HP C/iX Library Reference Manual

HP 3000 MPE/iX Computer Systems

Edition 4

HEWLETT® PACKARD

Manufacturing Part Number: 30026-90004
E1092

U.S.A. October 1992

Notice

Restricted Rights Legend

1 Introduction to the HP C/iX Library

HP C/iX has an extensive library of standard functions that are found in most implementations of the language. The functions provide facilities for such operations as input, output, mathematics, string manipulation, and time and date operations. The HP C/iX implementation provides a high degree of compatibility with the HP-UX standard library for the C language, and provides all the library functions required by the ANSI standard for the C language.

Organization of the HP C/iX Library

The HP C/iX library consists of several files that can be divided into three groups: standard library functions, mathematical library functions, and library functions available only on HP 3000 Series 900 computers.

If you are developing POSIX applications on MPE/iX, you will be using the POSIX/iX library. This library is separate from the HP C/iX library and is organized differently. The POSIX/iX library is described in the *MPE/iX Developer's Kit Reference Manual*.

The Standard Library

The standard library consists of the input/output functions, the general utility functions, and the program startup routines. All C programs must link in the standard library because it contains the startup routines necessary for program execution. Failure to link in this library results in a linker or loader error.

The standard C library functions are provided in three different forms: a relocatable library (`LIBC.LIB.SYS`), an executable library (`XL.PUB.SYS`), and a relocatable library that is suitable for adding to an executable library that you build (`LIBCXL.LIB.SYS`).

Each form of the library is intended for different uses. The relocatable form (`LIBC.LIB.SYS`) is for programmers who want the HP Link Editor/iX to bind copies of library functions to the application program at link time. This reduces the amount of dynamic binding that must be done at run time. However, this makes less efficient use of memory space because each application duplicates the library functions that it uses.

The executable forms of the library are placed into shared libraries that can be accessed when applications written in C are executed. System memory usage decreases when C programs access the shared library. Only one copy of each function is loaded into the computer's main memory. For example, if several applications use `printf`, only one copy of the actual `printf` executable code is loaded into memory. All applications share the same copy.

The files that contain the standard library are listed in <Undefined Cross-Reference>:

Table 1-1. Standard Library Files

File	Description
`XL.PUB.SYS`	The executable library (XL) form of the standard library, including executable libraries from other subsystems and languages.
`LIBCINIT.LIB.SYS`	The code necessary to initialize the XL form of standard library.
`LIBC.LIB.SYS`	The relocatable library (RL) form of the standard library functions.
`LIBCANSI.LIB.SYS`	The relocatable library that should be linked along with `LIBCINIT` or `LIBC` if conformance to ANSI C is desired.
`LIBCRAND.LIB.SYS`	The random number related functions, `rand` and `srand`, in RL form.

Table 1-1. Standard Library Files

File	Description
`LIBCWC.LIB.SYS`	The relocatable library that enables an HP C/iX program to expand valid file set *wildcards* into fully qualified permanent file names and pass them into the main program. See the *HP C/iX Reference Manual* for details.
`LIBCXL.LIB.SYS`	A relocatable library that is used to build the XL form of the standard library.
`LIBM.LIB.SYS`	The mathematical library functions in RL form.
`LIBMANSI.LIB.SYS`	An ANSI-conforming version of the mathematical library.

The executable form of the standard library in `XL.PUB.SYS` is automatically searched when any C program is executed. To use the XL form of the library, you must add the `LIBCINIT.LIB.SYS` file to the RL list when linking your program.

To use the relocatable form of the standard library, you must add the `LIBC.LIB.SYS` file to the RL list when linking your program. In this case, you must not specify the `LIBCINIT.LIB.SYS` file.

NOTE Either the `LIBCINIT.LIB.SYS` file or the `LIBC.LIB.SYS` file, but *not* both, must be specified in the RL list when linking a C program.

The relocatable library contained in `LIBCANSI` contains a flag that specifies that ANSI-conformant behavior is desired. Certain functions, primarily those that perform input/output operations on text files, interrogate this flag and behave accordingly. To specify ANSI-conforming behavior for the standard library functions, add `LIBCANSI.LIB.SYS` to the RL list when linking your program. If you want behavior consistent with pre-ANSI releases of the library, do not link with `LIBCANSI`. Normally, ANSI-conforming behavior should be requested for new programs.

If you use the `CCXLLK` command to compile and link a C program, or the `CCXLGO` command to compile, link, and run a C program, then `LIBCANSI.LIB.SYS` is automatically added to the RL list in the link step if you compiled in ANSI mode (`-Aa` option). If you compile and link with separate commands, specify `LIBCANSI` if you want ANSI-conforming behavior.

The `rand` and `srand` functions are not in the XL or RL versions of `LIBC.LIB.SYS`. To use the `rand` and `srand` functions, you must add the `LIBCRAND.LIB.SYS` file to the RL list when linking your program. There is no XL version of these functions. The special treatment for the `rand` and `srand` functions is due to a name conflict between the HP C/iX library function `rand` and the MPE/iX compiler library function `rand`.

The `LIBCXL.LIB.SYS` file facilitates building additional executable libraries containing the standard library and should not be linked into C programs.

The following examples link the standard library into a program. In the examples, assume that the object files `MYOBJ1` and `MYOBJ2` were created by the HP C/iX compiler and define the function `main`. To use the XL version of the standard library in a program called `MYