.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

`iswgraph` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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`.

See also isgraph, iswalnum, iswalpha, iswcntrl, iswdigit, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswlower - Test for lowercase wide character

Definition #include <wctype.h>

int iswlower(wint_t wc);

`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.

$\neq 0$	*wc* is a lowercase letter.
0	*wc* is not a lowercase letter.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

`iswlower` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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`.

See also islower, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswprint - Test for printing wide character

Definition

#include <wctype.h>

int iswprint(wint_t wc);

`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.

$\neq 0$	*wc* is a printing wide character (alphanumeric characters, special characters, and blanks).
0	*wc* is not a printing wide character.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

`iswprint` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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

isprint, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswpunct - Test for punctuation wide character

Definition

#include <wctype.h>

int iswpunct(wint_t wc);

`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.

$\neq 0$	*wc* is a punctuation wide character.
0	*wc* is not a punctuation wide character

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

`iswpunct` is implemented both as a function and as a macro (see section "Functions and macros" on page 17).

The behavior of `iswpunct` is determined by the class `punct` of the current locale. The current locale is the C locale, unless it was explicitly changed using `setlocale`.

See also

ispunct, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswspace, iswupper, iswxdigit, setlocale

iswspace - Test for white-space wide character

Definition #include <wctype.h>

int iswspace(wint_t wc);

`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.

$\neq 0$	*wc* is a white-space wide character.
0	*wc* is not a white-space wide character.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

`iswspace` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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`.

See also isspace, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswupper, iswxdigit, setlocale

iswupper - Test for uppercase wide character

Definition

#include <wctype.h>

int iswupper(wint_t wc);

`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.

$\neq 0$	*wc* is an uppercase letter.
0	*wc* is not an uppercase letter.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

`iswupper` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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`.

See also

isupper, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswxdigit, setlocale

iswxdigit - Test for hexadecimal wide-character digit

Definition #include <wctype.h>

int iswxdigit(wint_t wc);

`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.

$\neq 0$	*wc* is a hexadecimal digit.
0	*wc* is not a hexadecimal digit

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

`iswxdigit` is implemented both as a function and as a macro (see section “Functions and macros” on page 17).

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`.

See also iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, isxdigit

isxdigit - Test for hexadecimal digit

Definition #include <ctype.h>

int isxdigit(int c);

`isxdigit` checks whether the character *c* from the EBCDIC character set is a hexadecimal digit (0-9), (A-F) or (a-f).

Return val. ≠ 0 *c* is a hexadecimal digit.

0 *c* is not a hexadecimal digit.

Note `isxdigit` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

Example

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

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
  printf("%s : %c\n",((isxdigit(c))? "Hexadecimal digit" : "Other"), c);
  return 0;
}
```

See also isalnum, isascii, iscntrl, isalpha, islower, isprint, ispunct, isgraph, isspace, isupper, isdigit, isebcdic

j0, j1, jn - Bessel functions of the first kind

Definition

```
#include <math.h>

double j0(double x);

double j1(double x);

double jn(int n, double x);
```

The functions `j0`, `j1` and `jn` calculate the Bessel functions of the first kind for floating-point values x and the integer orders 0, 1 or n.

Return val. Bessel function for x.

See also y0, y1, yn

kill - Send signal to own program

Definition

#include <signal.h>

int kill(int pn, int sig);

`kill` continues to be supported for compatibility reasons; it works like the ANSI function `raise`.

The only difference is that the `kill` function expects the program number *pn* as the first argument, which must always be 0 since the signal may only be sent to its own program (see also return value -1).

Return val.

0	The signal was sent successfully.
-1	The signal could not be sent, because – *sig* is not a valid signal number or – the program number *pn* is not equal to 0. In addition, `errno` is set to the appropriate program error code: EINVAL (invalid signal number) ESRCH (program number not 0).

Example

A program that aborts itself.

```
#include <signal.h>

int main(void)
{
  for(;;)
    kill(0, SIGKILL);
  return 0;
}
```

See also

alarm, raise, signal

labs - Absolute value of an integer (long int)

Definition
#include <stdlib.h>

long int labs(long int j);

`labs` calculates the absolute value of an integer *j* of type `long` int.

Return val.

`|j|`	for an integer *j*.
undefined	in case of over- or underflow. `errno` is set to ERANGE to indicate the error.

Note
The absolute value of the highest presentable negative number cannot be presented. If the highest negative number of type `long int` is specified as argument *j*, the program is terminated with an error (ERANGE).

See also
abs, cabs, fabs, llabs

ldexp - Calculate binary value

Definition

#include <math.h>

double ldexp(double x, int exp);

Given its arguments *x* (mantissa) and *exp* (exponent), `ldexp` calculates the number:

$$x * 2^{exp}$$

`ldexp` is the inverse function of `frexp`.

Return val.

$x * 2^{exp}$ if successful.

+/-HUGE_VAL in the event of an overflow (depending on the sign for *x*). In addition, `errno` is set to ERANGE (result too large).

Example

`ldexp` is the inverse function of `frexp`:
`frexp` splits its floating-point argument into mantissa and exponent to the base 2, while `ldexp` uses these parts to calculate the original value in its internal floating-point representation. This is shown below for the number 5.342:

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

int main(void)
{
  double x;
  int ex;

  x = frexp(5.342, &ex);
  printf("Mantissa : %f\nexponent : %d\n", x, ex);
  printf("Initial value : %f\n", ldexp(x, ex));
  return 0;
}
```

See also

frexp, modf

ldiv - Division with integers (long int)

Definition #include <stdlib.h>

ldiv_t ldiv(long int dividend, long int divisor);

`ldiv` calculates the quotient and the remainder of the division of *dividend* by *divisor*.
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`
containing both the quotient *quot* and the remainder *rem* as `integer` values.

Example

```
ldiv_t d;

d = ldiv( 7, 3);        /*  d.quot =  2,  d.rem =  1  */
d = ldiv(-7, 3);        /*  d.quot = -2,  d.rem = -1  */
d = ldiv( 7,-3);        /*  d.quot = -2,  d.rem =  1  */
d = ldiv(-7,-3);        /*  d.quot =  2,  d.rem = -1  */
```

See also div, lldiv

_ _LINE_ _ - Output the current source program line number

Definition _ _LINE_ _

This macro generates the current line number of the source program as a decimal number.

Note This macro does not need to be defined in an include file. Its name is recognized and replaced by the compiler.

llabs - Absolute value of an integer (long long int)

Definition #include <stdlib.h>

long long int llabs(long long int j);

`llabs` calculates the absolute value of an integer *j* of type `long long` int.

Return val.

`|j|`	for an integer *j*.
undefined	in case of over- or underflow. `errno` is set to ERANGE to indicate the error.

Note The absolute value of the highest presentable negative number cannot be presented. If the highest negative number of type long `long int` is specified as argument *j*, the program is terminated with an error (ERANGE).

See also abs, cabs, labs

lldiv - Division with integers (long long int)

Definition

<stdlib.h> include line: #include <stdlib.h>

lldiv_t lldiv(long long int dividend, long long int divisor);

`lldiv` calculates the quotient and the remainder of the division of *dividend* by *divisor*. 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`
containing both the quotient *quot* and the remainder *rem* as `integer` values.

Example see `ldiv`.

See also div, ldiv

llrint, llrintf, llrintl - Round off to nearest whole number

Definition

```
#include <math.h>
long long int llrint(double x);
long long int llrintf (float x);
long long int llrintl (long double x);
```

Each of the functions returns the whole number nearest to x, represented as a number of type `long long int`.

The return value is rounded off in accordance with the rounding mode currently set for the system. If the rounding mode is ‘round-to-nearest’ and if the difference between x and the rounded result is exactly 0.5, the nearest even number is returned.

If the rounding mode currently set rounds off in the direction of positive infinity, then `llrint` is equivalent to `ceil`. If the defined rounding mode rounds off in the direction of negative infinity, then `llrint` is equivalent to `floor`.
In this version, the rounding mode is preset in the direction of positive infinity.

Return val.

integer	represented as a number of type `long long int` if successful.
undefined	in the event of an overflow or underflow, `errno` is set to ERANGE to indicate the error.

Note

In this version, the rounding mode is preset in the direction of positive infinity.

See also

abs, ceil, floor, llround, lrint, lround, rint, round

llround, llroundf, llroundl - Round off to nearest whole number

Definition #include <math.h>

long long int llround(double x);

long long int llroundf (float x);

long long int llroundl (long double x);

Each of the functions returns the whole number nearest to x, represented as a number of type `long long int`.

The return value is independent of the defined rounding mode. If the difference between x and the rounded result is exactly 0.5, the larger whole number is returned.

Return val.

integer	represented as a number of type `long long int` if successful.
undefined	in the event of an overflow or underflow, `errno` is set to ERANGE to indicate the error.

See also abs, ceil, floor, llrint, lrint, lround, rint, round

localeconv - Query/change locale-specific data

Definition #include <locale.h>

struct lconv *localeconv(void);

`localeconv` sets the components of a structure of type `struct lconv` to values which match the current locale. The supplied values can be used in formatted output to represent monetary and non-monetary numerical values on a locale-specific basis.

At the start of the program the default locale is "C" (LC_C_C). The locale can be changed by calling the `setlocale` function with the categories LC_MONETARY, LC_NUMERIC or LC_ALL. When `localeconv` is called again, it matches the values in the structure components to the new locale.

Return val. Pointer to the structure in which the values have been entered.

1. Components for non-monetary numerical values (LC_NUMERIC):

   char *decimal_point
   Decimal point.

   char *thousands_sep
   Separator for grouping the digits in front of the decimal point.

   char *grouping
   String whose elements specify the length of each group of digits.

2. Components for monetary values (LC_MONETARY):

   char *int_curr_symbol

   The international currency symbol appropriate to the locale. The first three characters contain the alphabetic international currency symbol, in accordance with the convention defined in ISO 4217:1897. The fourth character is the separator between the international currency symbol and the amount.
   In the locale “De.EDF04F@euro”, the value “EUR” is entered as an alphabetical currency symbol.

   char *currency_symbol
   The currency symbol corresponding to the locale.

   char *mon_decimal_point
   Decimal point.

   char *mon_thousands_sep
   Separator for grouping the digits in front of the decimal point.

char *mon_grouping
String whose elements specify the length of each group of digits.

char *positive_sign
String indicating a non-negative amount.

char *negative_sign
String indicating a negative amount.

char int_frac_digits
Number of decimal places for an internationally structured amount.

char frac_digits
Number of decimal places for a locally structured amount.

char p_cs_precedes
1 if the currency symbol precedes the non-negative amount.
0 if the currency symbol follows the non-negative amount.

char n_cs_precedes
1 if the currency symbol precedes the negative amount.
0 if the currency symbol follows the negative amount.

char p_sep_by_space
1 if the currency symbol is separated from a non-negative amount by a space.
0 if not.

char n_sep_by_space
1 if the currency symbol is separated from a negative amount by a space.
0 if not.

char p_sign_posn
Position of the *positive_sign* for a non-negative amount.

char n_sign_posn
Position of the *negative_sign* for a negative amount.

The `char` elements of *grouping* and *mon_grouping* define the number of digits for the groups to the left of the decimal point, beginning with the first group to the left of the decimal point (e.g. thousands). The entries are interpreted as follows:

CHAR_MAX Corresponds to the highest EBCDIC value (255) and causes no further grouping to be carried out.

0 The null byte causes the entry of the preceding char element to apply to the grouping of all remaining digits.

Others The integer value applies to the number of digits in the current group. The next char element defines the number of digits in the next group.

The values of *p_sign_posn* and *n_sign_posn* are interpreted as follows:

0 Amount and *currency_symbol* are enclosed in parentheses.

1 The sign precedes the amount and *currency_symbol*.

2 The sign comes after the amount and *currency_symbol*.

3 The sign immediately precedes *currency_symbol*.

4 The sign comes immediately after *currency_symbol*.

Notes The available locales are described in chapter “Locale” on page 93.

The components of the supplied structure must not be explicitly overwritten by the user. New values for the structure can be supplied only by calling `localeconv`.

In the current locale, no values can be defined for various structure components. This is indicated for components of type `char` * by a pointer to "", and for components of type `char` by the value CHAR_MAX (value 255).

See also setlocale

localtime, localtime64 - Date and current time as a structure

Definition #include <time.h>

struct tm *localtime(const time_t *sec_p);
struct tm *localtime64(const time64_t *sec_p);

`localtime` and `localtime64` interpret the time specification to which *sec_p* points as the number of seconds which have passed since the reference date (epoch). The functions calculate the date and time from this and store the result in a structure of the type `tm`. Negative values are interpreted as seconds before the reference date. The earliest displayable date is 01/01/1900 00:00:00 local time.

With `localtime` the reference date depends on the use of the TIMESHIFT bind option (see section “Time functions” on page 40):

– without TIMESHIFT bind option (default): 1/1/1950 00:00:00.
– with TIMESHIFT bind option: 1/1/1970 00:00:00.

With `localtime64` the reference date is always 1/1/1970 00:00:00

The latest date which can be displayed with `localtime` is 01/19/2018 03:14:07 (without TIMESHIFT bind option) or 01/19/2038 03:14:07 (with TIMESHIFT bind option).

Irrespective of the use of the TIMESHIFT bind option, `localtime64` can display dates up to 3/18/4317 02:44:48.

Return val. Pointer to the calculated structure. `localtime` and `localtime64` store the result in a structure declared in <time.h> as follows:

```
struct tm
{
  int   tm_sec;        /* seconds (0-59) */
  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 from the start of the 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 flag */
};
```

NULL In the event of an error

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` and `localtime64` map all dates before 1/1/1900 01:00:00 to 1/1/1900 01:00:00.

Example

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

struct tm *t;
time_t clk;
char *s;

int main(void)
{
  clk = time((time_t *)0);
  t = localtime(&clk);
  printf("Year: %d\n", t->tm_year + 1900);
  printf("Time in hours: %d\n", t->tm_hour);
  printf("Day of the year: %d\n", t->tm_yday);
  s = asctime(t);
  printf("%s", s);
  return 0;
}
```

See also

asctime, ctime, ctime64, gmtime, gmtime64, time, time64

log - Natural logarithm

Definition #include <math.h>

double log(double x);

`log` calculates the natural logarithm of the positive floating-point number x to the base `e`.

Return val.		
	ln(x)	for positive x.
	-HUGE_VAL	if x is less than or equal to 0. In addition, `errno` is set to EDOM (domain error).
	-HUGE_VAL	if x is equal to 0. In addition, `errno` is set to ERANGE.

Example

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

int main(void)
{
  double x;
  printf("Example of log(x): Please enter x\n");
  if(scanf("%lf", &x) == 1)
  printf("x = %g log(x) = %g\n", x, log(x));
  return 0;
}
```

See also log10, exp

log10 - Logarithm to the base 10

Definition

#include <math.h>

double log10(double x);

`log10` calculates the logarithm of the positive floating-point number *x* to the base 10.

Return val.

lg(x)	for positive *x*.
-HUGE_VAL	if *x* is less than 0. In addition, `errno` is set to EDOM (domain error).
-HUGE_VAL	if *x* is equal to 0. In addition, `errno` is set to ERANGE.

Example

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

int main(void)
{
  double x;
  printf("Example of log10(x): Please enter x\n");
  if(scanf("%lf", &x) == 1)
  printf("x = %g log10(x)= %g\n", x, log10(x));
  return 0;
}
```

See also log, exp

longjmp - Non-local jump

Definition

#include <setjmp.h>

void longjmp(jmp_buf env, int value);

`longjmp` can only be used in association with the `setjmp` function. This is because a `longjmp` call causes the program to branch to a position previously “marked” with `setjmp`. In contrast to `goto` jumps, which are only admissible within the same function (i.e. locally), `longjmp` and `setjmp` enable the transfer of control from any given function to some other active function (non-local jump).

`setjmp` stores the current program status (address in the C runtime stack, program counter, register contents) in a variable of type `jmp_buf` (defined in <setjmp.h>). `longjmp` restores the program status stored by `setjmp`, and the program is then continued with the statement immediately following the `setjmp` call.

Parameters

jmp_buf env
Field in which `setjmp` has stored its values. The type `jmp_buf` is defined in <setjmp.h>.

int value
Integer interpreted as the return value of the `setjmp` call when program execution is resumed. If *value* is equal to 0, `setjmp` returns a value of 1; 0 would imply that control was transferred "normally" at the position after the `setjmp` call, i.e. that no branch was made with `longjmp` (see `setjmp` for further information).

Notes

The behavior is undefined if `longjmp` is called with an *env* argument that was not previously given a value by means of a `setjmp` call.

The function containing the `setjmp` call with the *env* variable must still be active when `longjmp` is activated with the same variable, i.e. this function should not have been terminated in the meantime (e.g. with `exit` or `return`).

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 have the same values they would have received following a `goto` call:
Global variables have the values that they had at the time of the `longjmp` call.
Register variables and other local variables are undefined, i.e. they should be checked and re-initialized, if required.

Example
Text I/O in an interactive text editor represents a typical use for `longjmp` and `setjmp`. When the program is interrupted during input or output as a result of an externally originating signal (e.g. when the K2 key is pressed after "please acknowledge" or in response to an input prompt), text I/O is terminated. Otherwise, the text editor continues with I/O operations.
The following program shows how this can be implemented with `setjmp` and `longjmp` (only illustrates signal handling - not an editor!):

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>

FILE *fp;
jmp_buf env;

void intr(int sig)
{
   printf("\n   ***** You don't want the text? ****** \n");
   longjmp(env,0);
}

int main(void)
{
   int c; char reply;

   setjmp(env);
   signal(SIGINT,intr);
   printf("Text output? (y•n):\n");
   scanf("%1s",&reply);             /* Interruption possible with K2 */
   if(reply == 'y')
     {
      fp = fopen("text","r");        /* File text must exist */
      while((c=getc(fp)) != EOF)
            putc((char)c,stdout);    /* Interruption of text output possible
                                     /* with K2 after "please acknowledge" */
     }
   else printf("No text output\n");
   return 0;
}
```

See also
setjmp, signal

lrint, lrintf, lrintl - Round off to nearest whole number

Definition

#include <math.h>

long int lrint(double x);

long int lrintf (float x);

long int lrintl (long double x);

Each of the functions returns the whole number nearest to x, represented as a number of type `long int`.

The return value is rounded off in accordance with the rounding mode currently set for the system. If the rounding mode is ‘round-to-nearest’ and if the difference between x and the rounded result is exactly 0.5, the nearest even number is returned.

If the rounding mode currently set rounds off in the direction of positive infinity, then `lrint` is equivalent to `ceil`. If the defined rounding mode rounds off in the direction of negative infinity, then `lrint` is equivalent to `floor`.
In this version, the rounding mode is preset in the direction of positive infinity.

Return val.

integer	represented as a number of type `long int` if successful.
undefined	in the event of an overflow or underflow, `errno` is set to ERANGE to indicate the error.

Note

In this version, the rounding mode is preset in the direction of positive infinity.

See also

abs, ceil, floor, llrint, llround, lround, rint, round

lround, lroundf, lroundl - Round off to nearest whole number

Definition #include <math.h>

long int lround(double x);

long int lroundf (float x);

long int lroundl (long double x);

Each of the functions returns the whole number nearest to *x*, represented as a number of type `long int`.

The return value is independent of the defined rounding mode. If the difference between *x* and the rounded result is exactly 0.5, the larger whole number is returned.

Return val.	integer	represented as a number of type `long long int` if successful.
	undefined	in the event of an overflow or underflow, `errno` is set to ERANGE to indicate the error.

See also abs, ceil, floor, llrint, llround, lrint, rint, round

lseek, lseek64 - Position read/write pointer (elementary)

Definition #include <stdio.h>

off__t lseek(int fd, off_t offset, int loc);
off64_t lseek64(int fd, off64_t offset, int loc);

`lseek` and `lseek64` position the read/write pointer for the file with file descriptor *fd* according to the specifications in *offset* and *loc*. It is thus possible for you to process a file non-sequentially. The return value from `lseek` and `lseek64` is the current position in the file.

To process files > 2 GB, proceed as follows:

– If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `lseek`. `lseek64` is then used implicitly with the appropriate parameters.

– Otherwise, you have to call `lseek64`.

There is no functional difference between `lseek` and `lseek64`, except that the offset type `off64_t` and the return type `off64_t` are used for `lseek64`.

Text files (SAM, ISAM) can be absolutely positioned 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 files are always processed as text files with elementary text functions.

Parameters int fd

File descriptor of the file whose read/write pointer is to be positioned.

off_t / off64_t offset, int loc

Since the meaning, combination options, and effects of these parameters differ for text and binary files, they are individually described in the following.

Text files (SAM, ISAM)

Possible parameter values:

offset	0L or value determined by a previous `tell`/`lseek` call. 0LL or value determined by a previous seek64 call.
offset (64-bit interface)	0LL or value determined by a previous `ftell`/`ftell64` call.
loc	SEEK_SET (beginning of file) SEEK_CUR (current position) SEEK_END (end of file)

Meaningful combinations and their effects:

offset	loc	Effect
tell/lseek value or lseek64 value	SEEK_SET	Position to the location marked by `tell` or `lseek/lseek64`.
0L or 0LL	SEEK_SET	Position to the beginning of the file.
0L or 0LL	SEEK_CUR	Query current position without positioning.
0L or 0LL	SEEK_END	Position to the end of the file.

Binary files (PAM, INCORE)

Possible parameter values:

offset Number of bytes by which the current read/write pointer is to be shifted. This number may be a positive number:
forward positioning toward end of file negative number:
backward positioning toward beginning of file
OL: absolute positioning to beginning or end of file.

ort For absolute positioning to the beginning or end of the file, the point to which the read/write pointer is to be shifted.
For relative positioning, the point from which the read/write pointer is to be shifted by *offset* bytes:
SEEK_SET (beginning of file)
SEEK_CUR (current position)
SEEK_END (end of file)

Meaningful combinations and their effects:

offset	loc	Effects
0L or 0LL	SEEK_SET	Position to the beginning of the file.
0L or 0LL	SEEK_CUR	Query current position without positioning.
0L or 0LL	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 or lseek64 value	SEEK_SET	Position to the location marked by `tell` or `lseek/lseek64`.

Return val. The position in the file if successful, i.e.

for binary files, the number of bytes that offsets the read/write pointer from the beginning of the file;

for text files, the absolute position of the read/write pointer.

-1 if an error occurs.
In addition, the corresponding error information is stored in the `errno` variable:
EBADF: Invalid file descriptor
ESPIPE: Invalid positioning
EINVAL: Invalid argument.
EMDS: For binary file opened for reading only, positioned after the end of the file.

Notes The `lseek(fd, 0L, SEEK_CUR)` and `tell(fd)` calls are equivalent, i.e. they both call the current position in the file without positioning it.

If new records are written to a text file (opened for creation or in append mode) and an `lseek`/`lseek64` call is issued, any residual data is first written from the internal C buffer 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`/`lseek64` does not cause a change of line (or change of record), i.e. the data is not automatically terminated with a newline character when writing from the buffer. Subsequent data lengthens the record in the file. When an ISAM file is read, therefore, only those newline characters explicitly written by the program are read in.

If you position past the end of file in the case of a binary file opened for writing, a “gap” appears between the last physically stored data and the newly written data. Reading from this gap returns binary zeros.
If you position past the end of a binary file opened for reading only, an error occurs (EMDS).

System files (SYSDTA, SYSLST, SYSOUT) cannot be positioned.

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`:

1. SAM file

Record length	≤ 2048 byte
Number of records/block	≤ 256
Number of blocks	≤ 2048

2. ISAM file

   Record length ≤ 32 Kbytes

   Number of records ≤ 32 K

Example The following program reads the file passed as the first argument in the call from position 10 onwards and appends its contents to the end of another file if a second argument is specified. Otherwise, it writes to the standard output (only works with binary files, i.e. with PAM files in this case):

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

int fd1, fd2;
long result;
char c;

int main(int argc, char *argv[])
{
  if((fd1 = open (argv[1],0)) < 0) exit(1);
  if(argc < 3)
          fd2 = 1;
  else
          fd2 = open(argv[2], 1);

  result = lseek(fd1, 10L, SEEK_SET);
  printf("current position in file1 : %ld\n", tell(fd1));

  /* Other possible position queries:
  printf("current position in file1: %ld\n, result);
  printf("current position in file1: %ld\n, lseek(fd1, 0L, SEEK_CUR)); */

  while(read(fd1, &c, 1) > 0)
          write(fd2, &c, 1);
  close(fd1);
  close(fd2);
}
```

See also tell, fseek, fseek64, ftell, ftell64

malloc - Reserve memory space

Definition #include <stdlib.h>

void *malloc(size_t n);

`malloc` allocates contiguous memory space of *n* bytes at execution time.

`malloc` is part of a C-specific memory management package which internally administers memory areas that are requested and subsequently freed. Attempts are made to satisfy new requests by first using areas that are already being managed and only then by the operating system (cf. `garbcoll` function).

Return val.

Pointer to the new memory area
: provided `malloc` was able to allocate new memory space. This pointer may be used for any data type.

NULL pointer
: if `malloc` was not able to provide the memory space, e.g. because the memory space still available does not suffice for the request or because an error occurred.

Notes

The new data area begins on a double word boundary.

The actual length of the data area amounts to:
the requested length *n* + 8 bytes for internal administrative data. If necessary, this sum is rounded up to the next power of 2.

If `malloc` does not find enough memory space in the list of free blocks, the `memalloc` function is internally called in order to obtain more memory space from the system.

You should use the `sizeof` function to ensure that you are requesting sufficient space for a variable.

A serious disruption in working memory may be expected if the length of the memory area provided is exceeded when writing.

If *n* has the value 0, `malloc` returns an unambiguous address which can also be transferred to `free`.

Example 1

The following program fragment requests memory space for 30 integer elements.

```
#include <stdlib.h>

int *int_array;
     .
     .
     .
int_array = (int *)malloc(30 * sizeof(int));
```

Example 2 Dynamic reservation of memory space for data on second-hand cars:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 20;

struct car {
       char   *type;
       int    age;
       long   kilometers;
       char   inspect[6];
       int    cond;
       int    price;
       struct car *n;
       } *list;

int main(void)
{
  int mark;
  if((list = (struct car *)malloc(sizeof(*list))) == NULL)
    {
      printf("Memory space exhausted\n");
      exit(1);
    }
/* N.B. !! The preceding malloc call only provided space for a pointer     */
/* (4 bytes) for the member type. Space for the type identifier must still */
/* be provided.                                                            */

  if((list->type = (char *)calloc(1,20)) == NULL)
      exit(1);             /* error */

                           /* Input used car */
  scanf("%20s %d", list->type, &list->age);
  scanf("%d %6s %d %d", &list->kilometers, list->inspect, &list->cond,
    &list->price);
  list->n = NULL;

                           /* print input values */
  printf("%s\n%d\n", list->type, list->age);
  printf("%d\n%.6s\n%d\n%d", list->kilometers, list->inspect,
   list->cond, list->price);

                           /* free memory space */
  free(list);
  return 0;
}
```

See also calloc, realloc, free, garbcoll, memalloc, memfree

mblen - Determine number of bytes of a multibyte character

Definition #include <stdlib.h>

int mblen(const char *s, size_t n);

`mblen` returns the number of bytes of a multibyte character to which *s* points. A maximum of *n* bytes in *s* are evaluated.

Return val.	-1	if *n* = 0.
	0	if *s* is a NULL pointer or points to a null byte (\0).
	1	otherwise.

Note In this implementation, there are no characters that consist of several bytes. Multibyte characters always have a length of 1.

See also mbstowcs, mbtowc, wcstombs, wctomb

mbrlen - Determine remaining length of a multibyte character

Definition #include <wchar.h>

size_t mbrlen(const char *s, size_t n, mbstate_t *ps);

`mbrlen` determines the number of bytes as of position **s* that are needed to complete a multibyte character. A maximum of *n* bytes are examined.

`mbrlen` is equivalent to the call
`mbrtowc(NULL, s, n, ps!= NULL ? ps: internal)`
where *internal* is the `mbstate_t` object for the `mbrlen` function.

Description: see `mbrtowc`.

mbrtowc - Complete multibyte character and convert to wide character

Definition

#include <wchar.h>

size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);

If *s* is not a null pointer, the `mbrtowc` function inspects at most *n* bytes beginning with the byte pointed to by **s* to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If `mbrtowc` can complete the multibyte character, it determines the value of the corresponding wide character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by **pwc*.
If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.
If *s* is a null pointer, `mbrtowc` is equivalent to the call
`mbrtowc(NULL, "", 1, ps)`
In this case, the values of the parameters *pwc* and *n* are ignored.

Return val.

Depending on the value of the current conversion state, `mbrtowc` returns the first of the following that applies:

`0` if the next *n* or fewer bytes complete a valid multibyte character that corresponds to the null wide character.

Number of bytes needed to complete the multibyte character
if the next *n* or fewer bytes complete a valid multibyte character. The value stored is the wide character corresponding to that multibyte character.

`(size_t)-2` if the next *n* bytes contribute to an incomplete (but potentially valid) multibyte character. No value is stored.

`(size_t)-1` if an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro `EILSEQ` is stored in `errno`, and the conversion state is undefined.

Note

This version of the C runtime system only supports one-byte characters as wide character codes.

See also

mblen, mbtowc, wcstombs, wctomb

mbsinit - Test for initial conversion state

Definition

```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

If *ps* is not a null pointer, `mbsinit` determines whether whether the `mbstate_t` object pointed to by *ps* describes an initial conversion state.

Return val.

≠ 0	if *ps* is a null pointer or points to an object the describes an initial conversion state.
0	otherwise.

mbsrtowcs - Convert multibyte string to wide character string

Definition

<include <wchar.h>

size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);

`mbsrtowcs` converts a sequence of multibyte characters, beginning in the conversion state described by the object pointed to by **ps*, from the array indirectly pointed to by *src* into a sequence of corresponding wide characters. If *dst* is not a null pointer, the converted characters are stored into the array pointed to by *dst*. Each conversion takes place as if by a call to the `mbrtowc` function.

Conversion stops on encountering a terminating null character, which is also converted and stored in the array.

Conversion stops earlier in two cases:

– when a sequence of bytes is encountered that does not form a valid multibyte character or

– if *dst* is not a null pointer, when *len* codes have been stored into the array pointed to by *dst*.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned one of the following values:

– a null pointer if conversion stopped due to reaching a terminating null character

– the address just past the last multibyte character converted (if any).

If *dst* is not a null pointer and if the conversion stopped due to reaching a terminating null character, the resulting state described is the initial conversion state.

Return val.

`(size_t)-1` if a conversion error occurs, i.e. a sequence of bytes that do not form a valid multibyte character are encountered. The value of the `EILSEQ` macro is stored in `errno`, and the conversion state is undefined.

Number of successfully converted multibyte chatacters otherwise. The terminating null character, if any, is not included in the count.

See also mblen, mbtowc, wcstombs, wctomb

mbstowcs - Convert multibyte string to wide character string

Definition

#include <stdlib.h>

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

`mbstowcs` converts a sequence of multibyte characters in the string pointed to by *s* to the corresponding wide characters (of type `wchar_t`) and writes a maximum of *n* wide characters to the area specified by *pwcs.*
Conversion continues until either *n* values have been converted or the null value is encountered (the null value is converted to the `wchar_t` value 0).

If *pwcs* is a null pointer, `mbstowcs` returns the length needed to convert the entire string (regardless of the value of *n*), but does not store any values.

If an invalid character is encountered, `mbstowcs` returns the value (`size_t`)-1.

The wide characters stored by `mbstowcs` in the *pwcs* area correspond to the values of the individual bytes in string *s*.

Return val.

Number of wide characters 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 resulting 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. Multibyte characters and wide characters always have a length of 1 byte.
The shift state of the multibyte character is ignored.

See also

mblen, mbtowc, wcstombs, wctomb

mbtowc - Convert multibyte character to wide character

Definition

```
#include <stdlib.h>

int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

`mbtowc` converts a multibyte character in *s* to the corresponding wide character (type `wchar_t`) and stores this value in the area *pwc*. A maximum of *n* bytes in *s* are evaluated.

The wide character 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
- *n* = 0.

Return val.

-1	if *n* = 0.
0	if *s* is a NULL pointer or points to a null byte.
1	otherwise.

Note

This version of the C runtime system only supports one-byte characters as wide character codes. Multibyte characters and wide characters always have a length of 1 byte.

See also

mblen, mbstowcs, wcstombs, wctomb

memalloc - Reserve memory space

Definition

```
#include <stdlib.h>

void *memalloc(size_t n);
```

`memalloc` allocates contiguous memory space of *n* bytes at execution time.

`memalloc` passes the request for memory space directly to the appropriate operating system call. This function is particularly suitable for memory areas with a size of more than 2 Kbytes (also see `memfree`).

Return val.

Pointer to the new memory area
: provided `memalloc` was able to allocate new memory space. This pointer may be used for any data type.

NULL pointer
: if `memalloc` was not able to provide the memory space, e.g. because the memory space still available does not suffice for the request.

Notes

The new data area begins on a doubleword boundary.

The requested length *n* is rounded up to the next multiple of 2 Kbytes.

A serious disruption in working memory may be expected if the length of the memory area provided is exceeded when writing.

The memory area requested with `memalloc` can be released again by using `memfree`.

See also

memfree

memchr - Search for a character in memory area

Definition #include <string.h>

void *memchr(const void *s, int c, size_t n);

`memchr` searches for the first occurrence of the character *c* in the first *n* bytes of the memory area to which *s* points.

Return val. Pointer to the position of *c* in area *s*
if successful.

NULL pointer if *c* is not contained in the specified area.

Notes The function is suitable for processing character arrays containing the null byte (\0), since `memchr` does not interpret the null byte as the 'end of text'.

The following two prototypes of the `memchr` function are applicable to C++:

```
const void *memchr(const void *s, int c, size_t n);
      void *memchr(      void *s, int c, size_t n);
```

See also memcmp, memcpy, memset

memcmp - Compare memory areas

Definition

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

`memcmp` compares the contents of the first *n* bytes of the memory areas to which *s1* and *s2* point.

Return val.

< 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*.

Note This function is suitable for processing character arrays containing the null byte (\0), since `memcmp` does not interpret the null byte as the 'end of text'.

See also memchr, memcpy, memset

memcpy - Copy memory area

Definition #include <string.h>

void *memcpy(void *s1, const void *s2, size_t n);

`memcpy` copies the first *n* bytes of the memory area to which *s2* points into the memory area pointed to by *s1*.

Return val. Pointer to the memory area *s1*.

Notes This function is suitable for processing character arrays containing the null byte (\0), since `memcpy` does not interpret the null byte as the 'end of text'.

`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 memchr, memcmp, memset

memfree - Free memory area

Definition #include <stdlib.h>

void memfree(const void *p, size_t n);

`memfree` releases *n* bytes of the memory area to which *p* points. *p* must be the result of a preceding `memalloc` call.

`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 Kbytes.

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 a serious disruption in working memory!

See also memalloc

memmove - Copy memory area

Definition #include <string.h>

void *memmove(void *s1, const void *s2, size_t n);

`memmove` copies the first *n* bytes of the memory area to which *s2* points to the memory area to which *s1* points.
`memmove` first copies the *n* bytes to a temporary array that does not overlap memory areas *s1* and *s2* and only then to memory area *s1*.

Return val. Pointer to memory area *s1*.

Notes This function is suitable for processing character arrays containing the null byte (\0), since `memmove` does not interpret the null byte as the ’end of text’.

In contrast to `memcpy`, `memmove` also works with memory areas that overlap.

See also memcpy

memset - Initialize memory area

Definition #include <string.h>

void *memset(void *s, int c, size_t n);

`memset` copies the value of character *c* to the first *n* bytes of the memory area to which *s* points.

Return val. Pointer to the memory area *s*.

Notes This function is suitable for processing character arrays containing the null byte (\0), since `memset` does not interpret the null byte as the ’end of text’.

`memset` does not check whether data in result area *s* is in danger of being overwritten.

See also memchr, memcmp, memcpy

mktemp - Generate a unique temporary file name

Definition

```
#include <stdio.h>

char *mktemp(char *model);
```

`mktemp` generates unique names for temporary SAM files from a string *model*, which must contain at least 8 characters. The name is composed from the characters in *model* as follows:

- The first three characters are replaced by "#T.".
- The fourth character is replaced by a character which 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 contents of *model* were "XXXX.ABC.XXXX" and the TSN of the running task were 6082, the temporary name generated by `mktemp` at the first call would be:

#T.A.ABC.6082

Return val.

Pointer to the result string containing the new name
if successful.

NULL pointer if an error occurred, e.g. because *model* contains less than 8 characters or because the maximum permissible number (36) of `mktemp` calls has been exceeded (see notes for further information).

Notes

Since the letters A-Z and the digits 0-9 are used for the formation of a unique name, the number of `mktemp` calls is limited to 36 per program run.

Temporary files are automatically deleted on termination of a task (LOGOFF). However, the files are retained if the standard prefix (#) for temporary files was changed during system generation.

Example The following program generates three unique temporary file names and opens the files for writing and reading.

```
#include <stdio.h>
FILE *fp1, *fp2, *fp3;
char s[] = "XXXX.temp.XXXX";

int main(void)
{
   mktemp(s);
   fp1 = fopen(s,"w+r");
   printf("%s\n",s);              /* generated name: #T.A.TEMP.6082 */

   mktemp(s);
   fp2 = fopen (s,"w+r");
   printf("%s\n",s);              /* generated name: #T.B.TEMP.6082 */

   mktemp(s);
   fp3 = fopen (s,"w+r");
   printf("%s\n",s);              /* generated name: #T.C.TEMP.6082 */
   return 0;
}
```

mktime, mktime64 - Convert date and time (calendar function)

Definition #include <time.h>

time_t mktime(struct tm *tm_p);
time_t mktime64(struct tm *tm_p);

`mktime` and `mktime64` convert the date and time which the user specifies in a structure of the type `tm` to a time specification which is displayed as the number of seconds which have passed since the reference date (epoch). Dates before the reference date are returned as negative values. The earliest displayable date is 01/01/1900 00:00:00 local time.

In the calculation `mktime` and `mktime64` complement the `tm` structure with the values for weekday (0-6) and day since the start of the year (0-365) and adjust the values of the other components to the ranges of values provided by default (see also the parameter description).

With `mktime` the reference date depends on the use of the TIMESHIFT bind option (see section “Time functions” on page 40):

- without TIMESHIFT bind option (default): 1/1/1950 00:00:00.
- with TIMESHIFT bind option: 1/1/1970 00:00:00.

With `mktime64` the reference date is always 1/1/1970 00:00:00.

If the calendar time cannot be displayed because of missing specifications in the input parameter, `mktime/mktime64` supplies the return value -1. In the case of `mktime` this also applies for dates after 01/19/2018 03:14:07 (without TIMESHIFT bind option) or after 01/19/2038 03:14:07 (with TIMESHIFT bind option).

Irrespective of the use of the TIMESHIFT bind option, `mktime64` can display dates up to 3/18/4317 02:44:48.

Parameters struct tm *tm_p

Pointer to a structure of type `tm` which is supplied by the user with the date and time and is then updated by `mktime/mktime64`. The default value ranges are given in parentheses.

```
int   tm_sec;          seconds (0-59)
int   tm_min;          minutes (0-59)
int   tm_hour;         hours (0-23)
int   tm_mday;         day of the month (1-31)
int   tm_mon;          months since the start of the 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 flag:
                       0    daylight saving time is not in effect
                       >0   daylight saving time is in effect
                       <0   information is not available
```

1. User-specified date and time entries

   The components `tm_wday` and `tm_yday` need not be entered since `mktime` ignores these in calculating `time_t` and then supplies them itself with suitable values.

   All other components must have a value. These values are not limited to the above-mentioned default value ranges, i.e. they may be greater or smaller.

   Examples:

   -1 in `tm_hour` means 1 hour before midnight,
   0 in `tm_mday` means the last day of the previous month,
   -2 in `tm_mon` means 2 months before January in year `tm_year`.

2. Structure updating by `mktime`

   The components `tm_wday` and `tm_yday` are set to the values that match the user specifications.
   The other components are assigned so that their values correspond to the above-mentioned default ranges.
   The value of `tm_mday` is not assigned unless `tm_mon` and `tm_year` have been defined.

Return val.	Integer>0	for local times after the reference date (epoch): the number of seconds which have elapsed since then (positive value).
	Integer<0	for local times prior to the reference date (epoch): the number of seconds which have elapsed up to that point (negative value)
	(time_t) - 1	if the time cannot be represented.

See also asctime, ctime, ctime64, difftime, difftime64, ftime, ftime64, gmtime, gmtime64, localtime, localtime64, time, time64

modf - Split a number into its integer and fractional parts

Definition #include <math.h>

double modf(double n, double *i_p);

`modf` resolves a floating-point number *n* into its integral and fractional parts. The result of `modf` is the signed fraction and the integral part, the latter being returned indirectly via a result parameter *i_p*.

Return val. Fractional part of *n* with sign.

Note Note that the argument *i_p* must be a pointer!

Example The following program resolves the number -456.789 into its integral and fractional parts.

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

int main(void)
{
  double x, g;
  x = modf(-456.789, &g);
  printf("Fraction : %g\nIntegral part : %g\n", x, g);
  return 0;
}
```

See also frexp, ldexp

offsetof - Offset of a structure component from the start of the structure

Definition
#include <stddef.h>

size_t offsetof(type,component);

`offsetof` returns the offset in bytes between the structure component *component* and the start of the structure (label) of type *type*.
`offsetof` is a macro.

Return val. Offset of the structure component from the start of the structure in bytes.

Note If the specified structure component is a bit field, the behavior is undefined.

Example

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

struct S1 {
   char c;
   int  i;
   double d;
};

int main(void)
{
  typedef struct S1 t_s1;

  printf("offsetof(struct S1, c) = %d\n", offsetof(struct S1, c) );
  printf("offsetof(struct S1, i) = %d\n", offsetof(struct S1, i) );
  printf("offsetof(struct S1, d) = %d\n", offsetof(struct S1, d) );
  printf("\n");

  printf("offsetof(t_s1, c) = %d\n", offsetof(t_s1, c) );
  printf("offsetof(t_s1, i) = %d\n", offsetof(t_s1, i) );
  printf("offsetof(t_s1, d) = %d\n", offsetof(t_s1, d) );
  printf("\n");
  return 0;
}
```

open, open64 - Open a file (elementary)

Definition #include <stdio.h>

int open(const char *f_name, int mode);
int open64(const char *f_name, int mode);

`open` and `open64` open the file *f_name* with an access mode that depends on the (octal) value of *mode*. `open` and `open64` return a valid file descriptor that is used later to identify the file in elementary access operations (`read`, `write`).

There is no functional difference between `open` and `open64`, except that `open64` sets the bit O_LARGEFILE implicitly in the file status flag. The `open64` function corresponds to the use of the `open` function where O_LARGEFILE is set to *oflag*.

To process files > 2 GB, proceed as follows:

– If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `open`. `open64` is then used implicitly with the appropriate parameters.

– Otherwise, you have to call `open64`.

Parameters const char *f_name

String specifying the name of the file to be opened. *f_name* may be:

– any valid BS2000 file name
– "link=*linkname*"
linkname identifies a BS2000 link name
– "(SYSDTA)", "(SYSOUT)", "(SYSLST)"
the appropriate system file
– "(SYSTERM)"
terminal Input/Output
– "(INCORE)"
temporary binary file that is only initialized in virtual memory.

int mode

Integer variable whose octal value specifies the desired access mode, namely:

0000	Open for reading. The file must already exist.
0001	Open for writing. The file must already exist. The previous contents are retained.
01001	Open for writing. If the file exists, the previous contents are deleted. If the file does not exist, it is created.
0002	Open for reading and writing. The file must already exist. The previous contents are retained.
01002	Open for reading and writing. If the file exists, the previous contents are deleted. If the file does not exist, it is created.

0003	Open for writing and reading. If the file exists, the previous contents are deleted. If the file does not exist, it is created.
0401	Open for appending to the end of the file. The file must already exist. The file is positioned to end of file, i.e. the previous contents are preserved and the new text is appended to the end of the file.
0402	Open for appending to the end of the file and for reading. The file must already exist. The 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.

Return val.

File descriptor positive number that is used later to identify the file in elementary access operations (`read`, `write`).

-1 if the file could not be opened, e.g. due to the absence of access authorization, entry of an invalid file name or link name etc.

Notes

The BS2000 file name or link name can be written in both uppercase and lowercase. It is automatically converted to uppercase.

If a non-existent file is created, the following applies by default:
With KR functionality (applies to C/C++ versions prior to V3.0 only), a SAM file with variable record length and standard block length is created; with ANSI functionality, an ISAM file with variable record length and standard block length is created.
When opened with `open` or `open64`, SAM files are always text files.

By using a link name the following file attributes can be changed with the ADD-FILE-LINK command: access method, record length, record format, block length and block format. See also section “System files (SYSDTA, SYSOUT, SYSLST)” on page 70.

Whenever the old contents of an already existing file are deleted (0003, 01001), the catalog attributes of this file are preserved.

Position of the read/write pointer in append mode
If you explicitly position the read/write pointer away from the end of a file (`lseek`/`lseek64`) that was opened in append mode (0401, 0402), the way it is handled depends on whether you are using KR or ANSI functionality.
KR functionality (applies to C/C++ versions prior to V3.0 only): The current read/write pointer is ignored only when writing with the elementary function `write` and automatically positioned to the end of the file.
ANSI functionality: The current read/write pointer is ignored for all write functions and automatically positioned to the end of the file.

An attempt to open a non-existent file in the read (0000, 0002), update (0001), or append (0401, 0402) mode results in an error.

You may open a file for different access modes simultaneously, provided these modes are compatible with one another within the BS2000 data management system.

(INCORE) files can be only 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`/`lseek64` 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>.

Example The following program opens the file *joke* twice (for reading) and processes it with different file descriptors (*fd1*, *fd2*).

```
#include <stdio.h>

int fd1,fd2;
char c;
int n;

int main(void)
{
               /* open file "joke" for the
                  first time for reading */
   if((fd1=open("joke",0)) == -1)
               /* error in the first open */
      printf("Error1\n");

               /* open file "joke" for the
                  second time for reading */
   if((fd2=open("joke",0)) == -1)
               /* error in the second open */
      printf("Error2\n");

               /* reading is performed via fd1 until the first 'a' */
   while((n=read(fd1,&c,1)) > 0 && (c != 'a'))
               /* output the read in character on
                  standard output */
          write(1,&c,n);
```

```
/* reading is now performed via fd2
                  from the beginning of the file!!
                  to the end of the file */
while((n=read(fd2,&c,1)) > 0)
               /* output the read in character on
                  standard output */
          write(1,&c,n);

               /* reading continues via fd1 following
                  the first 'a', until the end of the file */
   while((n=read(fd1,&c,1)) > 0)
               /* output the read in character on
                  standard output */
          write(1,&c,n);
   return 0;
}
```

See also creat, creat64, fdopen, read, write, close

perror - Output error message

Definition #include <stdio.h>

void *perror(const char *s);

`perror` writes to the standard error output an error message corresponding to the error code in the internal C variable `errno`. *s*, a string passed as an argument, is output first, followed by a colon and the short error text from <errno.h>; the message is terminated with a newline character:

s : <short error message>\n

The following error information is provided:

– a text which briefly describes the error,
– the name of the function with the error, and
– the DMS error code (hexadecimal), if any.

Notes

`errno` error texts may contain the appropriate DMS error codes as supplementary information, e.g. in the case of I/O errors or when system commands are executed.
You will find a list of all `errno` error codes and error texts in the include file <errno.h>.

If a NULL pointer is passed as argument *s*, only the `errno` error text is output.

The contents of the area in which the error code and the error text are stored are not explicitly deleted. This means that the previous contents are retained until they are overwritten with appropriate information when a fresh error occurs. Consequently, `perror` calls are only useful immediately after a function has provided an error return value.

With KR functionality (applies to C/C++ versions prior to V3.0 only) a value of type `char` * is returned. It contains a pointer to an internal C buffer with the error message. The contents are overwritten for each new call to `perror`.

Example

The following program opens the file *fnam* for reading. If the file does not exist, the following error message is printed on the standard output:

```
Program fopen: dataset not found (cmd: OPEN), errorcode=DD33
```

DD33 is the DMS error code.

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  if((fp = fopen("fnam", "r")) == NULL)
    perror("Program fopen");
  return 0;
}
```

pow - General exponential function

Definition #include <math.h>

double pow(double x, double y);

`pow` calculates x^y.
If *x* is 0, *y* must be positive.
If *x* is negative, *y* must be an integer.

Return val.

x^y	if *x*, *y* and the result are in the permissible range of floating-point numbers.
+/-HUGE_VAL	in the event of an overflow (sign depending on the sign for *x*) . In addition, `errno` is set to ERANGE (result too large).
-HUGE_VAL	if *x* is equal to 0 and *y* is less than 0. In addition, `errno` is set to EDOM.
1.0	if *x* and *y* are equal to 0.
undefined	if *x* is less than 0 and *y* is not an integer. In addition, `errno` is set to EDOM.

Example The following program calculates x^y for the input arguments *x* and *y*.

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

int main(void)
{
  double x, y;
  scanf("%lf %lf", &x, &y);
  printf("%g**%g : %g\n", x, y, pow(x,y));
  return 0;
}
```

See also exp, hypot, log, log10, sinh, sqrt

printf - Formatted output on standard output

Definition #include <stdio.h>

int printf(const char *format, argumentlist);

`printf` edits data (characters, strings, numerical values) for output according to the specifications in the string *format* and writes this data to the standard output `stdout`. Numeric values are converted from their internal representation into printable characters in the process.

Parameters The format string may contain the following specifications:

– Ordinary characters (`char`)

  These are output on a 1 : 1 basis.

– Control characters for white space, beginning with a backslash (\).

– Format statements, which begin with the percent sign (%).

  Data passed in an argument list is formatted and converted in accordance with the specifications in the format statements. One format statement is required per argument, with the first format statement corresponding to the first argument and so forth.

Ordinary characters

All characters which are not elements of format statements and do not represent special control characters (beginning with a backslash) are output unchanged. If the percent character (%) is to be written, it must be specified twice in succession (%%).

Control characters for white space

\n	line feed
\t	tab
\f	form feed
\v	vertical tab
\b	bachspace
\r	carriage return

Further information on converting these control characters is given in section “White space” on page 65.

argumentlist
Variables or constants whose values are to be converted and formatted for output according to the information in the format statements.
If the number of format statements does not match the number of arguments the following applies:
If there are more arguments, the surplus arguments are ignored.
If there are fewer arguments, the results are undefined.

The format statement is described separately below for KR functionality and ANSI functionality.

Format statement (KR functionality, applies to C/C++ versions prior to V3.0 only)

Format statements may be structured as follows:

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

| |                        | |               |
| └───────────┬────────────┘ └───────┬───────┘
|             |                      |
|             |                      |
1)            2)                     3)
```

1. Every format statement must begin with a percent character (%).
2. General formatting characters, e.g. to control the output of signs, left or right justification, width of the output field, etc.
3. Characters which specify the actual conversion.

Meaning of the formatting characters:

- Left-justified alignment of the output field.
Default: right-justified alignment.

+ The result of a conversion with a sign is always output with a sign.
Default: only a negative sign (if present) is output.

0 Pad with zeros.
The output field is padded with zeros for all conversions.
Default: output field is padded with blanks.
Padding with zeros is also performed in the case of left-justified alignment (formatting character –).

n Minimum total field width (including decimal point). If more digits are required for the conversion of a number, this specification has no meaning. If the output is shorter than the specified field width it is padded with blanks or zeros to make up the field width (cf. formatting characters – and 0).

* The total field width (see *n*) is defined by an argument instead of in the format statement. The current value (integer) must immediately precede the argument to be converted or immediately precede the precision value (formatting character .*m*) in the argument list (separated by a comma).

.*m* Precision.
e, f, g conversion: precise number of digits after the decimal point (maximum 20).
Default: 6 digits.
s conversion: maximum number of characters to be output. Default: all characters up to the terminating null byte (\0).
In all other conversions the precision value is ignored.

.* The precision (see .*m*) is defined by an argument instead of in the format statement. The current value (integer) must immediately precede the argument to be converted in the argument list (separated by a comma).

Meaning of the conversion characters:

l	l before d, o, u, x:
Conversion of an argument of type `long int`
Identical to uppercase letters (D, O, U, X).

d, o, u, x
Representation of an integer (`int`) as a
signed decimal number (d),
unsigned octal number (o),
unsigned decimal number (u),
unsigned hexadecimal number (x).

f	Floating-point number (`float` or `double`) in the format
[–]ddd.ddd
The decimal point character is determined by the locale (category LC_NUMERIC).
The default setting is the period.
The number of digits after the decimal point depends on the precision specification in *.m*.
Default (no specification): 6 digits
If the precision is 0, the number is output without a decimal point.

e	Floating-point number (`float` or `double`) in the format [–]d.ddde{+|–}dd.
The decimal point character is determined by the locale (category LC_NUMERIC).
The default setting is the period.
The number of digits after the decimal point depends on the precision specification in *.m*.
Default (no specification): 6 digits
If the precision is 0, a decimal point with no digits after it is output.

g	Floating-point number (`float` or `double`) in the f or e format.
The number of digits after the decimal point depends on the precision specification in *.m*.
In each case, the representation chosen is the one that requires the least space while maintaining precision.

c	Format for the output of a single character (`char`). The character '\0' is ignored.

s	Format for the output of strings.
The string should be terminated with '\0'. `printf` writes as many characters of the string as is specified in *.m*.
Default (no specification): `printf` writes all characters up to '\0'.

%	Print a %, no conversion.

Format statement (ANSI functionality)

Format statements may be structured as follows:

```
                                     ⎧ [{h|l|ll}] {d|i|o|u|x|X} ⎫
                                     ⎪                          ⎪
                                     ⎪ {D|O|U|p}                ⎪
                                     ⎪                          ⎪
                       ⎧n⎫   ⎧.m⎫   ⎪ [L] {f|e|E|g|G}          ⎪
% [-][+][␣][#][0][{ }][{   }] {                           }
                       ⎩*⎭   ⎩.*⎭   ⎪ [l] {c|s}                ⎪
                                     ⎪                          ⎪
                                     ⎪ [{h|l|ll}] n             ⎪
                                     ⎪                          ⎪
                                     ⎩ %                        ⎭

|  |                          |  |                            |
|  └────────────┬─────────────┘  └─────────────┬──────────────┘
|               |                              |
|               |                              |
1)              2)                             3)
```

1. Every format statement must begin with a percent character (%).
2. General formatting characters, e.g. to control the output of signs, left or right justification, width of the output field, etc.
3. Characters which specify the actual conversion.

Meaning of the formatting characters:

- Left-justified alignment of the output field.
  Default: right-justified alignment.

+ The result of a conversion with a sign is always output with a sign.
  Default: only a negative sign (if present) is output.

␣ (blank)
  If the first character of a string to be converted with a sign is not a sign the result is prefixed by a blank.
  The formatting character ␣ is ignored if + is specified at the same time.

Conversion of the result to an alternative format.
  o conversion: precision is increased so that the first digit of the result is the digit 0.
  x or X conversion: the string 0x or 0X is prefixed to a result not equal to 0.
  e, E, f, g and G conversion: the result always contains a decimal point, even if there are no further digits (normally the result only contains a decimal point if it is followed by at least one digit). In addition, trailing zeros are not omitted for g or G conversion.
  The formatting character # has no effect in c, s, d, i or u conversions.

0 Pad with zeros.
The output field is padded with zeros for the conversion of integers (d, i, o, u, x, X) and floating-point numbers (e, E, f, g, G).
Default: the output field is padded with blanks.
0 is ignored if the formatting character – is specified or, in the case of the conversion of integers, a precision of *.m* is specified.
The formatting character 0 has no effect in c, p and s conversions.

n Minimum total field width (including decimal point). If more digits are required for the conversion of a number, this specification has no meaning. If the output is shorter than the specified field width it is padded with blanks or zeros to make up the field width (cf. formatting characters – and 0).

* The total field width (see *n*) is defined by an argument instead of in the format statement. The current value (integer) must immediately precede the argument to be converted or immediately precede the precision value (formatting character *.m*) in the argument list (separated by a comma).

.m Precision.
d, i, o, u, x or X conversion: minimum number of digits to be output. Default: 1.
e, E, f conversion: precise number of digits after the decimal point (maximum 20).
Default: 6 digits.
g or G conversion: maximum number of significant places.
s conversion: 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 in the format statement. The current value (integer) must immediately precede the argument to be converted in the argument list (separated by a comma).

Meaning of the conversion characters:

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 synonymous with uppercase 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 a
signed decimal number (d, i),
unsigned octal number (o),
unsigned decimal number (u),
unsigned hexadecimal number (x, X). For x the lowercase letters abcdef are used, for X the uppercase letters ABCDEF.
The precision value *.m* indicates the minimum number of digits to be output. If the value can be represented with fewer digits the result is padded with leading zeros. The default value is precision 1. If the precision is 0 and the value is 0, there is no output.

f Floating-point number (`float` or `double`) in the format [−]ddd.ddd
The decimal point character is determined by the locale (category LC_NUMERIC). The default setting is the period.
The number of digits after the decimal point depends on the precision specification in *.m*.
Default (no specification): 6 digits
If the precision is 0, the number is output without a decimal point.

e, E Floating-point number (`float` or `double`) in the format [−]d.ddde{+|−}dd.
The decimal point character is determined by the locale (category LC_NUMERIC). The default setting is the period.
For E conversion the exponent is preceded by the uppercase letter E.
The number of digits after the decimal point depends on the precision specification in *.m*.
Default (no specification): 6 digits
If the precision is 0, the number is output without a decimal point.

g, G Floating-point number (`float` or `double`) in the f or e format (or in the E format for G conversion).
The number of significant places depends on the precision value *.m*.
The e or E format is only used 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 character ’\0’ is ignored.

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

s Format for the output of strings.
The string should be terminated with '\0'. `printf` writes as many characters of the string as is specified in *.m*.
Default (no specification): `printf` writes all characters up to '\0'.

n There is no conversion or output of the argument. The argument is of type pointer to `int`. This integer variable is assigned the number of characters that `printf` has generated for output up to this time.

% Print a % character, no conversion.

Return val. Number of characters output
if successful.

Integer < 0 if an error occurs.

Notes `printf` rounds to the specified precision when converting floating-point numbers.

`printf` does not convert one data type to another. A value must be explicitly converted (e.g. with the `cast` operator) if it is not to be output in accordance with its type.

The data is not written immediately to the external file but is stored in an internal C buffer (see section "Buffering" on page 63).

Maximum number of characters to be output:
With KR functionality (applies to C/C++ versions prior to V3.0 only) a maximum of 1400 characters can be output per `printf` call,
with ANSI functionality a maximum of 1400 characters per conversion element (e.g. %s).

Attempts to output non-initialized variables or to output variables in a manner inconsistent with their data type can lead to undefined results.

The behavior is undefined if the percent sign (%) in a format statement is followed by an undefined formatting or conversion character.

`printf` works like `fprintf`, except that the data is written to the standard output and not to a file.

Example 1 Output of the date and time in the following form:

```
Thursday, February 14, 12:05 hours
```

The arguments *weekday* and *month* are pointers to strings terminated with '\0'.

```
printf("%s, %s %d, %02d:%02d hours\n", weekday, day, month, hrs, min);
```

Example 2 Output of the number pi to 5 decimal places.

```
printf("pi = %.5f\n", 4 * atan(1.0));
```

Example 3 The most common `printf` formats are self-explanatory in the way they are used in the other program examples. In the following table, you will find some additional format specifications listed along with their effects.
For clarity, the converted result is placed in > <.

Format specification	Argument()	Result
%.6s	"Konstanz"	> Konsta<
%10.5s	"Konstanz"	> Konst<
%-10.5s	"Konstanz"	> Konst <
%15.15s	"Konstanz"	> Konstanz<
%*.*s	20,7,"Konstanz"	> Konstan<
%-*.*s	15,10,"Konstanz"	> Konstanz <
%8d	721932	> 721932<
%-8d	721932	> 721932 <
%+d		> +d<
%-*.*f	3,2,27.31928	> 27.32<
%-0*.*f	1,12,19.84	> 19.840000000000<
%04.*f	12,10.60	> 10.600000000000<
%-0*.*g	1,12,19.84	> 19.84<
%e	1712.1961	> 1.712196e+03<
%.10e	1712.1961	> 1.7121961000e+03<
%10.10e	1712.1961	> 1.7121961000e+03<

See also fprintf, sprintf, snprintf, putc, putchar, puts, scanf, fscanf

putc - Write a character to a file

Definition #include <stdio.h>

int putc(int c, FILE *fp);

`putc` writes the character *c* to the file with file pointer *fp* at the current read/write position.

Return val. The written character *c*
if successful.

EOF otherwise.

Notes `putc` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

The characters are not written immediately to the external file but are stored in an internal C buffer (see section “Buffering” on page 63).

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” on page 65).

Example The following program reads characters from the standard input and writes them to the file *fnam*.

```
#include <stdio.h>

FILE *fp;
int c;

int main(void)
{
  fp = fopen("fnam","w");
  while((c=getchar()) != EOF)
        putc((char)c,fp);
  fclose(fp);
  return 0;
}
```

See also fputc, printf, putchar, fopen, fopen64, putwc

putchar - Write a character to the standard output

Definition #include <stdio.h>

```
int putchar(int c);
```

`putchar` writes the character *c* to the standard output.

Return val. The written character *c*
if successful.

EOF otherwise.

Notes `putchar` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

The characters are not written immediately to the external file but are stored in an internal C buffer (see section “Buffering” on page 63).

For further information on output to text files and on converting the control characters for white space (\n, \t, etc.) see section “White space” on page 65.

See also putc, fputc, putwchar

puts - Output a string to the standard output

Definition #include <stdio.h>

int puts(const char *s);

`puts` writes the string *s* to the standard output `stdout` and adds to it a terminating newline character.
s must be terminated with a null byte (\0)

Return val.

0	if successful.
EOF	otherwise.

Notes In contrast to `fputs`, `puts` automatically terminates output with a newline character. If the string to be output already contains a terminating newline (e.g. a record in SAM or ISAM files), an additional blank line will be inserted on output.

The terminating null byte of *s* is not output.

For further information on output to text files and on converting the control characters for white space (\n, \t, etc.) see section “White space” on page 65.

Example This example shows how `puts` and `fputs` differ in the way in which they terminate the output.

```
#include <stdio.h>
int main(void)
{
 FILE *fp;
 char s[BUFSIZ];
 fp=fopen("file","w");
 while(gets(s) != NULL)
      {
        fputs(s,fp);
        puts(s);
      }
 return 0;
}
```

If you look at *file* after this program has run, you will see that the strings from the input (`gets` deletes any existing newline) of `fputs` were written one following another and not by lines. In contrast, the output with `puts` is effected line by line, since a newline is automatically appended to every string that is read.

See also fputs, gets, fgets, putws, sprintf, snprintf

putw - Write a word at a time into a file

Definition #include <stdio.h>

int putw(int w, FILE *fp)

`putw` writes a machine word into the file with file pointer *fp* at the current read/write position.

Return val.

The written *w*	if successful.
EOF	otherwise.

Notes Since word length and the order of bytes are system-dependent, it is possible that files written with `putw` on a non-BS2000 operating system may not be readable with `getw` in BS2000.

Since `putw` does not explicitly indicate errors (-1 is a valid `integer` value), you should also use `ferror` to check whether an error occurred before or after the write operation.

The characters are not written immediately to the external file but are stored in an internal C buffer (see section “Buffering” on page 63).

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” on page 65).

Example The following program transfers the contents of the file *input* to the file *output*, one word at a time.

```
#include <stdio.h>
FILE *fp_in, *fp_out;  int w;
int main(void)
{
  fp_in = fopen("input","r");
  fp_out = fopen("output","w");
  while(!feof(fp_in) && !ferror(fp_in) && !ferror(fp_out))
       {
         w = getw(fp_in);
         putw(w,fp_out);
       }
  fclose(fp_in);  fclose(fp_out);
  return 0;
}
```

See also getw

putwc - Write wide character to a file

Definition
```
#include <wchar.h>
#include <stdio.h>

wint_t putwc(wchar_t wc, FILE *fp);
```

`putwc` is equivalent to the `fputwc` function, except for the following difference: when `putwc` is implemented as a macro, it can evaluate *fp* more than once, so the argument should never be an expression with side effects.
For this reason, it is better to use the `fputwc` function instead of `putwc`, especially in cases such as `putwc(`*wc, *f*`++)`.

Description: see `fputwc`.

putwchar - Write wide character to standard output

Definition
```
#include <wchar.h>

wint_t putwchar(wchar_t wc);
```

The function call `putwchar(`*wc*`)` is equivalent to `putwc(`*wc*, `stdout)`.
Description: see `fputwc`.

qsort - Sort a data field (quicksort)

Definition #include <stdlib.h>

void qsort(void *field, size_t n, size_t elsize,
    int (*comp)(const void *, const void *);

`qsort` sorts *n* elements of an array *field* using the quicksort algorithm. Each array element is *elsize* bytes in length.
In order to be able to sort the field, `qsort` requires a function *comp* (to be provided by the user), which compares two elements with each other.

Parameters void *field
Pointer to the first element of the array to be sorted.

size_t n
Number of elements to be sorted.

size_t elsize
Size of an element, in bytes.

int (*comp)(const void *, const void *)
Pointer to a function that compares two elements and returns a whole number as its result. This result is interpreted as follows:

< 0	argument1 is less than argument2
= 0	argument1 and argument2 are equal
> 0	argument1 is greater than argument2

The function has two parameters, i.e. two pointers to the type of the array elements.

The function may be defined something like this:

Example 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);
}
```

Example 2

```
/*compares two integer values */
int compare(const void *a, const void *b)
{
  return  ( *((const int *) a) - *((const int *) b) );
}
```

Note Array elements that are determined to be equal by the comparison function are retained in the same order.

Example The following program sorts a number field and outputs it on the standard output.

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

int comp   (const void *s, const void *t)
{
    return  ( *((const int *) s) - *((const int *) t) );
}

int main(void)
{
  int j;
  static int array[] = {4,7,2,1,54,9,2,3,1,23};

  qsort (array, 10, sizeof(int), comp);

  for (j=0; j<10; j++)
     printf("%d\n", array[j]);
  return 0;
}
```

See also bsearch

raise - Send signal to own program

Definition #include <signal.h>

int raise(int sig);

`raise` sends the signal *sig* to its own program.

`raise` can be used both to simulate STXIT events as well as to send STXIT-independent signals (self-defined or predefined by the C runtime system).

Parameters int sig

Signal to be sent to its own program. The symbolic constants listed in the following overview under “SIGNR” may be used for *sig*. These constants are defined in the include file <signal.h>.

SIGNR	**STXIT class**	**Meaning**
SIGHUP	ABEND	Disconnection of link to terminal
SIGINT	ESCPBRK	Interrupt from the terminal (K2)
SIGILL	PROCHK	Execution of an invalid instruction
SIGABRT	-	raise signal for program abortion with `_exit(-1)`
SIGFPE	PROCHK	Error in a floating-point operation
SIGKILL	-	raise signal for program abortion with `exit(-1)`
SIGSEGV	ERROR	Memory access with invalid segment access
SIGALRM	RTIMER	A time interval has elapsed (real time)
SIGTERM	TERM	Program termination
SIGUSR1	-	Defined by the user
SIGUSR2	-	Defined by the user
SIGDVZ	PROCHK	Division by 0
SIGXCPU	RUNOUT	CPU time has run out
SIGBPT +	SVC	Breakpoint (currently not supported)
SIGTIM	TIMER	A time interval has elapsed (CPU time)
SIGINTR	INTR	SEND-MESSAGE command
SIGSVC +	SVC	SVC call (currently not supported)

Signals marked with a “+” are currently not supported.

Return val. 0 the signal was sent successfully.

-1 the signal could not be sent, because *sig* is not a valid signal number. In addition, `errno` is set to EINVAL (invalid signal number).

Notes

With the exception of SIGKILL, the `raise` signals can be intercepted with the signal function. You will find detailed information on this topic under `signal`.

If the program does not provide for the handling of `raise` signals, it is terminated with `exit(-1)` when a signal arrives, and the following messages are displayed:

```
"CCM0101 signal occurred: signal"
"CCM0999 Exit -1"
```

Signal SIGABRT
SIGABRT 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.

Signal SIGKILL
SIGKILL 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.

Example

A program that aborts itself.

```
#include <signal.h>

int main(void)
{
  for(;;)
    raise(SIGKILL);
  return 0;
}
```

See also

alarm, atexit, exit, _exit, signal

rand - Random number generator

Definition #include <stdlib.h>

int rand(void);

`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}.

Return val. Random number within [0, 2^{15}-1].

Note 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, like `srand(1)` does.

Example 1 Generate the same five random numbers twice:

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

int i;

int main(void)
{
   for(i=1; i <= 10; ++i)
      {
        printf("%d\n", rand());
        if(i == 5)
          srand(1);
      }
   return 0;
}
```

Example 2 Simulation of rolling dice.

```
#include <stdio.h>
#include <stdlib.h>
#define A 32767                 /* 2**15 - 1 */

int cpu_t;                      /* Query variable for CPU time*/
int i,x;

int main(void)
{
cpu_t = cputime();
srand(cpu_t);                   /* Seed value for the random generator  */
for(i=1; i<= 6; ++i)            /* Simulation of six throws of a die    */
  {
  x =  rand()/(A/6)+1;          /* Determine random number in range 1-6 */
  printf("number thrown= %d\n",x);
  }
return 0;
}
```

See also rand, srand

read - Read from a file (elementary)

Definition #include <stdio.h>

int read(int fd, char *puf, int n);

`read` is the elementary read function.

`read` reads from the file with file descriptor *fd* a maximum of *n* characters into the area pointed to by *buf*.

In text files, `read` only reads the characters within one line per call. Input is terminated at the end of the line.
In binary files, newline (\n) characters are ignored by `read`.
SAM files are always processed as text files with elementary functions.

Parameters int fd

File descriptor for the input file.
A file descriptor (positive integer) is the result of a successful `open`/`open64` or `creat`/`creat64` call.
The file descriptors for `stdin` (0), `stdout` (1), and `stderr` (2) are automatically assigned when the program is started.

char *buf

Pointer to the area into which the read data is to be written. The area should be at least *n* bytes in size.

int n

Maximum number of bytes to be read. If the end of the line is reached first, fewer than *n* bytes will be read.

Return val. The number of bytes actually read
if successful.

0 for end of file.

-1 if nothing was `read` due to one of the following errors:

- physical I/O error
- *fd* is not a valid file descriptor
- the file is not present
- no access permission for the file exists
- *n* is impossible

Notes

The number of bytes actually read may be less than the specification in *n* if the end of the line is reached first (only applies to text files) or if end of file or an error is encountered.

You should use `sizeof` to ensure that the number of bytes read does not exceed the amount that can be accepted by the buffer.

Example

The following program copies the standard input (file descriptor 0) to the standard output (file descriptor 1). If you use the redirection mechanism for `stdin` and `stdout` (PARAMETER-PROMPTING in the RUNTIME option), you can copy from any source to any destination with this program. BUFSIZ (8192 bytes) is defined in the include file <stdio.h>.

```
#include <stdio.h>

int main(void)
{
  char buf[BUFSIZ];
  int n;

  while((n = read(0, buf, sizeof(buf))) > 0)
        write(1, buf, n);
  return 0;
}
```

See also

write, open, open64, creat, creat64

realloc - Alter memory space

Definition

#include <stdlib.h>

void *realloc(void *p, size_t n);

`realloc` changes the size of the memory area pointed to by *p* to *n* bytes. *p* must have been returned by a previous `malloc` or `calloc` call.

`realloc` is part of a C-specific memory management package which internally administers memory areas that are requested and subsequently freed. Attempts are made to satisfy new requests by first using areas that are already being managed and only then by the operating system (cf. `garbcoll` function).

Return val.

Pointer to the beginning of the modified memory area	if successful.
NULL pointer	if `realloc` was unable to alter the memory space, e.g. because the memory space still available is insufficient or because an error occurred.

Notes

If `realloc` alters the size of a memory area, it may happen that the allocated block is shifted. In such cases, the contents of the pointer passed as an argument are not identical with 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 the NULL pointer, the block to which *p* points may have been destroyed!

If *p* is a NULL pointer, `realloc` functions like a `malloc` call for the specified size.

If *n* has the value 0, `realloc` returns an unambiguous address which can also be transferred to `free`.

Example

The following program fragment first requests memory space for 20 characters and then extends this area to accept 80 additional characters (i.e. to a total of 100 bytes).

```
#include <stdlib.h>

char *char_array;
char_array = (char *)malloc(20 * sizeof(char));
     .
     .
char_array = (char *)realloc(char_array, 100 * sizeof(char));
```

See also

malloc, calloc, free, garbcoll

remove - Delete file

Definition

```
#include <stdio.h>

int remove(const char *f_name);
```

`remove` deletes the file *f_name*. *f_name* may be a fully or partially qualified file name.

Return val.

0	if successful.
-1	if the file cannot be deleted, e.g. if there is no file with the name *f_name* or the file has been opened by another task. In addition, `errno` is set to EDMS.

Notes

If a partially qualified file name is specified, then `remove` will delete all corresponding files without asking for confirmation (Y/N). 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.

Record I/O

`remove` can also be used unchanged on files with record I/O.

rename - Rename file

Definition #include <stdio.h>

```
int rename(const char *name_old, const char *name_new);
```

`rename` gives the file with the name *name_old* the new name *name_new*.

Return val.

0	if successful.
-1	if the file could not be renamed. If for example – there is no file with the name *name_old*, – a file is already cataloged under the name *name_new* or – the file to be renamed has been opened by a program. In addition, `errno` is set to EMACRO.

Record I/O `rename` can also be used unchanged on files with record I/O.

rewind - Position read/write pointer to beginning of file

Definition

```
#include <stdio.h>

void rewind(FILE *fp);
```

`rewind` positions the read/write pointer of the file with file pointer *fp* to the beginning of the file.

Notes

The calls `rewind(fp)`, `fseek(fp,0L,SEEK_SET)` and `fseek64(fp,0LL,SEEK_SET)` are equivalent, except that `rewind` does not return a value.

System files (SYSDTA, SYSOUT, SYSLST) cannot be positioned.

If new records are written to a text file (opened for creation or in append mode) and a `rewind` call is issued, any residual data is first written from the buffer to the file and terminated with an end-of-line character (\n).
Exception for ANSI functionality:
If the data of an ISAM file in the buffer does not end in a newline character, `rewind` does not cause a change of line (or change of record), i.e. the data is not automatically terminated with a newline character when writing from the buffer. Subsequent data lengthens the record in the file. When an ISAM file is read, therefore, only those newline characters explicitly written by the program are read in.

If the `rewind` function is called successfully, it deletes the EOF flag of the file and cancels all the effects of the preceding `ungetc` calls for this file.

Record I/O

`rewind` can also be used unchanged on files with record I/O.

Example The following program first processes a file from the 11th character onwards to the end of the file and then continues at the beginning of the file (only works with binary files, i.e. in this case only with SAM and PAM files).

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   int c,i;

   fp = fopen("input","rb");
               /* skip the first 10 characters */
   fseek(fp,10L,SEEK_SET);
   while((c=getc(fp)) != EOF)
         putc((char)c,stdout);
               /* position to the beginning of the file */
   rewind(fp);
   for(i=0; i<10; i++)
      {
        c=getc(fp);
        putc((char)c,stdout);
      }
   fclose(fp);
   return 0;
}
```

See also fseek, fseek64, fsetpos, fsetpos64

rindex - Last occurrence of a character in a string

Definition

#include <string.h>

char *rindex(const char *s, int c);

`rindex` searches for the last occurrence of the character *c* in the string *s* and, if successful, returns a pointer to the located position in *s*.

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*.

Note

The `rindex` and `strrchr` functions are equivalent.

Example

Find the last ’s’:

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

int main(void)
{
   char *s = "What fun in the ssun!";
   printf("%s\n", s);
   printf("Where is the mistake? %s\n", rindex(s, 's'));
   return 0;
}
```

See also

index, strchr, strrchr

rint, rintf, rintl - Round off to nearest whole number

Definition #include <math.h>

double rint(double x);

float rintf (float x);

long double rintl (long double x);

Each of the functions returns the whole number nearest to x, in floating point representation. `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 return value is rounded off in accordance with the rounding mode currently set for the system. If the rounding mode 'round-to-nearest' is set and if the difference between x and the rounded result is exactly 0.5, the next even whole number is returned.

If the defined rounding mode rounds off in the direction of positive infinity, then `rint` is identical to `ceil`. If the defined rounding mode rounds off in the direction of negative infinity, then `rint` is identical to `floor`.
In this version, the rounding mode is preset in the direction of positive infinity.

Return val.

integer	represented as a number of type `double`, `float` or `long double` if successful.
HUGE_VAL	in the event of an overflow, `errno` is set to ERANGE to indicate the error.

Note In this version, the rounding mode is preset in the direction of positive infinity.

See also abs, ceil, floor, llrint, llround, lrint, lround, round

round, roundf, roundl - Round off to nearest whole number

Definition

#include <math.h>

double round(double x);

float roundf (float x);

long double roundl (long double x);

Each of the functions returns the whole number nearest to *x*, in floating point representation. `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 return value is independent of the defined rounding mode. If the difference between *x* and the rounded result is exactly 0.5, the larger whole number is returned.

Return val.

integer	represented as a number of type `double`, `float` or `long double` if successful.
undefined	in the event of an overflow or underflow, `errno` is set to ERANGE to indicate the error.

See also

abs, ceil, floor, llrint, llround, lrint, lround, rint

scanf - Formatted input from the standard input

Definition

```
#include <stdio.h>

int scanf(const char *format, argumentlist);
```

`scanf` reads data (input fields) from the standard input `stdin`, converts this data according to specifications in the format string *format*, and stores the results in the areas which you specify with the result pointers in the argument list. Each argument must be a pointer to a variable whose data type corresponds to a type specification in the format string *format*. The format string controls how the input field is to be interpreted and converted.

Parameters const char *format

The format string may contain three classes of characters or specifications:

1. White space characters
2. Any characters except white space characters and the percent character (%).
3. Format statements beginning with the percent character (%)

White space (KR functionality, applies to C/C++ versions prior to V3.0 only)

␣	blank
\n	newline
\t	tab

White space (ANSI functionality)

␣	blank
\n	newline
\t	tab
\f	form feed
\v	vertical tab
\r	carriage return

The format string may contain any number of white space characters (or none). These characters have no control function.

Any white space characters in the input are treated as delimiters between input fields and are not converted (cf. %c and %[ ] for exceptions).

Any character except % and white space character

The character must match the next character of input. `scanf` reads the input character, but does not convert it or store it in a variable. If the input character does not match the character specified here, input processing is aborted.

The format statement is described below separately for KR functionality and ANSI functionality.

Format statement (KR functionality, applies to C/C++ versions prior to V3.0 only)

Format statements contain specifications on how the input fields are to be interpreted and converted. They may be structured as follows:

```
            ⎧ [{l|h}] {d|o|x}  ⎫
            ⎪                  ⎪
            ⎪ [l] {e|f}        ⎪
            ⎪                  ⎪
            ⎪ {D|O|X|E|F}      ⎪
% [*][n]   <                    >
            ⎪ {c|s}            ⎪
            ⎪                  ⎪
            ⎪ {[...]|[^...]}   ⎪
            ⎪                  ⎪
            ⎩ %                ⎭
```

A format statement is associated with one input field. An input field is a string of characters that is terminated

- by the first white space character
- by a character that does not match the type specification in the format statement
- when the explicitly specified field length *n* is reached.

Leading white space characters are ignored during input.

Each format statement 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 specification in the format statement appears before this, the length is correspondingly shortened.

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`. Internally the value is represented as `unsigned`.

x — A hexadecimal integer value is expected. The corresponding argument may be a pointer to `unsigned int` or `int`. Internally the value is 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 unsigned integer value).
The decimal point 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 `character`.
In this case `scanf` also reads blanks. "%1s" should be used to read the next character that is not a blank. c is suitable for reading strings that also contain 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 and large enough to be able to accept the string and a terminating null byte (\0). `scanf` automatically terminates the string with the null byte. Leading white space characters in the input are ignored and 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 and large enough to be able to accept the string (including the null byte that is automatically appended). 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, no conversion.

Format statement (ANSI functionality)

Format statements contain information as to how the input fields are to be interpreted and converted. They may be structured as follows:

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

A format statement is associated with an input field. An input field is a sequence of characters which is terminated

– by the first white space character,

– by a character which does not match the format statement (type specification),

– when the explicitly specified field length *n* is reached.

Leading white space characters are ignored.

Every format statement 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 specification in the format statement appears before this, the length is correspondingly shortened.

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 pointer to `long long int` (d, i) or `unsigned long long int` (o, u, x, X).

ll before n:
The argument is of the 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 the 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`. Internally the value is 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`. Internally the value is 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 unsigned integer value).
The decimal point 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 `character`.
In this case, `scanf` also reads blanks. "%1s" should be used to read the next character that is not a blank. c is suitable for reading strings that also contain 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 and large enough to be able to accept the string and a terminating null byte (\0). `scanf` automatically terminates the string with the null byte. Leading white space characters in the input are ignored and 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 and large enough to be able to accept the string (including the null byte that is automatically appended). 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 specified in the square brackets is encountered. In other words, the string may only consist of characters within the square brackets [ ]. All characters not specified are treated as delimiters.
The closing bracket] can be included in the list of characters to be read if it is specified as the first character immediately after the opening bracket: [ ]...].

[^...] 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.
The closing bracket] can be included in the list of delimiters if it is specified 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 that `scanf` has processed up to this time.

% Input of the % character, no conversion.

argumentlist
Pointer to variables in which `scanf` is to store the converted results.

No pointer arguments may be specified for %* statements (skip assignment) in *format*. There must be one pointer argument each for all other % statements. The data type of the pointer argument is determined by the type specification in the corresponding format statement.

Return val. Number of input fields read and successfully converted.
This does not include any input fields for which %* (skip assignment) was specified.

EOF if an error occurs before the start of the conversion.

Notes In converting integer values to `unsigned int` (o, u, x, X) the twos complement is formed from a value with a negative sign.
For example, format %u for input -1 gives X'FFFFFFFF'.

You should always check the result of a `scanf` call to be sure 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 there with `getc`; otherwise, the next `scanf` call will receive the same character again.

If there are more pointer arguments than format statements (excluding the %* specifications), the excess arguments are ignored. If there are fewer arguments, results are undefined.

Example 1
```
int i;
float x;
char name[20];
scanf("%2d %f %*d %6s", &i, &x, name);

Input data: 234567 678 Hubertxy

Content of the variable after scanf:
```

```
i:     23
x:     4567.0
name:  Hubert\0
```

In the above example, 678 is not assigned due to the %* specification. The next read operation that is called starts with the character ’x’.

Example 2

```
int i;
float x;
char name[50];
scanf("%2d %f %*d %6s", &i, &x, name);

Input data: 25 54.32E-1 thomson

Content of the variable after scanf:

i:     25
x:     5.432
name:  thomso\0
```

Example 3

```
char string1[20];
char string2[20];
scanf("%[1234567890] %[^,!:]", string1, string2);

Input data: 234567ab  c,de

Content of the variable after scanf:

string1: 234567
string2: ab  c
```

See also fscanf, sscanf

setbuf - Set input/output buffer

Definition #include <stdio.h>

void setbuf(FILE *fp, char *buffer);

`setbuf` sets up a memory area for the file with the file pointer *fp*. This memory area is then used instead of the area assigned by the system for buffering the input/output data.

The file pointer *fp* must point to a file which is already open and for which no read or write functions have yet been performed.

Parameters FILE *fp

Pointer to the file for which an input/output buffer is to be made available.

char *buffer

Pointer to the area to be used as the buffer or NULL.
If the argument is a NULL pointer, the buffer assigned by the system is used.

Note The pointer *buffer* must point to an area of size BUFSIZ for a file with default attributes. BUFSIZ is defined in <stdio.h>.
If the blocking factor is explicitly defined with the BUFFER-LENGTH parameter of the ADD-FILE-LINK command, the size of the area must correspond to this defined blocking size.

See also setvbuf

setjmp - Set label for non-local jumps

Definition

#include <setjmp.h>

int setjmp(jmp_buf env);

`setjmp` is only meaningful when used in conjunction with the `longjmp` function. These functions can be used to implement non-local jumps, i.e. a jump from any given function to another, still active function.

`setjmp` stores the current program state (address in the C runtime stack, program counter, register contents) in a field of type `jmp_buf`. The type `jmp_buf` is defined in <setjmp.h>. A subsequent `longjmp` call re-establishes the program state stored by `setjmp` and continues program execution from this state.

A detailed description and notes on `setjmp`/`longjmp` are provided under the `longjmp` function.

Return val.

0	normal return, i.e. a `longjmp` call was not used to branch to the position after the `setjmp` call.
≠ 0	if `longjmp` was used to branch to the position after the `setjmp` call. This return value is the argument *value* of the `longjmp` call (if *value* is equal to 0, `setjmp` returns 1).

Example

See example under `longjmp`

See also

longjmp, signal

setlocale - Set/query locale

Definition #include <locale.h>

char *setlocale(int category, const char *locale);

With `setlocale` you can query the current locale or select a new locale. The locale may relate to some or all the locale variables of the program.
The locale variables are defined in <locale.h>.

Parameters int category

Category of locale variables to which the selected *locale* is to refer. *category* may contain the following predefined values:

LC_ALL	Locale variables of all categories.
LC_COLLATE	The sorting sequence affects the behavior of the `strcoll` and `strxfrm` functions.
LC_CTYPE	The character type affects the behavior of the macros for character processing is... (not `isdigit` or `isxdigit`), `tolower`, `toupper`, `strlower` and `strupper`.
LC_MONETARY	The conventions for representing monetary values affect the values returned by `localeconv`.
LC_NUMERIC	The conventions for representing non-monetary numerical values affect the type of decimal point for formatted input/output and for converting strings (`atof`, `strtod`), and the values returned by `localeconv`.
LC_TIME	The conventions for representing date and time affect the behavior of `strftime`.

const char *locale

String which selects the locale. The following predefined locales are available (a detailed description is provided in section “Predefined locale C” on page 94ff):

"C"	Defines the minimum environment for compiling a C program and is the default setting when the program starts (exception: see locale "V1CTYPE").
""	Standard locale. In this version it corresponds to locale “C”.
"V1CTYPE"	Locale compatible with the C runtime system C1.0. "V1CTYPE" is automatically set when the program starts if the main routine is a C V1.0 object.
"V2CTYPE"	Locale compatible with C runtime systems V2.0 and V2.1.
"GERMANY"	Country-specific locale that conforms to the national conventions.
"De.EDF04F"	Country-specific locale whose conversion tables are based on ASCII code ISO 8859-15 or EBCDIC code EDF04F, and which supports the currency “DM” in the category LC_MONETARY.
"De.EDF04F@euro"	Country-specific locale whose conversion tables are based on ASCII code ISO 8859-15 or EBCDIC code EDF04F, and which supports the currency “euro” in the category LC_MONETARY.

The strings are predefined in the include file <locale.h> as follows:

LC_C_C	"C"
LC_C_DEFAULT	""
LC_C_V1CTYPE	"V1CTYPE"
LC_C_V2CTYPE	"V2CTYPE"
LC_C_GERMANY	"GERMANY"
LC_C_De.EDF04F	"De.EDF04F"
LC_C_De.EDF04F@euro	"De.EDF04F@euro"

If a NULL pointer is passed for *locale*, the current locale for *category* is not changed.

Return val. Pointer to a string specifying the current locale for the specified *category*.

This string can be used as the *locale* parameter in setlocale calls.
The string can assume the following values:
“C”, “V1CTYPE”, “V2CTYPE”, “GERMANY”, “De.EDF04F”, “De.EDF04F@euro”.
For the LC_ALL category, the string contains the value “C”, provided this value has been set for all categories.
As soon as a locale other than “C” is set for a category, the string contains the locales for all the categories. The values for the individual categories are prefixed by a slash (/), which indicates the beginning of a new value. The sequence of locales corresponds to the above-mentioned sequence of

categories (see parameter description *int category*).
The last (sixth) locale in the string refers to the LC_MESSAGES category which is currently not supported in the locales “C”, “GERMANY”, “VC1TYPE” and “VC2TYPE” and is set to “C” if you enter one of these locales.
If you enter the locale “De.EDF04F” or “De.EDF04F@euro” here, the corresponding value is entered in the category LC_MESSAGES.

If a string containing the locales for all categories is used as the *locale* parameter in a setlocale call and a category other than LC_ALL is specified, only the locale for the specified category is taken from this string (without the leading slash).

Example of the return value for LC_ALL:

```
"/V2CTYPE/C/GERMANY/C/GERMANY/C"
     1      2    3    4    5    6
```

Position	**Category**	**Locale**
1	LC_COLLATE	V2CTYPE
2	LC_CTYPE	C
3	LC_MONATARY	GERMANY
4	LC_NUMERIC	C
5	LC_TIME	GERMANY
6	LC_MESSAGES	C

NULL pointer if the selected category cannot be recognized. The current locale remains unchanged.

Notes The available locales are described in detail in chapter “Locale” on page 93.

User-specific locales:
In addition to the predefined locales mentioned above, you may implement your own locales and select them with `setlocale` (see section “User-specific locales” on page 110).

The string to which the return value of setlocale points must not be explicitly changed by the program. It may only be overwritten by `setlocale` calls.

If you are only querying the current locale and not changing it, a NULL pointer must be passed for *locale*.

See also localeconv

setvbuf - Set input/output buffer

Definition #include <stdio.h>

int setvbuf(FILE *fp, char *buffer, int type, size_t n);

`setbuf` sets up a memory area for the file with the file pointer *fp*. This memory area is then used instead of the area assigned by the system for buffering the input/output data.

The file pointer *fp* must point to a file which is already open and for which no read or write functions have yet been performed.

Parameters FILE *fp

Pointer to the file for which an input/output buffer is to be made available.

char *buffer

Pointer to the area to be used as the buffer or NULL.

If the argument is a NULL pointer, the buffer assigned by the system is used.

int type

Type of buffering for the file. This parameter is checked only syntactically and otherwise ignored. It must contain one of the following predefined values:

_IOFBF (full buffering)
_IOLBF (line buffering)
_IONBF (no buffering, not supported).

The type of buffering is determined by the type of file and cannot be changed by the user:
Text files are line-buffered, i.e. the data is written to the file whenever a newline character (\n) occurs.
Binary files are full-buffered, i.e. the data is written to the file when the buffer is full.
Unbuffered input/output is not supported.

size_t n

Size of the buffer in bytes.

Return val.

0	if the `setvbuf` function has been successfully executed.
≠ 0	if a (syntactically) invalid value has been passed for *type* or the function cannot be executed.

Note The pointer *buffer* must point to an area of size BUFSIZ for a file with default attributes. BUFSIZ is defined in <stdio.h>.
If the blocking factor is explicitly defined with the BUFFER-LENGTH parameter of the ADD-FILE-LINK command, the size of the area must correspond to this defined blocking size.

See also setbuf

signal - Signal processing control

Definition

#include <signal.h>

void (*signal(int sig, void (*fct) (int))) (int);

The `signal` function is provided for the handling of signals.

Signals that can be received and processed by a program can be distinguished into two types, depending on the way in which they are triggered. The internal handling of a signal varies in implementation on the basis of its type.

1. STXIT events

   STXIT events are triggered

   – by program errors, e.g. address error, execution of invalid instructions, division by zero etc.

   – by the `alarm` function

   – externally, e.g. by pressing the K2 key, entering specific commands (ABEND, INFORM-PROGRAM etc.)

   The handling of these events is implemented internally via the BS2000-specific STXIT contingency mechanism. This mechanism as well as the STXIT event classes are described in detail in the "Executive Macros" manual.

2. `raise` signals

   All events that can be triggered by the `raise` function are grouped under this type of signal. `raise` can be used to simulate STXIT events and to send (user-own and predefined) signals unrelated to STXIT events.

   The handling of this type of signal is C-specific, i.e. not implemented via the mechanism mentioned above.

If there is no provision for the handling of signals in a program, the program will be aborted when a signal arrives.

A program can, however, also intercept a signal. This is achieved by calling the `signal` function and by passing to it a function *fct* as its argument.
This makes it possible to respond to a signal in the following ways:

– If *fct* is the default function SIG_DFL, the program is aborted.

– If *fct* is the predefined function SIG_IGN, the signal is ignored.

– If *fct* is a user-defined routine, the signal is handled as defined by this routine.

These three signal handling options are discussed below in somewhat greater detail in order to underline the differences in the handling of STXIT events and `raise` signals.

Program abortion

Program abortion occurs if the program does not provide for signal handling or if `signal` is called with the SIG_DFL function.

STXIT event:

The implementation-defined default termination response is executed by the operating system. The program is aborted, and information on the interruption address and the severity of the error is output together with a DUMP message:

```
... PROCESSING INTERRUPTED AT address ..., EC=severity
... DUMP DESIRED? REPLY(Y=YES,N=NO)?
```

`raise` signal:

A C-specific program termination is effected via exit(-1), and the following messages are output:

```
"CCM0101 signal occurred: signal"
"CCM0999 Exit -1"
```

Ignore signal

A signal is ignored if the `signal` function is called with the predefined function SIG_IGN. Program execution continues as if no signal had occurred. No distinction is made in this case between the handling of STXIT events and `raise` signals.

Handling the signal with a user-defined function fct

A signal is handled in accordance with a user-defined function *fct* if the `signal` function is called with the name of this function. When a signal arrives, the calling program is interrupted and the function *fct* is executed. On termination of signal processing, the program is continued at the point at which it was interrupted (unless the `exit` or `longjmp` functions were called in *fct*).

STXIT event:

fct is implemented internally as an independent STXIT contingency process; the rest of the program as a so called "basic task". Control is effected by the operating system.

`raise` signal:

fct is treated internally as a "normal" C function and is not implemented via the contingency mechanism. Control is under the C runtime system.

Further details pertaining to the different implementation of `signal` calls and their various related options are provided in the "Notes".

Parameters int sig

Signal to be processed.

The symbolic constants listed under "SIGNR" in the following table may be used for *sig*. These constants are defined in the include file <signal.h>.
In its last column, the table additionally lists the various ways in which the signal can be triggered. The particular STXIT event class is specified for STXIT events.

SIGNR	**Meaning**	**Signal triggered via STXIT event / raise / alarm**
SIGHUP	Disconnection of link to terminal	ABEND / raise
SIGINT	Interrupt from the terminal (K2)	ESCPBRK / raise
SIGILL	Execution of an invalid instruction	PROCHK / raise
SIGABRT	raise signal for program abortion with `_exit(-1); abort`	raise / abort
SIGFPE	Error in a floating-point operation	PROCHK / raise
SIGKILL	raise signal for program abortion with `exit(-1)`	raise
SIGSEGV	Memory access with invalid segment access	ERROR / raise
SIGALRM	A time interval has elapsed (real time)	RTIMER / raise / alarm
SIGTERM	Program termination	TERM / raise
SIGUSR1	Defined by the user	raise
SIGUSR2	Defined by the user	raise
SIGDVZ	Division by 0	PROCHK / raise
SIGXCPU	CPU time has run out	RUNOUT / raise
SIGBPT	Breakpoint (not supported)	SVC
SIGTIM	A time interval has elapsed (CPU time, SETIC)	TIMER / raise
SIGINTR	SEND-MESSAGE command	INTR / raise
SIGSVC	SVC call (not supported)	SVC

The symbolic constant for the signal number can be supplemented with an additional symbolic name, e.g. `signal(SIGDVZ + SIG_PSK, fct)`. This addition (“+ SIG_PSK” in the example) controls whether the function *fct* is to be activated only on the basis of an STXIT event or also on the basis of a raise signal. In addition, it also determines whether *fct* is to be temporarily or permanently assigned to the associated signal. Technical details on this topic are provided in the “Notes”. The symbolic names are defined in <signal.h>.

If no addition is specified, the system defaults to SIG_TSK.

Symbolic name	**Assignment**	**Activation via**
SIG_TSK	temporary	STXIT / raise (default)
SIG_TS	temporary	STXIT
SIG_PSK	permanent	STXIT / raise
SIG_PS	permanent	STXIT

void (*fct)(int)

Name of the function to be called if a signal occurs. This function receives the signal number of type `int` as its only argument.
The function must be defined *before* the corresponding `signal` call!

There are two predefined functions in <signal.h>:

SIG_DFL — This function is the default and causes the program to abort. The manner of termination depends on whether an STXIT event or a `raise` signal is involved (see above).

SIG_IGN — The signal is ignored.

Return val. The signal handling function valid prior to the `signal` call,
if successful. `signal` returns the last setting for signal handling, which can be SIG_DFL, SIG_IGN, or a user-defined function *fct*.

SIG_ERR (= 1)
in case of error, e.g. if *sig* is not a valid signal number or *fct* points to an invalid address.
In addition, `errno` is set to the appropriate error code:
EINVAL (invalid argument)
EFAULT (invalid address).

Notes The signal SIGKILL cannot be intercepted, i.e. neither a user-defined function nor SIG_IGN may be assigned to it.

If a second function for signal handling is registered for a signal that already has a signal handling function assigned to it, the first function is unassigned before the new function is registered. Consequently, there will be *no* signal handling registered for that signal for a brief period of time.

It is not possible to use a `longjmp` call to return from a function assigned to the signal SIGTERM. This is because entries in the C runtime stack have already been cleared for all functions, including `main`, at the time the signal is triggered.

Temporary/permanent allocation :
Provisions for the temporary assignment of a signal to a function have been made in many implementations (e.g. UNIX) as well as in the ANSI standard. This means that the user-defined assignment of a function to a signal number is only valid temporarily, i.e. for a single occurrence of the signal. The assignment is cancelled after the signal arrives, and the system resets to the default SIG_DFL (abort program).
Only the SIG_IGN assignment (ignore signal) is permanently valid for multiple occurrences of the associated signal.

– In BS2000, signal handling for "STXIT event" type signals is implemented via the STXIT contingency mechanism. This mechanism is based on a permanent assignment of an STXIT event to an STXIT contingency routine, i.e. a temporary assignment can only be achieved by explicitly deactivating the routine.

– In order to provide for the temporary assignment of many implementations on one hand, and to effectively support the permanent nature of BS2000 implementations on the other, both options have been made available, i.e. the user may choose whether a signal routine is assigned temporarily or permanently.

– For performance reasons, the user is additionally offered the option of deciding whether a signal routine can only be triggered by STXIT events (which is more efficient), or whether it may also be triggered by `raise` signals.

– The options noted above are implemented by means of symbolic additions to the actual signal number: SIG_TSK, SIG_TS, SIG_PSK, SIG_PS (see the parameter description for *sig*).

– If you want to intercept a signal with *fct* without exception, the following `signal` calls are among those that can be used:

```
signal(SIGDVZ + SIG_PSK, fct);  /* fct is activated by the STXIT */
                                /* event and the raise signal SIGDVZ */

signal(SIGDVZ + SIG_PS, fct);   /* fct is activated only by the STXIT */
                                /* event SIGDVZ. */
```

The following calls are equivalent, i.e. both provide for temporary assignment and cause the signal routine to be activated by an STXIT event as well as a `raise` signal:

```
signal(SIGDVZ, fct);
signal(SIGDVZ + SIG_TSK, fct);
```

Problems may arise in the case of the three different signal numbers that are mapped by the same STXIT event class (PROCHK). The following `signal` calls are handled differently, depending on the way in which the signal was triggered:

```
signal(SIGILL, fct1);
signal(SIGFPE, fct2);
signal(SIGDVZ, fct3);
```

STXIT event:

fct3 is called in any case if the SIGILL and SIGFPE signals are intercepted via the STXIT contingency mechanism. In fact, even if a `signal` call is only provided for one signal, the assigned routine is activated when any of the three signals arrives.

`raise` signal:

If the signals are triggered with the `raise` function, on the other hand, the currently assigned function is activated. Signals for which no `signal` call has been provided are handled as defined by the default setting (SIG_DFL, program abortion).

SIG_DFL is the default for all signals at the start of a program.

In order to execute, a function which is assigned to a signal requires an intact C environment. Consequently, when a program terminates properly, all signal routines are logged off immediately before the C environment is cleared down. No events which occur after this – not even SIGTERM – are then intercepted.

Example The following program intercepts the STXIT events SIGDVZ (division by 0) and SIGINT (interrupt with the K2 key) with the function *fct* and outputs a corresponding error message. After handling both interrupt events (which occur at different positions in the program), the program continues to execute at the same program location (new input prompt) by using the setjmp and longjmp functions.

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

jmp_buf env;

void fct(int sig)
{
  if(sig == SIGDVZ + SIG_PS)
    printf("Division error, please repeat input\n");
  if(sig == SIGINT + SIG_PS)
    printf("K2 key pressed, please repeat input\n");
  longjmp(env, 1);
}

int main(void)
{
  float a;
  float b;
  double z;

  signal(SIGDVZ + SIG_PS, fct);
  signal(SIGINT + SIG_PS, fct);
  setjmp(env);
  printf("Please enter a and b\n");    /* Interrupt with K2 possible */
  scanf("%f %f", &a, &b);
  z = a / b;                           /* Division by 0 possible, */
                                       /* if b = 0 */
  printf("z = %f\n", z);
  printf ("End of program\n");
  return 0;
}
```

See also alarm, longjmp, raise, setjmp

sin - Sine

Definition #include <math.h>

double sin(double x);

`sin` calculates the trigonometric sine function for the floating-point number *x*, specifying the angle in radians.

Return val. `sin(x)` a floating-point number in the interval [-1.0, +1.0].

Example The following program outputs the sine of values in the range -pi to +pi.

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

#define pi 3.14159265358979

int main(void)
{
   double x;
   for (x = -pi; x <= pi; x = x + pi/4.)
       printf(" sin(%1.2f) = %.4f \n ", x, sin(x));
   return 0;
}
```

See also cos, asin, sinh

sinh - Hyperbolic sine

Definition #include <math.h>

double sinh(double x);

`sinh` calculates the hyperbolic sine for the floating-point number *x*.

Return val. `sinh(x)` for a floating-point value *x*, if successful.

+/-HUGE_VAL in the event of an overflow (depending on the sign for *x*). In addition, `errno` is set to ERANGE (result too large).

Example The following program outputs the hyperbolic sine of values in the range -1 to 1 (increment 0.1).

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

int main(void)
{
   double x;
   for (x = -1.0; x < 1.0; x = x + 0.1)
       printf(" sinh(%.2f) = %.4f \n ", x, sinh(x));
   return 0;
}
```

See also cosh, asin, sin

sleep - Suspend a program for a fixed period of time

Definition #include <signal.h>

int sleep(unsigned int sec);

`sleep` suspends a program for *sec* seconds.

Return val. Requested time minus actual time.

If `sleep` was ended earlier than specified in *sec*, the time still remaining will be indicated (cf. note).

Note `sleep` delays the program for *sec* seconds by internally calling the VPASS macro with a value of one second in a loop.

Although the program is suspended for *sec* seconds when `sleep` is called, time continues to run for a previously set alarm clock (with `alarm`). The implications of this are detailed below:

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 *sec* 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, signal

snprintf - Formatted output to a string

Definition #include <stdio.h>

int snprintf(char *s, size_t n, const char *format, argumentlist);

`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`.

Parameters char *s

Pointer to the result string. `snprintf` terminates the string with the null byte (\0). The maximum length of the output is therefore *n*-1.

size_t n

Length of the area reserved for the result string. *n* may not be greater than `INT_MAX`. When *n* = 0, no output takes place.

const char *format

Format string as described under `printf` with KR or ANSI functionality (cf. `printf`).

The only difference is with regard to the way in which the control characters for white space (\n, \t, etc.) are handled. As opposed to `printf`, `sprintf` enters the EBCDIC value of the control character in the result string. It is only during output to text files that the control characters are converted to their appropriate effect depending on the type of text file (see section “White space” on page 65).

argumentlist

Variables or constants whose values are to be converted and formatted for output according to the information in the format statements.
If the number of format statements does not match the number of arguments, the following applies:
If there are more arguments, the surplus arguments are ignored.
If there are fewer arguments, the results are undefined.

Returnwert

< 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.

Notes

You must see to it that the area to which *s* points is large enough for the result!

`snprintf` rounds to the specified precision when converting floating-point numbers.

`snprintf` does not convert one data type to another. A value must be explicitly converted (e.g. with the `cast` operator) if it is not to be output in conformity with its type.

Maximum number of characters to be output:
With KR functionality (applies to C/C++ versions prior to V3.0 only) a maximum of 1400 characters can be output per sprintf call,
with ANSI functionality a maximum of 1400 characters per conversion element (e.g. %s).

The behavior is undefined if memory areas overlap.

Attempts to output non-initialized variables or to output variables in a manner inconsistent with their data type can lead to undefined results.

The behavior is undefined if the percent sign (%) in a format statement is followed by an undefined formatting or conversion character.

See also

printf, fprintf, sprintf, putc, putchar, puts, sscanf

sprintf - Formatted output to a string

Definition #include <stdio.h>

int sprintf(char *s, const char *format, argumentlist);

`sprintf` edits data (characters, strings, numerical values) according to specifications in the string *format* and writes this data to the area pointed to by *s*.

`sprintf` works like `printf`, except that the edited data is written to a string and not to the standard output.

Parameters char *s

Pointer to the result string. `sprintf` terminates the string with the null byte (\0).

const char *format

Format string as described under `printf` with KR or ANSI functionality (cf. `printf`).

The only difference is with regard to the way in which the control characters for white space (\n, \t, etc.) are handled. As opposed to `printf`, `sprintf` enters the EBCDIC value of the control character in the result string. It is only during output to text files that the control characters are converted to their appropriate effect depending on the type of text file (see section "White space" on page 65).

argumentlist

Variables or constants whose values are to be converted and formatted for output according to the information in the format statements.
If the number of format statements does not match the number of arguments, the following applies:
If there are more arguments, the surplus arguments are ignored.
If there are fewer arguments, the results are undefined.

Return val. Number of characters stored in *s*.

The terminating null byte (\0) generated by `sprintf` is not included in this total.

Notes You must see to it that the area to which *s* points is large enough for the result!

`sprintf` rounds to the specified precision when converting floating-point numbers.

`sprintf` does not convert one data type to another. A value must be explicitly converted (e.g. with the `cast` operator) if it is not to be output in conformity with its type.

Maximum number of characters to be output:
With KR functionality (applies to C/C++ versions prior to V3.0 only) a maximum of 1400 characters can be output per sprintf call,
with ANSI functionality a maximum of 1400 characters per conversion element (e.g. %s).

The behavior is undefined if memory areas overlap.

Attempts to output non-initialized variables or to output variables in a manner inconsistent with their data type can lead to undefined results.

The behavior is undefined if the percent sign (%) in a format statement is followed by an undefined formatting or conversion character.

Example

You can use `sprintf` to copy a string, for example. It is thus possible to implement the `strncpy` function. The example under `strncpy` would then appear as follows:

```
#include <stdio.h>

int main(void)
{
   int n;
   char *s2 = "Peter is going swimming !";
   char s1[BUFSIZ];
   printf("The sentence is : %s \nCopy how many characters ?\n", s2);
   scanf("%d",&n);

               /* Alternatively, the following call
                  could appear at this point:
                  strncpy(s1,s2,n);        */

  sprintf(s1,"%.*s",n,s2);
  printf("%s \n",s1);
  return 0;
}
```

See also

printf, fprintf, snprintf, putc, putchar, puts, sscanf

sqrt - Square root

Definition

#include <math.h>

double sqrt(double x);

`sqrt` calculates the square root of a non-negative floating-point number x.

Return val.

`sqrt(x)`	if x is >= 0.
0	if x is negative. In addition, `errno` is set to EDOM (domain error, i.e. invalid argument).

Example

The following program calculates the square root of an input value x.

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

int main(void)
{
   double x;
   scanf("%lf", &x);
   printf("Square root of %g : %g\n", x, sqrt(x));
   printf("%d\n", errno);
   return 0;
}
```

See also

exp, pow, log, log10, hypot, sinh

srand - Initialize the random number generator

Definition

#include <stdlib.h>

void srand(unsigned int i);

`srand` is used to initialize the random number generator called by rand. `i=1` sets the random number generator to its default starting number.

Example

See example under `rand`

See also

rand

sscanf - Formatted input from a string

Definition

#include <stdio.h>

int sscanf(const char *s, const char *format, argumentlist);

`sscanf` reads data (input fields) from a string *s*, converts the data according to specifications in the format string *format*, and stores the results in areas which you specify with the result pointers in the argument list.

`sscanf` works like `scanf`, except that the input fields are read from a string and not from the standard input (`stdin`).

Parameters

const char *s
String containing the input data. It should be terminated with the null byte (\0).

const char *format
Format string as described under `scanf` with KR or ANSI functionality (cf. `scanf`).

argumentlist
Pointers to variables in which `sscanf` is to store the converted results.
No pointer arguments may be specified for %* statements (skip assignment) in *format*.
There must be one pointer argument each for all other % statements. The data type of the pointer argument is determined by the type specification of the corresponding format statement.

Return val.

Number of input fields read and successfully converted.	This does not include any input fields for which %* (skip assignment) was specified.
EOF	if an error occurred before the start of the conversion.

Notes

The result is undefined if memory areas overlap.

A detailed description, notes, and examples relating to formatted input can be found under `scanf`.

See also

scanf, fscanf

_ _STDC_ _ - Test for compliance with ANSI standard

Definition _ _STDC_ _

This macro generates the value 1 for compilation with SOURCE-PROPERTIES=PARAMETERS(LANGUAGE-STANDARD=ANSI). In all other language modes of the compiler the value of this macro is undefined.

Note This macro does not have to be defined in an include file. Its name is recognized and replaced by the compiler.

_ _STDC_ VERSION_ _ - Test for compliance with Amendment 1

Definition _ _STDC _VERSION_ _

This macro is expanded to the decimal constant 199409L and thus indicates that the implementation complies with Amendment 1.

Note This macro does not have to be defined in an include file. Its name is recognized and replaced by the compiler.

strcat - Concatenate strings

Definition

#include <string.h>

char *strcat(char *s1, const char *s2);

`strcat` appends a copy of 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*. `strcat` terminates the string with the null byte (\0).

Return val. Pointer to the result string.

Notes

Strings terminated with the null byte (\0) are expected as arguments.

`strcat` does not check whether memory area *s1* is large enough for the result!

The behavior is undefined if memory areas overlap.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
  char text1[BUFSIZ];
  char text2[BUFSIZ];
  printf("Example of strcat - please enter 2 text lines!\n");
  if(scanf("%s %s", text1, text2) == 2)
  printf("%s\n", strcat(text1, text2));
  return 0;
}
```

See also strncat

strchr - First occurrence of a character in a string

Definition
#include <string.h>

char *strchr(const char *s, int c);

`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 not counted 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 `strchr` and `index` functions are equivalent.

The following two prototypes of the `strchr` function are applicable to C++:
const char *strchr(const char *s, int c);
char *strchr(char *s, int c);

Example
Find the first ’s’:

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

int main(void)
{
   char *s = "What fun in the ssun!";
   printf("%s\n", s);
   printf("Where is the mistake? %s\n", strchr(s, 's'));
   return 0;
}
```

See also
index, rindex, strrchr

strcmp - Compare two strings

Definition

#include <string.h>

int strcmp(const char *s1, const char *s2);

`strcmp` compares strings *s1* and *s2* lexically, e.g.:

"circle" is lexically less than "circular",
"bustle" is lexically greater than "bus".

Return val.

< 0	*s1* is lexically less than *s2*.
= 0	*s1* and *s2* are lexically equal.
> 0	*s1* is lexically greater than *s2*.

Note

Strings terminated with the null byte (\0) are expected as arguments.

Example

The following program searches the name list *list* for an input name:

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

char *list[] = {"anne", "peter", "walter", "john" };

int main(int argc, char *argv[])
{
  int j, i = 0;
  while((i <= 3) && (j = strcmp(argv[1], list[i++])));
  if (j == 0)
     printf("The candidate is already known!\n");
  else
     printf("This is a new candidate!\n");
     return 0;
}
```

See also

strncmp

strcoll - Compare two strings

Definition

#include <string.h>

int strcoll(const char *s1, const char *s2);

`strcoll` compares strings *s1* and *s2* lexically. The lexical sequence of the individual characters is interpreted according to the LC_COLLATE category of the current locale.

Return val.

< 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.

The locale concept is described in detail in chapter “Locale” on page 93.

Example

See under `strxfrm`.

See also

setlocale, strxfrm

strcpy - Copy string

Definition

#include <string.h>

char *strcpy(char *s1, const char *s2);

`strcpy` copies string *s2* (including the null byte (\0)) to string *s1*. *s1* must be long enough to accept string *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.

`strcpy` does not check whether *s1* is large enough for the result. If *s1* is less than *s2* (including the null byte), the result is a string that is not terminated with the null byte!

The behavior is undefined if memory areas overlap.

Example

The following program outputs the contents of *s1* and *s2*, then calls `strcpy` and outputs both contents again.

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

int main(void)
{
    char s1[] = "Anne is pretty !";
    char s2[] = "Mary too !";
    printf("Contents s1: %s\nContents s2: %s\n", s1, s2);

    strcpy(s1, s2); /* copy s2 to s1 */
    printf("After strcpy:\nContents s1: %s\nContents s2: %s\n", s1, s2);
    return 0;
}
```

See also

strncpy

strcspn - Compare strings and calculate segment length

Definition

#include <string.h>

size_t strcspn(const char *s1, const char *s2);

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*.

As soon as a character in *s1* matches a character in *s2*, the function is terminated and the segment length is returned.

If the first character in *s1* already matches a character in *s2*, the segment length is equal to 0.

Return val.

Integer	specifying the segment length (number of non-matching characters) starting from the beginning of string *s1*.

Note

Strings terminated with the null byte (\0) are expected as arguments.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
 char text1[40];
 static char text2[] = "/*#$&";
 size_t n;
 printf("Example of strcspn. Please enter a text line:\n");
 scanf("%s",text1);
 n = strcspn(text1, text2);
 printf("Length of initial segment without /, *, #, $, &: %d\n", n);
 return 0;
}
```

See also

strspn

strerror - Return error message text

Definition

#include <string.h>

char *strerror(int errnum);

`strerror` maps the error number passed in errnum to a message text that is language-dependent, and returns a pointer to this string.

The returned 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*.

Return val.

Pointer to an internal C memory area containing a string with the error message text.

Note

The area to which `strerror` points may not be modified by the program. It can be overwritten only by repeated `strerror` calls.

Example

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

int main(void)

{
     printf("Error message for EDOM: %s\n", strerror(EDOM));
     return 0;
}
```

See also

perror

strfill - Copy part of a string

Definition

`#include <string.h>`

`char *strfill(char *s1, const char *s2, size_t n);`

`strfill` copies a maximum of *n* characters from string *s2* to string *s1*.

The manner in which copying takes place is determined by the lengths and contents of strings *s1* and *s2* and the value specified for *n*.

1. Regardless of the length of string *s1*, *n* characters are always copied to *s1* (in all cases except case 5). In other words,
   - If *s1* contains more than *n* characters, the characters remaining at the right in *s1* are retained.
   - If *s1* contains fewer than *n* characters, *s1* is lengthened up to a length of *n*. In this case, *s1* is not automatically terminated with a null byte (cf. notes).
2. *s2* contains fewer than *n* characters:

   In addition to the characters copied from *s2*, the number of blanks required to achieve a total of *n* are added.
3. *s2* contains more than *n* characters:

   Only the first *n* characters from *s2* are copied.
4. *s2* is a null string:

   s1 is padded 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 check whether *s1* is large enough for the result and does not automatically terminate the result string with the null byte (\0)! To avoid an unpredictable result, you should explicitly terminate string *s1* with the null byte after each `strfill` call (see the example).

The behavior is undefined if memory areas overlap.

Example

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

int main(void)
{
    size_t n;
    char s1[10];
    char s2[10];
    printf("Please input 2 strings!\n");
    scanf("%s %s", s1, s2);
    printf("Copy how many characters?\n");
    scanf("%d", &n);
    strfill(s1, s2, n);
    /* strfill(s1, NULL, n);       Example of the transfer of s2 as a
                                   NULL pointer */
    *(s1 + n) = '\0';  /* Terminate result string with null byte */
    printf("s1 after strfill: %s\n", s1);
    printf("Current length of s1: %d\n", strlen(s1));
    return 0;
}
```

See also strncpy

strftime - Locale-specific representation of date and time

Definition #include <time.h>

```
size_t strftime(char *s, size_t max_n, const char *format,
                const struct tm *tm_p);
```

`strftime` writes a maximum of *max_n* characters according to the information in the *format* string to the area to which *s* points.
The format string consists of any ordinary characters and conversion characters (beginning with %). All ordinary characters, including the terminating null byte (\0), are transferred 1:1 to the string. The conversion characters are replaced by appropriate date/time information. This information is determined by the current locale (category LC_TIME) and the values of the structure to which *tm_p* points.

Parameters char *s

Result string. It must be large enough to take *max_n* characters, including the null byte.

size_t max_n

Maximum number of characters, including the null byte, to be written to the result string.

const char *format

Format string containing the ordinary characters and conversion characters. The conversion characters are replaced by locale-specific and current data as described below:

%a	Abbreviated locale-specific name of the weekday.
%A	Full locale-specific name of the weekday.
%b	Abbreviated locale-specific name of the month.
%B	Full locale-specific name of the month.
%c	Locale-specific representation of the time and date.
%d	Day of the month as a decimal number (01 - 31).
%H	Hour as a decimal number (00 - 23). 24-hour clock.
%I	Hour as a decimal number (00 - 12). 12-hour clock.
%j	Day of the year as a decimal number (001 - 366).
%m	Month as a decimal number (01 - 12).
%M	Minutes as a decimal number (00 - 59).
%p	Locale-specific equivalent for AM and PM.
%S	Seconds as a decimal number (00 - 59).
%U	Week number in the year (00 - 53). The first week starts with the first Sunday in the year.

%w	Weekday as a decimal number (0 - 6). Sunday is 0.
%W	Week number in the year (00 - 53). The first week starts with the first Monday in the year.
%x	Locale-specific date representation.
%X	Locale-specific time representation.
%y	Year without century as a decimal number (00 - 99).
%Y	Year with century.
%z	Name of the time zone or no character if the time zone cannot be determined.
%%	The character %.

const struct tm *tm_p

Pointer to a structure of type tm from which `strftime` can take the time and the date. A structure of type `tm` is returned by the `gmtime`, `gmtime64`, `localtime`, `localtime64`, `mktime` and `mktime64` functions.

Return val.

Number of characters written excluding the terminating null byte.

0 if an error occurs. If, for example, conversion produces more than *max_n* characters (including the null byte).

Notes

The behavior is undefined if memory areas overlap.

The available locales are described in chapter “Locale” on page 93.

See also

gmtime, gmtime64, localtime, localtime64, mktime, mktime64, setlocale

strlen - Determine length of a string

Definition #include <string.h>

size_t strlen(const char *s);

`strlen` determines the length of string *s*, excluding the terminating null byte (\0).

While the `sizeof` operator always returns the defined length, `strlen` calculates the number of characters currently in a string. A newline (\n) character is also included.

Return val. Length of the string *s*.
The terminating null byte is not counted.

Note Strings terminated with the null byte (\0) are expected as arguments.

Example 1 This program reads a string and calculates its current memory space requirements, taking into account the null byte (strlen + 1) as well as the defined length of the string (`sizeof(s)` = 8192 bytes).

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

int main(void)
{
  char s[BUFSIZ];
  printf("Please enter your string.\n");
  scanf("%s", s);
  printf("Memory space required for the string: %d\n", strlen(s)+1);
  printf("Memory space defined for the string: %d\n", sizeof(s));
  return 0;
}
```

Example 2 This program calculates the current record length (including the newline character ’\n’) for each record in a file.

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

int main(void)
{
  FILE *fp;
  int n = 200, z = 0;
  char string[BUFSIZ];
  fp = fopen("input", "r");

  while (fgets(string, n, fp) != NULL)
  {
    z++;
    printf("record %d contains %d characters \n", z, strlen(string));
  }
  return 0;
}
```

strlower - Copy a string and convert to lowercase letters

Definition #include <string.h>

char *strlower(char *s1, const char *s2);

`strlower` copies string *s2* (including the null byte (\0)) to string *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.

Return val. Pointer to the result string *s1*.

Notes Strings terminated with the null byte (\0) are expected as arguments.

`strlower` does not check whether *s1* is large enough for the result. If *s1* is shorter than *s2* (including the null byte), the memory space after *s1* is overwritten!

The behavior is undefined if memory areas overlap.

Example The following program copies the contents of *s2* to *s1*, converting uppercase letters to lowercase in the process.

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

int main(void)
{
    char s1[] = "          ";
    char s2[] = "UPPERCASE!";
    printf("Contents s2: %s\n", s2);

                /* Copy s2 to s1 and convert to lowercase */
    strlower(s1, s2);

    printf("After strlower:\ncontents s1: %s\n", s1);
    return 0;
}
```

See also strupper, tolower, toupper

strncat - Concatenate strings

Definition

```
#include <string.h>

char *strncat(char *s1, const char *s2, size_t n);
```

`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*.

Return val. Pointer to the result string.
`strncat` terminates the string with the null byte (\0).

Notes

Strings terminated with the null byte (\0) are expected as arguments.

`strncat` does not check whether *s1* is large enough for the result!

The behavior is undefined if memory areas overlap.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
  char text1[BUFSIZ];
  char text2[BUFSIZ];
  int n;
  printf("Example of strncat - please enter 2 text lines and n!\n");
  if(scanf("%s %s %d", text1, text2, &n) == 3)
     printf("%s\n", strncat(text1, text2, n));
  return 0;
}
```

See also strcat

strncmp - Compare two strings

Definition

#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);

`strncmp` compares strings *s1* and *s2* lexically up to a maximum length of *n*, e.g.

`strncmp("for","formula",3)`

returns 0 (equal), because the first three characters of both arguments match one another.

Return val.

< 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*.

Note

Strings terminated with the null byte (\0) are expected as arguments.

Example In the following guessing program, `strncmp` is used to determine the lexical order of two strings.

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

int main(void)
{
    int i, n, result;
    char s[BUFSIZ], w[BUFSIZ];
    printf("Please enter the word to be guessed:\n");
    scanf("%s", w);
    n = strlen(w);
    printf("\nThe word entered has %d letters.\n", n);
    i = 0;
    do
    {
       i++;
       printf("Your attempt: \n");
       scanf("%s", s);
       if (strlen(s) > n)
       {
          printf("Your input is too long!\n");
          continue;
       }
       result = strncmp(s, w, n);         /* result is assigned
                                             the result of strncmp */
       if (result > 0)
           printf("%s is lexically greater.\n", s);
       else
       {
         if (result < 0)
             printf("%s is lexically less.\n", s);
       }
    }
    while (result != 0);
    printf("Correct! The word was : %s\n", w);
    printf("You needed %d attempts.\n", i);
    return 0;
}
```

See also strcmp

strncpy - Copy string

Definition #include <string.h>

char *strncpy(char *s1, const char *s2, size_t n);

`strncpy` copies a maximum of *n* characters from string *s2* to string *s1*.

If string *s2* contains fewer than *n* characters, only the length of *s2* (`strlen` + 1) will be copied.

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.

Return val. Pointer to the result string *s1*.

`strncpy` does not automatically terminate *s1* with the null byte.

Notes `strncpy` does not check whether *s1* is large enough for 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 the case, for example, when only a segment of *s2* is being copied and *s2* does not contain a null byte either.

The behavior is undefined if memory areas overlap.

Example 1 The following program fragment copies the entire string *s2* to string *s1* (like the `strcpy` function).

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    int n;
    char s1[20];
    char s2[20];
    printf("Please enter s2 (max. 19 characters)\n");
    scanf("%s", s2);
    printf("s1: %s\n", strncpy(s1, s2, (strlen(s2) + 1)));
    return 0;
}
```

Example 2 This program copies only a segment (8 characters) of *s2* to *s1*. The result string is explicitly terminated with the null byte.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
  char *s1 = "                         ";
  char *s2 = "Peter is going swimming !";
  strncpy(s1, s2, 8);
  *(s1 + 8) = '\0';
  printf("s1: %s\n", s1);    /* Contents of s1: "Peter is" */
  return 0;
}
```

Example 3 In this example, only a segment (5 characters) of *s2* is copied to *s1*. The remaining data in *s1* is retained.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
  char *s1 = "James is going shopping !";
  char *s2 = "Peter is going swimming !";
  strncpy(s1, s2, 5);
  printf("s1: %s\n", s1);    /* Contents of s1: "Peter is going
                                 shopping !" */
  return 0;
}
```

See also strcpy, strlen

strpbrk - Search for a character in a string

Definition

<string.h> is included with #include <string.h>

char *strpbrk(const char *s1, const char *s2);

`strpbrk` searches string *s1* for the first character matching any character in string *s2*. 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.

The following two prototypes of the `strpbrk` function are applicable to C++:
const char *strpbrk(const char *s1, const char *s2);
char *strpbrk(char *s1, const char *s2);

Example

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

int main(void)
{
  char text1[40];
  static char text2[] = "0123456789";
  char *result;
  printf("Example of strpbrk()\n");
  printf("Please enter a string (max. 40 characters) !\n");
  scanf("%s",text1);
  result = strpbrk(text1,text2);
  if(result == NULL)
    printf("The entered string does not contain any digits.\n");
  else printf("%s\n", result);
  return 0;
}
```

See also

index, strchr

strptime - Convert a string into date and time

Definition #include <time.h>

char *strptime(const char *buf, const char *format, struct tm *tm);

`strptime` converts the string indicated by **buf* into individual date and time values which are stored in the structure indicated by **tm*.

Parameters const char *buf

Date and time string to be converted.

struct tm *tm

Result structure in which the converted individual date and time values are stored. The structure is not initialized with zeros by `strptime`. 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 or %W. 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 or %A .

const char *format

The *format* string contains none, one or more conversion directives. Each conversion directive comprises one of the following elements:
one or more white-space characters (as defined in `isspace`)
a standard character (neither % nor white-space character)
or a conversion specification.

Each conversion specification consists of a % sign followed by a conversion character which specifies the desired conversion. A white-space character or a non-alphanumeric character must appear between two conversion specifications.

The following conversion characters are supported:

%% Replaced by %

%a Day of the week, whereby the name from the locale is used. Either the abbreviated or full name can be specified.

%A Same meaning as %a

%b Month, whereby the name from the locale is used. Either the abbreviated or full name can be specified.

%B	Same meaning as %b
%c	Date and time display according to the definition in the locale.
%C	Century (four-digit year number divided by 100 as whole number) (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	Hour (00-23), 24-hour clock.
%I	Hour (01-12), 12-hour clock.
%j	Day of the year (001-366).
%m	Number of the month (01-12).
%M	Minute (00-59).
%n	Replaced by a white-space character.
%p	Equivalent identifier of the locale for AM or PM.
%r	Time in the format %I:%M:%S%p
%R	Time in the format %H:%M
%S	Seconds (00-61), permits leap seconds
%t	Replaced by a white-space character.
%T	Time in the format %H:%M:%S
%U	Number of the week in 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	Day of the week as a number (0-6), Sunday = 0.
%W	Number of the week in the year (00-53), Monday is the first day of week 1. All days before the first Monday of the year belong to week 0.
%x	Date as represented in the locale.
%X	Time as represented in the locale.
%y	Two-digit year number (00-99). Year numbers between 00 and 68 are interpreted as the years 2000 through 2068, while year numbers between 69 and 99 are interpreted as the years 1969 through 1999.
%Y	Four-digit year number in the form *ccyy* (e.g. 1966 or 2001).

A conversion directive comprising white-space characters is implemented by reading the input up to the first character that is not a white-space character (this character remains unread) or until no further characters exist.

A conversion directive comprising a standard character is implemented by reading the next character from the buffer. If the character read from the buffer does not match the character in the conversion directive, the action fails and the buffer character and all subsequent characters remain unread.

A sequence of conversion directives comprising %n, %t, white-space characters, and combinations thereof is implemented by reading up to the first character that is not a white-space character (this character remains unread) or until no further characters exist.

All other conversion specifications are implemented by reading all characters until a character matching the next conversion directive is read (this character remains in the buffer) or until no further characters exist. The characters that have been read are then compared with the values in the locale that correspond to the conversion specification. If the appropriate value is found in the locale, the corresponding structure elements of the `tm` structure are set to the values corresponding to this information.
The search is not case-sensitive if elements such as the names of days or months are being compared.
If no appropriate value is found in the locale, `strptime` fails and no further characters are read.

Return val. Pointer to the character behind the last character read
if successful

NULL pointer in all other cases

Note The special handling of white-space characters and many “identical formats” should make it easier to implement identical format strings for `strftime` and `strptime`.

See also scanf, strftime, time.

strrchr - Last occurrence of a character in a string

Definition #include <string.h>

char *strrchr(const char *s, int c);

`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 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 The `strrchr` and `rindex` functions are equivalent.

The following two prototypes of the `strchr` function are applicable to C++:
const char *strrchr(const char *s, int c);
char *strrchr(char *s, int c);

Example Find the last ’s’:

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

int main(void)
{
   char *s = "What fun in the ssun!";
   printf("%s\n", s);
   printf("Where is the mistake? %s\n", strrchr(s, 's'));
   return 0;
}
```

See also index, rindex, strchr

strspn - Compare strings and calculate segment length

Definition #include <string.h>

size_t strspn(const char *s1, const char *s2);

Starting at the beginning of string *s1*, `strspn` calculates the length of the segment that contains only characters from string *s2*.

As soon as a character in *s1* fails to match any character in *s2*, the function is terminated and the segment length is returned.

If the first character in *s1* already fails to match any character in *s2*, the segment length is equal to 0.

Return val. Integer value specifying the segment length (the number of identical characters) starting from the beginning of string *s1*.

Note Strings terminated with the null byte (\0) are expected as arguments.

Example

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

int main(void)
{
 char text1[40];
 char *text2 = "0123456789";
 size_t n;
 printf("Example of strspn. Please enter a text line:\n");
 scanf("%s", text1);
 n = strspn(text1, text2);
 printf("Length of initial segment with digits (0 - 9): %d\n", n);
 return 0;
}
```

See also strcspn

strstr - First occurrence of one string in another

Definition #include <string.h>

char *strstr(const char *s1, const char *s2);

`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*
if *s2* is contained in *s1*.

0 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 are expected as arguments.

The following two prototypes of the `strstr` function are applicable to C++:
const char *strstr(const char *s1, const char *s2);
char *strstr(char *s1, const char *s2);

Example

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

int main(void)
{
   char *s1 = "City: Munich, Name: Peter Mueller";
   char *s2 = "Peter";
   printf("Full name? %s\n", strstr(s1, s2));  /* Peter Mueller */
   return 0;
}
```

See also strchr

strtod - Convert a string into a floating-point number

Definition

#include <stdlib.h>

double strtod(const char *s, char **p);

`strtod` converts a string to which *s* points into a floating-point number of type `double`. The string to be converted may be structured as follows:

```
  ⎧tab⎫      ⎧+⎫                             ⎧E⎫ ⎧+⎫
[⎨   ⎬...][⎨ ⎬][digit...][.][digit...][⎨ ⎬[⎨ ⎬]digit...]
  ⎩ ␣ ⎭      ⎩−⎭                             ⎩e⎭ ⎩−⎭
```

Any control character for white space may be used for *tab* (see definition under `isspace`).

`strtod` also recognizes strings that start with a digit but end with any character. In such cases, `strtod` first truncates the numeric part and converts it to a floating-point value.

`strtod` additionally provides a pointer (**p*) to the first non-convertible character in string *s* via the second argument *p* of type char **. If no conversion is possible at all, **p* is set to the start address of string *s*.
However, this occurs only if *p* is not passed as a NULL pointer.

If *p* is a NULL pointer, `strtod` is executed like the `atof` function:

`strtod(s, (char **)NULL)` and `strtod(s, NULL)` are both equivalent to `atof(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. In addition, `errno` is set to ERANGE (result too large).

Note

The decimal point character (period or comma) in the string to be converted is determined by the locale (category LC_NUMERIC). The default setting is a period.

Example The following program converts a string passed during the call (Enter Options) into its corresponding floating-point number and outputs the first non-convertible character, if any.

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

int main(int argc, char *argv[])

               /* Numbers are passed as strings!!
                  A conversion is necessary if the
                  numeric value is required */
{
  char *p;

  printf("floating : %f\n", strtod(argv[1], &p));
  putchar(*p);
  return 0;
}
```

See also atof, atoi, atol, strtol, strtoul

strtok - Split a string into tokens

Definition

#include <string.h>

char *strtok(char *s1, const char *s2);

`strtok` can be used to split a complete string *s* into substrings called “tokens”, e.g. a sentence into individual words, or a source program statement into its smallest syntactical units.
The start and end criterion for each token are separator characters (delimiters), which you specify in a second string *s2*. Tokens may be delimited by one or more such delimiters or by the beginning and end of the entire string *s1*. Blanks, colons, commas, etc. are typical delimiters between the words of a sentence. A different delimiter sequence *s2* may be specified for each call or token.

`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. The `strtok` function terminates each token with the null byte (\0).

To ensure that `strtok` processes the entire string *s1* in succession, the start address, i.e. a pointer to *s1*, must only be passed in the first call. In all subsequent calls, *s1* must be passed as a NULL pointer.

Return val.

Pointer to the beginning of a token.
At the first call, a pointer to the first token; at the next call, a pointer to the following token, etc. `strtok` terminates each token in *s1* with a null byte (\0), each time overwriting the first delimiter it finds with \0.

NULL pointer — if no token, or no further token was found.

Example

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

int main(void)
{
  static char str[] = "?a???b,,,#c";
  char *t;;
  t = strtok(str, "?");         /* t points to the token "a" */
  t = strtok(NULL, ",");        /* t points to the token "??b" */
  t = strtok(NULL, "#,");       /* t points to the token "c" */
  t = strtok(NULL, "?");        /* t is a NULL pointer */
  return 0;
}
```

strtol - Convert a string into a whole number (long int)

Definition
#include <stdlib.h>

long int strtol(const char *s, char **p, int base);

`strtol` converts a string to which *s* points into an integer of type `long int`. The string to be converted may be structured as follows:

```
 ⎧tab⎫      ⎧+⎫  ⎧0 ⎫
[{    }...][{  }][{   }]digit...
 ⎩ ␣ ⎭      ⎩-⎭  ⎩0X⎭
```

All control characters for white space 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 start with convertible digits (including octal and hexadecimal digits) but then end with any character. In such cases, `strtol` first truncates the numeric part and converts it.

`strtol` additionally provides a pointer (*p) to the first non-convertible character in string *s* via the second argument *p* of type char **. However, this occurs only if *p* is not passed as a NULL pointer.

A third argument, *base*, defines the base (e.g. decimal, octal or hexadecimal) for the conversion.

Parameters
const char *s
Pointer to the string to be converted.

char **p
A pointer (*p) to the first character in *s* that terminates the conversion is returned if *p* is not a NULL pointer.
If no conversion is possible at all, *p is set to the start address of string *s*.

int base
Integer from 0 to 36, which is to be used as the base for the computation.

From base 11 to base 36, letters a (or A) to z (or Z) in the string to be converted are assumed to be digits with the corresponding values 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 and 0x are ignored after any sign in string *s*.

Return val. Integer value of type `long int`
for strings that have a structure as described above and 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.

LONG_MAX or LONG_MIN
depending on the sign.

ULONG_MAX
if the result overflows
`errno` is set to ERANGE to indicate the error.

Note If *p* is a NULL pointer and *base* is equal to 10, `strtol` is executed like the function `atol`: `atol(s)` is equivalent to `strtol(s, NULL, 10)`.

Example

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

int main(void)
{
  char *str1 = "  0x1ff";
  char *str2 = "h0***";
  char *end;
  long l;

  l = strtol(str1, &end, 0);          /* Base 16 is derived */
                                      /* from the string str1. */
  printf("First value: %ld\n", l);    /* 511 is output. */

  l = strtol(str2, &end, 20);         /* Base = 20 */
  printf("Second value: %ld\n", l);   /* 340 (17*20) is output. */
  printf("Rest of str2: %s\n", end);  /* "***" is output. */
  return 0;
}
```

See also atol, atoi, strtod, strtoll, strtoul, strtoull, wcstol, wcstoll, wcstoul, wcstoull

strtoll - Convert a string into a whole number (long long int)

Definition #include <stdlib.h>

long long int strtoll(const char restrict *s, char ** restrict p, int base);

`strtoll` converts a string to which *s* points into an integer of type `long long int`. The string to be converted may be structured as follows:

```
 ⎧tab⎫      ⎧+⎫  ⎧0 ⎫
[⎨   ⎬...][⎨ ⎬][⎨  ⎬]digit...
 ⎩ ␣ ⎭      ⎩-⎭  ⎩0X⎭
```

All control characters for white space 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 or hexadecimal digits) but then end with any character. In this case, `strtoll` first truncates the numeric part and converts it.

`strtoll` additionally provides a pointer to the first non-convertible character in string *s* via the second argument *p* of type char **. However, this occurs only if *p* is not transferred as a NULL pointer.

A third argument, *base*, defines the base (e.g. decimal, octal or hexadecimal) for the conversion.

Parameters const char *s

Pointer to the string to be converted.

char **p

A pointer (**p*) to the first character in *s* that terminates the conversion is returned if *p* is not a NULL pointer.
If no conversion is possible at all, **p* is set to the start address of string *s*.

int base

Integer from 0 to 36, which is to be used as the base for the computation.

From base 11 to base 36, letters a (or A) to z (or Z) in the string to be converted are assumed to be digits with the corresponding values 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 and 0x are ignored after any sign in string *s*.

Return val. Integer value of type `long long int`
for strings that have a structure as described above and 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
in the event of an overflow `errno` is set to ERANGE.

Notes If *p* is a NULL pointer and *base* is equal to 10, the only difference between `strtoll` and the function `atoll` lies in the error handling.

`atoll(s)` corresponds to `strtoll(s, (char **)NULL, 10)`.

See also atol, atoll, atoi, strtol, stroul, stroull, wcstol, wcstoll, wcstoul, wcstoull

strtoul - Convert a string into a whole number (unsigned long int)

Definition

#include <stdlib.h>

unsigned long int strtoul(const char *s, char **p, int base);

`strtoul` converts a string to which *s* points into an integer of type `unsigned long int`. The string to be converted may be structured as follows:

```
 ⎧tab⎫       ⎧0 ⎫
[⎨   ⎬...][⎨  ⎬]digit...
 ⎩ ␣ ⎭       ⎩0X⎭
```

All control characters for white space 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 start with convertible digits (including octal and hexadecimal digits) but then end with any character. In such cases, `strtoul` first truncates the numeric part and converts it.

`strtoul` additionally provides a pointer (**p*) to the first non-convertible character in string *s* via the second argument *p* of type `char` **. However, this occurs only if *p* is not passed as a NULL pointer.

A third argument, *base*, defines the base (e.g. decimal, octal or hexadecimal) for the conversion.

Parameters

const char *s

Pointer to the string to be converted.

char **p

A pointer (**p*) to the first character in *s* that terminates the conversion is returned if *p* is not a NULL pointer.
If no conversion is possible at all, **p* is set to the start address of string *s*.

int base

Integer from 0 to 36, which is to be used as the base for the computation.

From base 11 to base 36, letters a (or A) to z (or Z) in the string to be converted are assumed to be digits with the corresponding values 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 and 0x are ignored after any sign in string *s*.

Return val. Integer value of type `unsigned long`
for strings that have a structure as described above and 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

LONG_MAX, LONG_MIN
depending on the sign.

ULONG_MAX)

if the result overflows, `errno` is set to ERANGE (result too large).

See also atol, atoll, atoi, strtol, strtoll, stroull, wcstol, wcstoll, wcstoul, wcstoull

strtoull - Convert a string into a whole number (unsigned long long)

Definition

#include <stdlib.h>

unsigned long long int strtoull(const char restrict *s, char **restrict p, int base);

`strtoull` converts a 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⎭
```

All control characters for white space 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 start with convertible digits (including octal and hexadecimal digits) but then end with any character. In such cases, `strtoull` first truncates the numeric part and converts it.

`strtoull` additionally provides a pointer to the first non-convertible character in string *s* via the second argument *p* of type `char` **. However, this occurs only if *p* is not transferred as a NULL pointer.

A third argument, *base*, defines the base (e.g. decimal, octal or hexadecimal) for the conversion.

Parameters

const char *s

Pointer to the string to be converted.

char **p

A pointer (**p*) to the first character in *s* that terminates the conversion is returned if *p* is not a NULL pointer.
If no conversion is possible at all, **p* is set to the start address of string *s*.

int base

Integer from 0 to 36, which is to be used as the base for the computation.

From base 11 to base 36, letters a (or A) to z (or Z) in the string to be converted are assumed to be digits with the corresponding values 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 and 0x are ignored after any sign in string *s*.

Return val. Integer value of type `unsigned long long int`
for strings that have a structure as described above and 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
in the event of an overflow, `errno` is set to ERANGE.

See also atol, atoll, atoi, strtol, strtoll, stroul, wcstol, wcstoll, wcstoul, wcstoull

strupper - Copy a string and convert to uppercase letters

Definition #include <string.h>

char *strupper(char *s1, const char *s2);

`strupper` copies string *s2* (including the null byte (\0)) to string *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 long enough to accept *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 check whether *s1* is large enough for the result. If *s1* is shorter than *s2* (including the null byte), the memory space after *s1* is overwritten!

The behavior is undefined if memory areas overlap.

Example The following program copies the contents of *s2* to *s1*, converting lowercase letters to uppercase in the process.

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

int main(void)
{
    char *s1 = "          ";
    char *s2 = "lowercase!";
    printf("Contents s2: %s\n", s2);

               /* Copy s2 to s1 and convert to uppercase*/
    strupper(s1, s2);
    printf("After strupper:\ncontents s1: %s\n", s1);
    return 0;
}
```

See also strlower, tolower, toupper

strxfrm - Transform a string

Definition

```
#include <string.h>

size_t strxfrm(char *s1, const char *s2, size_t n);
```

`strxfrm` transforms the characters in string *s2* so that the lexical sequence of each character is interpreted according to the LC_COLLATE category of the current locale.
A maximum of *n* transformed characters (including the terminating null byte) are then copied to string *s1*.
If *n* has the value 0 then result string *s1* can be a NULL pointer.

A comparison of two strings transformed with `strxfrm` using the `strcmp` function will then return the same result as a comparison with the `strcoll` function applied to the same original strings.

Return val. Length of the transformed string (excluding the terminating null byte).

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* are indeterminate because 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, then this character terminates the transformed string as the null byte (see also section “User-specific locales” on page 110).

The behavior is undefined if memory areas overlap.

The locale concept is described in detail in chapter “Locale” on page 93.

Example

```
#include <stdio.h>
#include <string.h>
#include <locale.h>

int main(void)
{
  char alpha2[11];
  char num2[11];
  int comp1;
  int comp2;
  int comp3;
  size_t i = 11;
  char *alpha1 = "ABCDEFGHIJ";
  char *num1 = "0123456789";

  setlocale(LC_COLLATE, "ANNE");       /* Activate the user-specific
                                          locale, in which digits have
                                          a lower sorting value than
                                          letters */

  comp1 = strcoll(alpha1, num1);       /* Compare the original strings */
  if(comp1 > 0)                        /* using strcoll */
     printf ("alpha1 greater than num1\n");
  else printf("Fehler\n");

  comp2 = strcmp(alpha1, num1);        /* Compare the original strings */
  if(comp2 < 0)                        /* using strcmp */
     printf ("alpha1 less than num1\n");
  else printf("Error\n");

  strxfrm(num2, num1, i);              /* Transform with strxfrm */
  strxfrm(alpha2, alpha1, i);

  comp3 = strcmp(alpha2, num2);        /* Compare the transformed */
  if(comp3 > 0)                        /* result strings using strcmp */
     printf ("alpha2 greater than num2\n");
  else printf("Error\n");
  return 0;
  }
```

See also setlocale, strcoll, strcmp

swprintf - Formatted output to a wide character string

Definition #include <wchar.h>

int swprintf(wchar_t *s, size_t n, const wchar_t *format [, arglist]);

Description: see `fwprintf`.

swscanf - Formatted input from a wide character string

Definition #include <wchar.h>

int swscanf(const wchar_t *s, const wchar_t *format [, arglist]);

Description: see `fwscanf`.

system - Execute system command

Definition #include <stdlib.h>

int system(const char *cmd);

`system` executes the BS2000 system command in the string *cmd*.

Return val.

0	The system command was executed successfully (return value of the corresponding system command: 0).
-1	The system command was not executed successfully (return value of the system command: error code ≠ 0).

The return value remains undefined (see "Notes") if control is not returned to the program following the system command.

Notes The system command must not exceed a maximum length of 2048 characters and need not be specified with the system slash (/).

After certain commands (START-PROG, LOAD-PROG, CALL-PROCEDURE, DO, HELP-SDF), control is not returned to the calling program. If a program permits such premature program terminations, it should flush buffers (`fflush`) or close the files before the `system` call.

The `system` function passes on the *cmd* string as input to the BS2000 command processor MCLP without changing it (see also the "Executive Macros" manual). No conversion to uppercase is performed.

Example

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

int main(void)
{
  char cmd[225];
  int result;
  printf("Please enter system command\n");
  gets(cmd);
  result = system(cmd);
  printf("Return value: %d\n", result);
  return 0;
}
```

tan - Tangent

Definition

#include <math.h>

double tan(double x);

`tan` calculates the trigonometric function tangent within the permissible range of floating-point numbers. *x* specifies the angle in radians.

Return val.

`tan(x)` for any valid floating-point number *x*.

{+/-}HUGE_VAL
in the event of an overflow. In addition, `errno` is set to ERANGE (result too large).

Example

The following program outputs the tangent of an input number.

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

int main(void)
{
     double x;
     printf("Please enter a number :\n");
     scanf("%lf", &x);
     printf("The tangent of %g is %g \n", x, tan(x));
     return 0;
}
```

See also

sin, cos, tanh, atan

tanh - Hyperbolic tangent

Definition #include <math.h>

double tanh(double x);

`tanh` calculates the hyperbolic tangent function of *x*. *x* must be in the permissible range of floating-point numbers.

Return val. `tanh(x)` for a permissible floating-point number *x*.

Example The following program outputs the hyperbolic tangent of an input number.

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

int main(void)
{
     double x;
     printf("Please enter a number :\n");
     scanf("%lf", &x);
     printf("The hyperbolic tangent of %g is %g \n", x, tanh(x));
     return 0;
}
```

See also sin, cos, tan, atan

tell - Return current position of read/write pointer (elementary)

Definition #include <stdio.h>

long tell(int fd);

`tell` returns the current position of the read/write pointer for the file with file descriptor *fd*.

The `tell` function 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.

Return val. Position in the file if successful, i.e.

for binary files, the number of bytes that offsets the read/write pointer from the beginning of the file;
for text files, the absolute position of the read/write pointer.

-1 if an error occurs. In addition, corresponding error information is stored in `errno` (e.g. `tell` not permitted, number of blocks or records too large).

Notes The calls `tell(fd)` and `lseek(fd, 0L, SEEK_CUR)` are equivalent.

`tell` cannot be used for 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`:

1. SAM file

Record length	≤ 2048 bytes
Number of records/block	≤ 256
Number of blocks	≤ 2048

2. ISAM file

Record length	≤ 32 Kbytes
Number of records	≤ 32 K

Example See example under `lseek`.

See also lseek, lseek64, fseek, fseek64, ftell, ftell64

time, time64 - Get current time

Definition

```
#include <time.h>

time_t time(time_t *sec_p);
time64_t time64(time64_t *sec_p);
```

`time` and `time64` return the current time (local time) as the number of seconds that have elapsed since the reference date (epoch).

With `time` the reference date depends on the use of the TIMESHIFT bind option (see section “Time functions” on page 40):

- without TIMESHIFT bind option (default): 1/1/1950 00:00:00.
- with TIMESHIFT bind option: 1/1/1970 00:00:00.

With `time64` the reference date is always 1/1/1970 00:00:00.

When converting to summertime/wintertime, the value jumps by 3600 or -3600 seconds.

From 01/19/2018 03:14:08 (without TIMESHIFT bind option) or from 01/19/2038 03:14:08 (with TIMESHIFT bind option) `time` will issue the message `CCM0014` and terminates the program.

Irrespective of the use of the TIMESHIFT bind option, `time64` will supply correct results up to 3/18/4317 02:44:48.

Parameters

time_t *sec_p
time64_t *sec_p

Pointer to the result returned by time.

If a NULL pointer is passed as an argument, this parameter has no significance.

If no NULL pointer is passed, the result of `time` or `time64` is additionally entered into the area to which *sec_p* points.

Return val. Number of seconds that have elapsed since the reference date.

See also ctime, ctime64, difftime, difftime64, ftime, ftime64, mktime, mktime64

_ _TIME_ _ - Output compilation time (macro)

Definition _ _TIME_ _

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 does not have to be defined in an include file. Its name is recognized and replaced by the compiler.

Example

```
#include <stdio.h>

int main(int argc, char *argv[])

{
printf("Program %s was compiled on %s at %s hours\n", argv[0],
__DATE__, __TIME__);
return 0;
}
```

See also asctime, _ _DATE_ _

tmpfile, tmpfile64 - Open temporary binary file

Definition

```
#include <stdio.h>

FILE *tmpfile(void);
FILE *tmpfile64(void);
```

`tmpfile` and `tmpfile64` generate a unique file name (in an analogous manner to the `tmpnam` function) and open a binary SAM file with default attributes under this name. The file is opened in the `wb+` mode (write and read).

The file is automatically removed when the program terminates normally or when the file is closed.

There is no functional difference between `tmpfile` and `tmpfile64`, except that `tmpfile64` returns a file pointer to a temporary file that can be > 2 GB.

To process files > 2 GB, proceed as follows:

– If the `_FILE_OFFSET_BITS 64` define (see page 68) is set, call `tmpfile`. `tmpfile64` is then used implicitly with the appropriate parameters.

– Otherwise, you have to call `tmpfile64`.

Return val.

Pointer to the assigned FILE structure	if successful.
NULL pointer	if the file could not be opened.

Note

If the program is terminated abnormally with `abort` or `_exit(-1)`, the temporary files are not deleted.

See also

tmpnam, mktemp, abort

tmpnam - Generate unique temporary file name

Definition

```
#include <stdio.h>

char *tmpnam(char *s);
```

`tmpnam` generates a unique file name from the TSN number of the current task, an internal identifier, the time, the date, and a number of up to four digits. Each time `tmpnam` is called this number changes; so, too, does the time each time a second elapses. This ensures that the name is always different from the names of existing files.
`tmpnam` can be called at most TMP_MAX times.

The file name can then be used for creating any new file.

Return val.

Pointer to the generated name

If *s* is a NULL pointer, `tmpnam` writes the result to an internal C memory area which is overwritten with each call.
If *s* is not a NULL pointer `tmpnam` writes the result to the result string *s*. Sufficient memory to take at least `L_tmpnam` characters must be made available for *s*. `L_tmpnam` is defined in <stdio.h>.

0 — if `tmpnam` has been called more than TMP_MAX times.

Notes

`tmpnam` generates a maximum of TMP_MAX names. TMP_MAX is defined in the include file <stdio.h>.

Files opened with names generated by `tmpnam` are not automatically deleted at the end of the program or task. The files must be explicitly deleted (e.g. with `remove`).

Example

```
#include <stdio.h>

int main(void)
{
 FILE *fp1;
 FILE *fp2;
 char nam1[L_tmpnam];
 char nam2[L_tmpnam];

 tmpnam(nam1);
 printf("Name1: %s\n", nam1); /* Name1: S.C.UNQ.1RCP.00.13211.2709199.0000 */
 fp1 = fopen(nam1, "w+r");

 tmpnam(nam2);
 printf("Name2: %s\n", nam2); /* Name2: S.C.UNQ.1RCP.00.13211.2709199.0001 */
 fp2 = fopen(nam2, "w+r");

 fclose(fp1);
 fclose(fp2);

 remove(nam1);
 remove(nam2);
}
```

See also tmpfile, tmpfile64, mktemp, remove

toascii - Convert an integer value to a valid EBCDIC value

Definition #include <ctype.h>

int toascii(int i);

`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.

Return val. Value of the least significant byte of the variable *i*.

Notes `toascii` does not convert values from other character sets (e.g. ASCII on EBCDIC computers).

See also toebcdic

toebcdic - Convert an integer value to a valid EBCDIC value

Definition #include <ctype.h>

int toebcdic(int i);

`toebcdic` returns a legal value from the EBCDIC character set.

`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.

Return val. The least significant byte of the variable *i*.

Notes `toebcdic` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

`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 toascii

tolower - Convert uppercase letters to lowercase

Definition #include <ctype.h>

int tolower(int c);

`tolower` converts the uppercase letter *c* (from the EBCDIC character set) to the corresponding lowercase letter.

Return val. The lowercase letter corresponding to *c*
if *c* is an uppercase letter.

c unchanged if *c* is not an uppercase letter.

Note `tolower` is implemented both as a macro and as a function (see section “Functions and macros” on page 17).

Example The following program reads a string and converts the characters first to lowercase letters and then to uppercase letters. Characters that are neither uppercase nor lowercase letters (digits, special characters, etc.) remain unchanged.

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

int main(void)
{
  int i;
  char s[81];

  printf("Please enter a string (max. 80 characters)\n");
  scanf("%s", s);

  printf("And now everything in lowercase letters \n");
  for (i=0; s[i] != '\0'; ++i)
       if (isupper(s[i]))
           printf("%c", tolower(s[i]));
       else printf("%c", s[i]);

  printf("\n And in uppercase letters \n");
  for (i=0; s[i] != '\0'; ++i)
       if (islower(s[i]))
           printf("%c", toupper(s[i]));
       else printf("%c", s[i]);

  printf("\n");
  return 0;
}
```

See also strlower, strupper, toupper, toascii, toebcdic, towlower

toupper - Convert lowercase letters to uppercase

Definition #include <ctype.h>

int toupper(int c);

`toupper` converts the lowercase letter *c* to the corresponding uppercase letter.

Return val. The uppercase letter corresponding to *c*
if *c* is a lowercase letter.

c unchanged if *c* is not a lowercase letter.

Note `toupper` is implemented both as a macro and as a function (see section "Functions and macros" on page 17).

Example See example under `tolower`

See also strupper, strlower, tolower, toascii, toebcdic, towupper

towctrans - Map wide characters

Definition

#include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);

`towctrans` maps the wide character *wc* using the mapping described by *desc*. The current setting of the LC_CTYPE category must be the same as during the `towctrans` call that returned the value *desc*.

The following two calls to `towctrans` behave the same as the calls for conversion to lowercase and uppercase indicated in the comments that follow:

```
towctrans(wc, wctrans("tolower"))          /* towlower(wc) */
towctrans(wc, wctrans("toupper"))          /* towupper(wc) */
```

Return val.

Mapped wide character
if successful.

Note

This version of the C runtime system only supports one-byte characters as wide character codes.

See also

tolower, toupper, towlower, towupper, wctrans

towlower - Convert wide character to lowercase

Definition #include <wctype.h>

wint_t towlower(wint_t wc);

`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.

wc unchanged if *wc* is not an uppercase letter.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also setlocale, tolower, towupper

towupper - Convert wide character to uppercase

Definition #include <wctype.h>

wint_t towupper(wint_t wc);

`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.

wc unchanged if *wc* is not a lowercase letter.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also setlocale, toupper, towlower

ungetc - Push back a character to the buffer

Definition

#include <stdio.h>

int ungetc(int c, FILE *fp);

`ungetc` pushes the character *c* back to the buffer assigned to the file described by file pointer *fp*. The next read operation that reads one character at a time from this file (`getc`) will then return *c* once again.
If *c* is equal to EOF, `ungetc` has no effect, and EOF is returned.

Return val.

The pushed back character *c*
if successful.

EOF if `ungetc` cannot push back the character (due to an error or if *c* is EOF).

Notes

At least one character must always have been read from the file before the first `ungetc` call.

EOF cannot be pushed back.

After a successful `ungetc` call, the read/write pointer is moved back one character.

A call to one of the following functions cancels the effects of the `ungetc` call (e.g. backward positioning): `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `lseek`/`lseek64`, `rewind`, `fflush`.

If a character other than the one read previously is returned to the buffer, the result differs depending on whether KR or ANSI functionality is being used.
With KR functionality (applies to C/C++ versions prior to V3.0 only) the original data is changed when the buffer contents are written to the external file.
With ANSI functionality the original data is not changed when the buffer contents are written to the external file, i.e. the original data prior to the `ungetc` call is always written into the external file.

See also

getc, ungetwc

ungetwc - Push wide character back onto input stream

Definition

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

wint_t ungetwc(wint_t c, FILE *fp);
```

`ungetwc` pushes the wide character *c* back to the buffer assigned to the file described by file pointer *fp*. The next read operation that reads one character at a time from this file (`getwc`) will then return *c* once again.

One byte of pushback is guaranteed. This applies even if `ungetwc` directly follows a call for the formatted input of wide characters (`fwscanf` or `wscanf`). If `ungetwc` is called too many times on the same file without an intervening read or file-positioning operation on that file, the pushback operation may fail.

If the value of *c* is equal to the macro WEOF, the operation will fail, and WEOF is returned.

Return val.

Pushed back wide character
if successful.

WEOF if the function fails.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

WEOF cannot be pushed back.

After a successful `ungetwc` call, the read/write pointer is moved back one character.

A call to one of the following functions cancels the effects of the `ungetwc` call (e.g. backward positioning): `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `lseek`/`lseek64`, `rewind`, `fflush`.

If a character other than the one read previously is returned to the buffer, the original data is not changed when the buffer contents are written to the external file, i.e. the original data prior to the `ungetc` call is always written into the external file.

See also

getc, getwc, ungetc, ungetwc

unlink - Delete a file

Definition

```
#include <stdio.h>

int unlink(const char *f_name);
```

`unlink` continues to be supported for compatibility reasons; it works in the same way as the ANSI function `remove` (q.v.).

See also remove

va_arg - Process variable argument list

Definition #include <stdarg.h>

<type> va_arg(va_list arg_p, <type>);

Together with the `va_start` and `va_end` macros, the `va_arg` macro is used to process a list of arguments which may vary in number and type from function call to function call. A variable argument list is indicated in the formal parameter list of the function definition by ", ...".

The `va_arg` macro returns the data type and value of the next argument in a variable argument list, beginning with the first argument. Technically speaking, the macro expands into an expression of the data type and value of the argument.

Before `va_arg` is called for the first time, the variable argument list to which *arg_p* points must be initialized with `va_start`. Each time `va_arg` is called, *arg_p* changes so that the value of the next argument is made available.

Parameters va_list arg_p

Pointer to the argument list initialized with `va_start` before `va_arg` is called for the first time.

<type>

Type name matching the type of the current argument. All C data types are valid for which a pointer to an object of type *type* is defined by simply appending * to *type*. Array and function types, for example, are invalid.

Return val. Value of the argument

The first call after `va_start` is called returns the value of the first argument. This argument comes after the last “named” argument *parmN* in the formal parameter list (cf. `va_start`).
Subsequent calls return the remaining argument values in succession.

Undefined — The behavior is undefined if there is no next argument or <type> does not match the current argument.

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:
All unsigned types (including `char`) are represented as `unsigned int` (right-justified in a word).
All other integer types are represented as `int` (right-justified in a word).
`float` is represented as double (right-justified in a doubleword).

The macro `va_end` must be called before the return from a function whose argument list has been processed with `va_arg`.

Example

The *f1* function fills an array with a list of arguments which are of the type pointer to string. No more than MAXARGS arguments are to be processed. The number of pointer arguments is defined as the first argument for *f1*. The filled array is than passed to function *f2*.

```
#include <stdarg.h>
#include <stdio.h>
#define MAXARGS 20

extern int f2(int i, char *a[]);

void f1(int n_ptrs, ...)
{
   va_list ap;
   char *array[MAXARGS];
   int ptr_no = 0;

   if (n_ptrs > MAXARGS)
      n_ptrs = MAXARGS;
   va_start(ap, n_ptrs);
   while (ptr_no < n_ptrs)
      array[ptr_no++] = va_arg(ap, char *);
   va_end(ap);
   f2(n_ptrs, array);
   return 0;
}
```

See also

va_start, va_end

va_end - Terminate variable argument list

Definition #include <stdarg.h>

void va_end(va_list arg_p);

Together with the `va_start` and `va_arg` macros, the `va_end` macro is used to process a list of arguments which may vary in number and type from function call to function call. A variable argument list is indicated in the formal parameter list of the function definition by ", ...".

`va_end` performs termination activities on variable argument list *arg_p*. The macro must be called before the return from a function whose argument list has been processed with `va_start` and `va_arg`.

`va_end` may change argument list *arg_p* so that it can no longer be used. If it is to be used again, therefore, the argument list must be re-initialized with `va_start`.

Example See under `va_arg`

See also va_arg, va_start

va_start - Initialize variable argument list

Definition

#include <stdarg.h>

void va_start(va_list arg_p, parmN);

Together with the `va_arg` and `va_end` macros, the `va_start` macro is used to process a list of arguments which may vary in number and type from function call to function call. A variable argument list is indicated in the formal parameter list of the function definition by ", ...".

`va_start` must be called before an unnamed argument is accessed for the first time. The macro initializes variable argument list *arg_p* for subsequent `va_arg` and `va_end` calls.

Parameters

va_list arg_p
  Pointer to the argument list.

parmN
  Name of the last "named" parameter in the formal parameter list of the function definition. This is the parameter which is followed by ", ...". Functions which process variable argument lists must define at least one named parameter.

  parmN must not be of type register, function or array.

Notes

The behavior is undefined if *parmN* has an invalid data type or if the data type does not match the current argument.

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:
All unsigned types (including `char`) are represented in the same way as `unsigned int` (right-justified in a word).
All other integer types are represented in the same way as `int` (right-justified in a word).
`float` is represented in the same way as double (right-justified in a doubleword).

Example

See under `va_arg`

See also

va_arg, va_end

vfprintf - Formatted output to a file

Definition #include <stdio.h>

int vfprintf(FILE *fp, const char *format, va_list arg);

`vfprintf` is similar to the `fprintf` function. In contrast to `fprintf`, `vfprintf` enables arguments to be output whose number and data types are not known at compilation time. `vfprintf` is used within functions to which the caller can pass a different format string and different arguments for output. The formal parameter list of the function definition provides for a format string *format* and a variable argument list ", ..." for this purpose.
format is a format string as described under `printf` with ANSI functionality (see `printf`).

`vfprintf` steps through an argument list *arg* with internal `va_arg` calls and writes the arguments according to format string *format* to the file with file pointer *fp*. Variable argument list *arg* must be initialized with the `va_start` macro before `vfprintf` is called.

Return val. Number of characters output
if successful.

Integer< 0 if an error occurs.

Notes `vfprintf` 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.

`vfprintf` does not call the `va_end` macro. Since `vfprintf` uses the `va_arg` macro, the value of *arg* is undefined on return.

Example In the following program extract the `vfprintf` function outputs different types of information each time the error routine *error* is called.

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

void error(char *f, ...);
int main(void)
{
  .
  .
  char *weight = "WARNING";
  int num = 20;
  error("Error class: %s, Number: %d\n", weight, num);
  .
  .
  error("No error\n");
  .
  .
}

void error(char *format, ...)
{
  va_list arg;
  va_start(arg, format);
  vfprintf(stderr, format, arg);
  va_end (arg);
}
```

See also vprintf, vsprintf, vsnprintf

vfwprintf - Formatted output of wide characters

Definition #include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwprintf(FILE *fp, const wchar_t *format, va_list arg);

Description: see `fwprintf`.

vprintf - Formatted output to the standard output

Definition

#include <stdio.h>

int vprintf(const char *format, va_list arg);

`vprintf` is similar to the `printf` function except that, unlike `printf`, `vprintf` permits the output of arguments whose number and data types are not known at compilation time. `vprintf` is used within functions to which the caller can pass a different format string and different arguments for output each time. The formal parameter list of the function definition provides for a format string *format* and a variable argument list ", ..." for this purpose. *format* is a format string as described under `printf` with ANSI functionality (see `printf`).

`vprintf` successively steps through an argument list *arg* using internal `va_arg` calls and writes the arguments according to format string *format* on the standard output `stdout`. The variable argument list *arg* must be initialized with the `va_start` macro before `vprintf` is called.

Return val.

Number of characters 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 on return.

Example

See under `vfprintf`

See also

vfprintf, vsprintf, vsnprintf

vsnprintf - Formatted output to a string

Definition

#include <stdarg.h>
#include <stdio.h>

int vnsprintf(char *s, size_t n, const char *format, va_list arg);

`vsnprintf` is similar to the `snprintf` function. In contrast to `snprintf`, `vsnprintf` enables arguments to be output whose number and data types are not known at compilation time.
`vsnprintf` is used within functions to which the caller can pass a different format string and different arguments for output each time. The formal parameter list of the function definition provides for a format string *format* and a variable argument list ", ..." for this purpose.

`vsnprintf` successively steps through an argument list *arg* using internal `va_arg` calls and writes the arguments according to format string *format* to string *s*. The variable argument list *arg* must be initialized with the `va_start` macro before `vsprintf` is called.

`vsnprintf` only outputs up to the buffer limit specified by the n parameter. This prevents buffer overrun.

Parameters

char *s
Pointer to the result string. `vsprintf` terminates the string with the null byte (\0). The maximum length of the output is therefore *n*-1.

size_t n
Length of the area reserved for the result string. *n* may not be greater than `INT_MAX`. When *n* = 0, no output takes place.

const char *format
Format string as described under printf with ANSI functionality (cf. `printf`).

The only difference is the way in which the control characters for white space (\n, \t, etc.) are handled: `vsprintf` enters the value of the control character into the result string. It is only during output to text files that the control characters are converted to their appropriate effect depending on the type of text file (see section “White space” on page 65).

va_list arg
Pointer to the variable argument list initialized with `va_start`.

Returnwert	< 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.

Notes

`vsnprintf` 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.

`vsnprintf` does not call the `va_end` macro. Since `vsnprintf` uses the `va_arg` macro, the value of *arg* is undefined on return.

The behavior is undefined if memory areas overlap.

See also vfprintf, vprintf , vsprintf

vsprintf - Formatted output to a string

Definition

#include <stdarg.h>
#include <stdio.h>

int vsprintf(char *s, const char *format, va_list arg);

`vsprintf` is similar to the `sprintf` function. In contrast to `sprintf`, `vsprintf` enables arguments to be output whose number and data types are not known at compilation time. `vsprintf` is used within functions to which the caller can pass a different format string and different arguments for output each time. The formal parameter list of the function definition provides for a format string *format* and a variable argument list ", ..." for this purpose.

`vsprintf` successively steps through an argument list *arg* using internal `va_arg` calls and writes the arguments according to format string *format* to string *s*. The variable argument list *arg* must be initialized with the `va_start` macro before `vsprintf` is called.

Parameters

char *s
Pointer to the result string. `vsprintf` terminates the string with the null byte (\0).

const char *format
Format string as described under printf with ANSI functionality (cf. `printf`).

The only difference is the way in which the control characters for white space (\n, \t, etc.) are handled: `vsprintf` enters the value of the control character into the result string. It is only during output to text files that the control characters are converted to their appropriate effect depending on the type of text file (see section “White space” on page 65).

va_list arg
Pointer to the variable argument list initialized with `va_start`.

Return val.

Number of characters stored in *s*. The terminating null byte (\0) generated by `vsprintf` is not included in this total.

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 on return.

The behavior is undefined if memory areas overlap.

Example See under `vfprintf`

See also vfprintf, vprintf , vsnprintf

vswprintf - Formatted output of wide characters

Definition #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: see `fwprintf`.

vwprintf - Formatted output of wide characters

Definition #include <stdarg.h>
#include <wchar.h>

int vwprintf(const wchar_t *format, va_list arg);

Description: see `fwprintf`.

wcrtomb - Convert wide character to multibyte character

Definition

#include <wchar.h>

size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);

If *s* is a null pointer, `wcrtomb` is equivalent to the call
`wcrtomb(buf, L'\0', ps)`
where *buf* designates an internal buffer.

If *s* is not a null pointer, the `wcrtomb` function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by *wc* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most {`MB_CUR_MAX`} bytes are stored.
If *wc* is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state.

The resulting state described is the initial conversion state.

Return val.

`(size_t)-1` if *wc* is not a valid wide character. The value of the EILSEQ macro is stored in `errno`, and the conversion state is undefined.

Number of bytes written to the array pointed to by **s* otherwise.

Note

This version of the C runtime system only supports one-byte characters as wide character codes.

See also

mblen, mbtowc, wcstombs, wctomb

wcscat - Concatenate two wide character strings

Definition #include <wchar.h>

wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);

`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 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 This version of the C runtime system only supports one-byte characters as wide character codes.

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.

See also strcat, wcsncat

wcschr - Scan wide character string for wide characters

Definition #include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wchar_t wc);

`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 `wchar_t` type 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
This version of the C runtime system only supports one-byte characters as wide character codes.

The following two prototypes of the `wcschr` function are applicable to C++:
const wchar_t* wcschr(const wchar_t *ws, wchar_t wc);
wchar_t* wcschr(wchar_t *ws, wchar_t wc);

See also
strchr, wcsrchr

wcscmp - Compare two wide character strings

Definition
#include <wchar.h>

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);

`wcscmp` compares wide character strings *ws1* and *ws2* lexically.

Return val.

< 0	*ws1* is lexically less than *ws2*.
= 0	*ws1* and *ws2* are lexically equal.
> 0	*ws1* is lexically greater than *ws2*.

Notes
This version of the C runtime system only supports one-byte characters as wide character codes.

Wide character strings terminated with the null wide character (\0) are expected as arguments.

See also
strncmp, wcsncmp

wcscoll - Compare two wide character strings according to LC_COLLATE

Definition #include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);

`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.

< 0	*ws1* is less than *ws2* according to the defined collation sequence.
= 0	*ws1* and *ws2* are equal according to the defined collation sequence.
> 0	*ws1* is greater than *ws2* according to the defined collation sequence

If one of the two wide character strings cannot be converted into a multibyte string, `wcscoll` will fail, and `errno` is set to EINVAL.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

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 `wcsxfrm` and `wcscmp` functions should be used.

See also strcoll, wcsncmp, wcsxfrm

wcscpy - Copy wide character string

Definition #include <wchar.h>

wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);

`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 null wide character (\0).

Return val. Pointer to the resulting wide character string *ws1*.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

Wide character strings terminated with the null wide character (\0) are expected as arguments.
`wcscpy` does not verify whether *ws1* has enough space to accommodate the result!
The behavior is undefined if memory areas overlap.

See also strcpy, wcsncpy

wcscspn - Get length of complementary wide character substring

Definition #include <wchar.h>

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);

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 the wide character string *ws1*.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also strcspn, wcsspn

wcsftime - Convert date and time to wide character string

Definition #include <wchar.h>

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const wchar_t *format,
                const struct tm *timptr);
```

`wcsftime` writes wide character codes to the array 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 a maximum of *maxsize* wide character codes.

If copying is between overlapping objects, the result is undefined.

Return val.

Integer>0	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 contents of the array are undefined.

See also strftime, mbtowcs

wcslen - Get length of wide character string

Definition #include <wchar.h>

```
size_t wcslen(const wchar_t *ws);
```

`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 This version of the C runtime system only supports one-byte characters as wide character codes.

A wide character string terminated with the null wide character code (`\0`) is expected as the argument.

See also strlen

wcsncat - Concatenate two wide character substrings

Definition

#include <wchar.h>

wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);

`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 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*.

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 byte (`\0`).

Return val.

Pointer to the resulting wide character string *ws1*.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

Wide character strings terminated with the null wide character (\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.

See also

strncat, wcscat

wcsncmp - Compare two wide character substrings

Definition #include <wchar.h>

int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

`wcsncmp` compares the wide character strings *ws1* and *ws2* lexically up to a maximum length of *n*. For example:

Characters that follow the null wide character code are not included in the comparison.

Return val.		
	< 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 This version of the C runtime system only supports one-byte characters as wide character codes.

Wide character strings terminated with the null wide character (\0) are expected as arguments.

See also strncmp, wcscmp

wcsncpy - Copy wide character substring

Definition

#include <wchar.h>

wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);

`wcsncpy` copies a maximum of *n* characters from the wide character string *ws2* to the memory area pointed to by *ws1*. Characters that follow the null wide character code are not copied.

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

This version of the C runtime system only supports one-byte characters as wide character codes.

`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.

See also

strncpy, wcscpy

wcspbrk - Get first occurrence of wide character in wide character string

Definition

#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);

`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*
if successful.

NULL pointer if not a single match is present.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

Wide character strings terminated with the null wide character (\0) are expected as arguments.

The following two prototypes of the `wcspbrk` function are applicable to C++:

```
const wchar_t* wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
      wchar_t* wcspbrk(      wchar_t *ws1, const wchar_t *ws2);
```

See also

strpbrk, wcschr, wcsrchr

wcsrchr - Get last occurrence of wide character in wide character string

Definition #include <wchar.h>

```
wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);
```

`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 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

This version of the C runtime system only supports one-byte characters as wide character codes.

The following two prototypes of the `wcsrchr` function are applicable to C++:

```
const wchar_t* wcsrchr(const wchar_t *ws, wchar_t wc);
      wchar_t* wcsrchr(      wchar_t *ws, wchar_t wc);
```

See also strrchr, wcsch

wcsrtombs - Convert wide character string to multibyte character string

Definition #include <wchar.h>

size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);

`wcrtombs` converts a sequence of wide characters, beginning in the conversion state described by the object pointed to by **ps*, from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters. If *dst* is not a null pointer, the converted characters are stored into the array pointed to by *dst*. Each conversion takes place as if by a call to the `wcrtomb` function.

Conversion stops on encountering a terminating null character, which is also converted and stored in the array.

Conversion stops earlier in two cases:

– when a sequence of bytes that does not correspond o a valid multibyte character is encountered or

– if *dst* is not a null pointer, when the next multibyte character would exceed the maximum length *len* of the bytes to be stored in the array

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned one of the following values:

– a null pointer if conversion stopped due to reaching a terminating null character

– the address just past the last converted wide character (if any).

If *dst* is not a null pointer and if the conversion stopped due to reaching a terminating null character, the resulting state described is the initial conversion state.

Return val. `(size_t)-1` if a conversion error occurs, i.e. a sequence of bytes that do not correspond to a valid multibyte character are encountered. The value of the `EILSEQ` macro is stored in `errno`, and the conversion state is undefined.

Number of bytes in the converted multibyte string
otherwise. The terminating null character, if any, is not included in the count.

See also mblen, mbtowc, wcstombs, wctomb

wcsspn - Get length of wide character substring

Definition #include <wchar.h>

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);

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) at the beginning of the wide character string *ws1*.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

Wide character strings terminated with the null wide character (\0) are expected as arguments.

See also strspn, wcscspn

wcsstr - Find first occurrence of wide character string

Definition #include <wchar.h>

wchar_t *wcsstr(const wchar_t *ws1, const wchar_t *ws2);

`wcsstr` searches for the first occurrence of a wide character string *ws2* (without the terminating null byte) in the wide character string *ws1*.

Return val. Pointer to the start of the wide character string found
if *ws2* is found in *ws1*.

NULL pointer if *ws2* is not found in *ws1*.

ws1 if *ws2* is a null pointer.

Note The following two prototypes of the `wcsstr` function are applicable to 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 floating-point number (double)

Definition #include <wchar.h>

double wcstod(const wchar_t *nptr, wchar_t **endptr);

`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 split 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, and 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 LS_NUMERIC).

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 ERANGE to indicate the error.

Notes This version of the C runtime system only supports 1-byte characters as wide character codes.

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.

See also iswspace, localeconv, scanf, setlocale, strtod, wcstol

wcstok - Split wide character string into tokens

Definition

```
#include <wchar.h>

wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2, wchar_t **ptr);
```

`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`. The `wcstok` function stores the information necessary for it to continue scanning the same wide string in *ptr*.
In the second and all subsequent calls, a null pointer must be specified for *ws1*, and the value in *ptr* should match that stored by the previous call for the same wide string.

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.

Note — This version of the C runtime system only supports one-byte characters as wide character codes.

See also — strtok

wcstol - Convert wide character string to long integer

Definition #include <wchar.h>

long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);

`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 split 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 byte 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 digits 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.

LONG_MAX, LONG_MIN
depending on the sign of the value.

ULONG_MAX if the correct value is outside the range of representable values. `errno` is set to ERANGE to indicate the error.

Notes This version of the C runtime system only supports 1-byte characters as wide character codes.

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 `wcstol`, then check `errno`, and if it is non-zero, assume that an error has occurred.

See also iswalpha, iswspace, scanf, strtol, strtoll, strtoul, strtoull, wcstod, wcstoull

wcstoll - Convert a wide character string to a whole number (long long)

Definition
#include <wchar.h>

long long int wcstoll(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);

The first part of the wide character string, to which *nptr* points, is converted by `wcstoll` into the representation `long long int`. The input string of wide character codes is first split into three parts:

– an initial, possibly empty, sequence of white-space wide-character codes (as specified by `iswspace`),

– a 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 byte 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 digits 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 the error.

Notes This version of the C runtime system only supports 1-byte characters as wide character codes.

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 `wcstoll`, then check `errno`, and if it is non-zero, assume that an error has occurred.

The C compiler that supports the data type `long long` only creates objects in LLM format. For this reason, the `long long` library functions are also only available as LLMs and are not contained in the prelinked modules. Like data modules, they must either be integrated or reloaded from the library.

See also iswalpha, iswspace, scanf, strtol, strtoll, strtoul, strtoull, wcstod, wcstol, wcstoul

wcstombs - Convert wide characters to multibyte strings

Definition #include <stdlib.h>

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

`wcstombs` converts a sequence of wide characters (type `wchar_t`) in *pwcs* to the corresponding multibyte characters and stores these in string *s*. *n* indicates the maximum number of bytes to be stored in *s*.
n specifies the maximum number of bytes to be stored in *s*.

The assignment is terminated if

- the wide character 0 occurs in *pwcs*,
- *n* bytes have already been assigned or
- a wide character cannot be represented in one byte.

Return val. (size_t)-1 if a wide character cannot be converted to a multibyte character.

Number of assigned bytes
otherwise.

Notes If a wide character in *pwcs* cannot be converted to a multibyte character, the wide characters already converted are stored in *s*.

The behavior is undefined if memory areas overlap.

No characters consisting of multiple bytes are implemented in this version. Multibyte and wide characters always have a length of 1 byte. `wcstombs` converts each wide character in *pwcs* to a one-byte multibyte character and saves it in string *s*.

See also mblen, mbtowc, mbstowcs, wctomb

wcstoul - Convert wide character string to unsigned long

Definition #include <wchar.h>

unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);

`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 split 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 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 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.

LONG_MAX, LONG_MIN
: depending on the sign.

ULONG_MAX
: if the correct value is outside the range of representable values
`errno` is set to ERANGE to indicate the error.

Notes

This version of the C runtime system only supports one-byte characters as wide character codes.

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 `wcstol`, then check `errno`, and if it is non-zero, assume that an error has occurred.

See also

iswalpha, iswspace, scanf, strtol, strtoll, strtoul, stroull, wcstod, wcstol, wcstoll

wcstoull - Convert wide character string to whole number (unsigned long long)

Definition

```
#include <wchar.h>

unsigned long long int wcstoull(const wchar_t *restrict nptr, wchar_t **restrict endptr,
                                int base);
```

`wcstoull` 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 split 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 `unsigned 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 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 (see 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.

ULLONG_MAX
if the correct value is outside the range of representable values
`errno` is set to ERANGE to indicate the error.

Notes This version of the C runtime system only supports one-byte characters as wide character codes.

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 `wcstoull`, then check `errno`, and if it is non-zero, assume that an error has occurred.

The C compiler that supports the data type `long long` only creates objects in LLM format. For this reason, the `long long` library functions are also only available as LLMs and are not contained in the prelinked modules. Like data modules, they must either be integrated or reloaded from the library.

See also iswalpha, iswspace, scanf, strtoul, wcstod, wcstol

wcsxfrm - Transform wide character string

Definition

#include <wchar.h>

size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);

`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 < n	indicating the number of wide character codes written to the field (without terminating null).
Integer ≥ 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:

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

This version of the C runtime system only supports one-byte characters as wide character codes.

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 deter-mined 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.

See also

strxfrm, wcscmp, wcscoll

wctob - Convert wide character to (one-byte) multibyte character

Definition

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

int wctob(wint_t c);
```

The `wctob` function determines whether the character *c* corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

Return val.

`EOF` if no corresponding multibyte character with length one in the initial shift state exists for *c*.

Multibyte character, with a length of 1 byte, that corresponds to *c* otherwise.

See also mblen, mbtowc, wcstombs, wctomb

wctomb - Convert wide character to multibyte character

Definition

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wc);
```

`wctomb` converts the wide character *wc* to the corresponding multibyte character and stores this in string *s*.

No assignment is made if *s* is a NULL pointer or if the wide character cannot be represented in one byte.

Return val.

0	if *s* is a NULL pointer.
-1	if the wide character cannot be converted to a multibyte character.
1	otherwise.

Note

This version of the C runtime system only supports one-byte characters as wide character codes. Multibyte characters and wide character codes always have a length of 1 byte.

See also

mblen, mbtowc, mbstowcs, wcstombs

wctrans - Define mapping between wide characters

Definition #include <wctype.h>

wctrans_t wctrans(const char *property);

The `wctrans` function constructs a value with type `wctrans_t` that describes a mapping between wide characters identified by the string argument *property*.

The two strings listed in the description of the "`tolower`" and "`toupper`" functions shall be valid in all locales as *property* arguments to the `wctrans` function.

If *property* identifies a valid mapping of wide characters according to the LC_CTYPE category of the current locale, the `wctrans` function returns a non-zero value that is valid as the second argument to the `towctrans` function.

Return val.

Value ≠ 0	if *property* identifies a valid mapping.
0	otherwise.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also towctrans

wctype - Define wide character class

Definition #include <wctype.h>

wctype_t wctype(const char *charclass);

`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.

Return val.

0	if the character class name is not valid for the current locale (category LC_CTYPE).
≠ 0	An object of type `wctype_t` that can be used in calls to `iswctype` is returned.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also iswctype

wmemchr - First occurrence of wide character in wide character string

Definition #include <wchar.h>

wchar_t *wmemchr(const wchar_t *ws, wchar_t *wc, size_t n);

The `wmemchr` function searches for the 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 found position in *ws*.

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 one-byte characters as wide character codes.

The following two prototypes of the `wmemchr` function are applicable to C++:
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

Definition #include <wchar.h>

int wmemchr(const wchar_t *ws1, const wchar_t *ws2, size_t n);

`wmemcmp` lexically compares the first *n* bytes of the two wide character strings *ws1* and *ws2*.

Return val.

< 0	*ws1* is lexically less than *ws2.*
= 0	*ws1* and *ws2* sare lexically equal.
> 0	*ws1* is lexically greater than *ws2.*

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also memcmp, wcsstr, wmemchr, wmemcpy

wmemcpy - Copy wide character string

Definition #include <wchar.h>

wchar_t *wmemcpy(wchar_t *ws1, const wchar_t *ws2, size_t n);

`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*.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also memcmp, wmemmove, wmemset

wmemmove - Copy wide character string to memory with overlapping areas

Definition #include <wchar.h>

wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);

`wmemmove` copies the first *n* bytes of the wide character string *ws2* to the first *n* bytes of the wide character string *ws1*. Copying takes place as if the *n* wide characters are first copied to a temporary array that does not overlap the objects pointed to by *ws1* and *ws2* and then copied from this array to *ws1*.

Return val. Pointer to the wide character string *ws1*.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also memmove, wmemcpy, wmemset

wmemset - Set *n* wide characters in wide character string

Definition #include <wchar.h>

wchar_t *wmemset(wchar_t *ws, wchar_t *c, size_t n);

`wmemset` sets the first *n* wide characters in the wide character string *ws* to a value *c*.

Return val. Pointer to *ws*.

Note This version of the C runtime system only supports one-byte characters as wide character codes.

See also memset, wmemcpy, wmemmove

wprintf - Formatted output to standard output (wide character format)

#include <wchar.h>

int wprintf(const wchar_t *format [, arglist]);

Description: see `fwprintf`.

write - Write to a file (elementary)

Definition
#include <stdio.h>

int write(int fp, const char *buf, int n);

`write` is the elementary write operation.

`write` writes up to *n* contiguous bytes from the area to which *buf* points into the file with file descriptor *fd*.

SAM files are always processed as text files with elementary functions.

Parameters
int fd
File descriptor of the output file.

A file descriptor (positive integer) is the result of a successful `open`/`open64` or `creat`/`creat64` call.
File descriptors for `stdin` (0), `stdout` (1), and `stderr` (2) are assigned automatically when the program is started.

const char *buf
Pointer to the area containing the data to be written to the output file.

int n
Number of bytes to be written to the file. There is no guarantee that `write` will actually write *n* bytes!

Return val.
Number of bytes actually written
if successful.

-1
Nothing was written due to one of the following errors:
– physical I/O error
– *fd* is not a valid file descriptor
– the file is not present
– there is no access authorization or write permission for the file
– the area in which the data is located was not correctly specified.

Notes

After each `write` call, you should check the number of bytes actually written. If the result is smaller than the specification in *n*, there usually has been an error. If the result is greater than the specification in *n*, tab characters (\t) were written to a text file. In such cases, tab characters are converted to the corresponding blanks and counted in the number of bytes returned.

You should use the `sizeof` function to be sure that your specification in *n* does not exceed the size of the buffer.

The data is not written immediately to the external file but is stored in an internal C buffer (see section “Buffering” on page 63).

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” on page 65).

Example

The following program copies the standard input (file descriptor 0) to the standard output (file descriptor 1). If you utilize the redirection mechanism, you can use this program to copy from any source to any destination. BUFSIZ (8192 bytes) is defined in the include file <stdio.h>.

```
#include <stdio.h>

int main(void)
{
  char buf[BUFSIZ];
  int n;

  while((n = read(0, buf, sizeof(buf))) > 0)
        write(1, buf, n);
  return 0;
}
```

See also

read, open, open64, creat, creat64

wscanf - Read formatted input

#include <wchar.h>

int wscanf(const wchar_t *format [, arglist]);

Description: see `fwscanf`.

y0, y1, yn - Bessel functions of the second kind

Definition

```
#include <math.h>
double y0(double x);
double y1(double x);
double yn(int n, double x);
```

The functions `y0`, `y1` and `yn` calculate the Bessel functions of the second kind for real arguments x and the integer orders 0, 1 or n.

Return val. Bessel function for the real argument $x > 0$.

-HUGE_VAL for arguments ≤ 0. In addition, `errno` is set to EDOM (domain error, i.e invalid argument).

See also j0, j1, jn

8 Appendix

Overview of functions in BS2000/OSD and in the ANSI standard

The following pages list all the functions provided by the C runtime system. Whether or not a function is defined in the ANSI standard or is an extension is indicated in the lists as follows:

X	ANSI standard
-	Extension, declared in an ANSI-defined include header
A	AMENDMENT 1 to the ISO/IEC 9899:1990 Standard
o	Extension, declared in a BS2000-specific include header (no query of the define `_STRICT_STDC`, see page 41).

Function	ANSI
_a2e, _e2a	-
_a2e_dup, _e2a_dup	-
_a2e_dup_n, _e2a_dup_n	-
_a2e_max, _e2a_max	-
_a2e_n, _e2a_n	-
abort	X
abs	X
acos	X
alarm	-
asctime	X
asin	X
assert	X
atan	X
atan2	X
atexit	X
atof	X
atoi	X
atol	X
atoll	-
bs2cmd	-
bs2exit	-
bs2fstat	-
bsearch	X
btowc	A
cabs	-
calloc	X
cdisco	o
ceil	X
cenaco	o
clearerr	X
clock	X
close	-
cos	X

Function	ANSI
cosh	X
cputime	-
creat	-
creat64	-
cstxit	o
_cstxit	o
ctime	X
ctime64	-
_ _DATE_ _	X
difftime	X
difftime64	-
div	X
double2ieee	-
ecvt	-
_edt	-
erf, erfc	-
exit	X
_exit	-
exp	X
fabs	X
fclose	X
fcvt	-
fdelrec	-
fdopen	-
feof	X
ferror	X
fflush	X
fgetc	X
fgetpos	X
fgetpos64	X
fgets	X
fgetwc	A
fgetws	A

Function	ANSI
_ _FILE_ _	X
float2ieee	-
flocate	-
floor	X
fmod	X
fopen	X
fprintf	X
fputc	X
fputwc	A
fputws	A
fputs	X
fread	X
free	X
freopen	X
freopen64	-
frexp	X
fscanf	X
fseek	X
fseek64	-
fsetpos	X
fsetpos64	-
ftell	X
ftell64	-
ftime	o
ftime64	-
fwide	A
fwprintf	A
fwrite	X
fwscanf	A
gamma	-
garbcoll	-
gcvt	-
getc	X

Function	ANSI
getchar	X
getenv	X
getlogin	-
getpgmname	-
gets	X
gettsn	-
getw	-
getwc	A
getwchar	A
gmtime	X
gmtime64	-
hypot	-
ieee2double	-
ieee2float	-
index	-
isalnum	X
isalpha	X
isascii	-
iscntrl	X
isdigit	X
isebcdic	-
isgraph	X
islower	X
isprint	X
ispunct	X
isspace	X
isupper	X
iswalnum	A
iswalpha	A
iswcntrl	A
iswctype	A
iswdigit	A
iswgraph	A

Function	ANSI
iswlower	A
iswprint	A
iswpunct	A
iswspace	A
iswupper	A
iswxdigit	A
isxdigit	X
j0, j1, jn	-
kill	-
labs	X
ldexp	X
ldiv	X
_ _LINE_ _	X
llabs	-
lldiv	-
llrint	-
llrintf	-
llrintl	-
llround	-
llroundf	-
llroundl	-
localeconv	X
localtime	X
localtime64	-
log	X
log10	X
longjmp	X
lrint	-
lrintf	-
lrintl	-
lround	-
lroundf	-
lroundl	-

Function	ANSI
lseek	-
lseek64	-
malloc	X
mblen	X
mbrlen	A
mbrtowc	A
mbsinit	A
mbsrtowcs	A
mbstowcs	A
mbtowc	X
memalloc	-
memchr	X
memcmp	X
memcpy	X
memfree	-
memmove	X
memset	X
mktemp	-
mktime	X
mktime64	-
modf	X
offsetof	X
open	-
open64	-
perror	X
pow	X
printf	X
putc	X
putchar	X
puts	X
putw	-
putwc	A
putwchar	A

Function	ANSI
qsort	X
raise	X
rand	X
read	-
realloc	X
remove	X
rename	X
rewind	X
rindex	-
rint	-
rintf	-
rintl	-
round	-
roundf	-
roundl	-
scanf	X
setbuf	X
setjmp	X
setlocale	X
setvbuf	X
signal	X
sin	X
sinh	X
sleep	-
snprintf	-
sprintf	X
sqrt	X
srand	X
sscanf	X
_ _STDC_ _	X
_ _STDC_ _VERSION	A
strcat	X
strchr	X

Function	ANSI
strcmp	X
strcoll	X
strcpy	X
strcspn	X
strerror	X
strfill	-
strftime	X
strlen	X
strlower	-
strncat	X
strncmp	X
strncpy	X
strpbrk	X
strrchr	X
strspn	X
strstr	X
strtod	X
strtok	X
strtol	X
strtoll	-
strtoul	X
strtoull	-
strupper	-
strxfrm	X
swprintf	A
swscanf	A
system	X
tan	X
tanh	X
tell	-
time	X
time64	-
_ _TIME_ _	X

Function	ANSI
tmpfile	X
tmpfile64	-
tmpnam	X
toascii	-
toebcdic	-
tolower	X
toupper	X
towctrans	A
towlower	A
towupper	A
ungetc	X
ungetwc	A
unlink	-
va_arg	X
va_end	X
va_start	X
vfprintf	X
vfwprint	A
vprintf	X
vsnprintf	-
vsprintf	X
vswprintf	A
vwprintf	A
wcrtomb	A
wcscat	A
wcschr	A
wcscmp	A

Function	ANSI
wcscoll	A
wcscpy	A
wcscspn	A
wcsftime	A
wcslen	A
wcsncat	A
wcsncmp	A
wcsncpy	A
wcspbrk	A
wcsrchr	A
wcsrtombs	A
wcsspn	A
wcsstr	A
wcstombs	X
wctob	A
wcstod	A
wcstok	A
wcstol	A
wcstoll	A
wcstoul	A
wcstoull	A
wcsxfrm	A
wctomb	A
wctrans	A
wctype	A
wmemcmp	A
wmemcpy	A

KR or ANSI functionality for C/C++ versions prior to V3.0

The C library functions were provided for the first time with C V1.0. At the time there was no ANSI-defined C library set. The implementation was based on the “preliminary” definition by Kernighan/Ritchie or on the common UNIX/SINIX implementations.
Adapting C library functions to the ANSI standard (C V2.0) has led to a number of differences in the execution of some input/output functions compared with the predecessor version. In order to meet the requirements of the ANSI standard in full and at the same time to preserve the runtime behavior of "old-style" programs, the input/output functions affected by the differences are offered in two variants: with the new ANSI functionality or with the KR functionality compatible with C V1.0.

The desired functionality is selected at compilation time with the following compiler options:

```
SOURCE-PROPERTIES = PAR(LIBRARY-SEMANTICS = STD / V1-COMPATIBLE)
```

KR functionality (V1-COMPATIBLE) can only be selected in the KR and ANSI compilation modes. In STRICT-ANSI and CPLUSPLUS compilation modes the specification V1-COMPATIBLE is ignored and STD assumed automatically.

With the CPLUSPLUS language mode, the C library functions are usually executed with ANSI functionality.

The differences between KR and ANSI functionality are listed on the following pages.

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

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. Position of the read/write pointer in append mode

   If the read/write pointer in a file opened in append mode has been explicitly positioned away from end of file (with `rewind`, `fsetpos`, `fseek` or `lseek`), it is only ignored when writing with the elementary function `write` and automatically positioned to the end of the file.

   If a file has been opened in append mode and for reading, it is positioned at the end of the file after being opened. The old contents of existing files are preserved.

3. ISAM files (buffer flushing)

   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 is changed if a different character has been returned to the buffer instead of the character previously read in.

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 C V1.0 are used.

ANSI functionality

1. Default attributes of text files

   When a new text file is created, it is generated as an ISAM file with variable record length.

2. Position of the read/write pointer in append mode

   If the read/write pointer in a file opened in append mode has been explicitly positioned away from end of file (with `rewind`, `fsetpos`, `fseek` or `lseek`), it is ignored in all write functions and automatically positioned at end of file.

   If a file has been opened in append mode and for reading, it is positioned at the start of the file after being opened. The old contents of existing files are preserved.

3. ISAM files (buffer emptying)

   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 lengthens the record in the file. When an ISAM file is read, therefore, only newline characters explicitly written by the program are read in.

   If reading from any text file makes data transfer necessary from the external file to the internal C buffer, the data of all the ISAM files still stored in buffers is automatically written out to the files.

4. `ungetc`

   When the contents of the buffer are written to the external file, the original data is not changed if a different character has been returned to the buffer instead of the character previously read in. The original data existing prior to 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).

Related publications

You will find the manuals on the internet at *http://manuals.ts.fujitsu.com*. You can order manuals which are also available in printed form at *http://manualshop.ts.fujitsu.com*.

[1] **BS2000/OSD**
Softbooks English

[2] **CRTE (BS2000/OSD)**
Common RunTime Environment
User Guide

[3] **C Library Functions** (BS2000/OSD)
for POSIX Applications
Reference Manual

[4] **C** (BS2000/OSD)
C Compiler
User Guide

[5] **C/C++ V3.1A** (BS2000/OSD)
C/C++ Compiler
User Guide

[6] **BS2000/OSD-BC**
Executive Macros
User Guide

[7] **JV (BS2000/OSD)**
Job Variables
Reference Manual

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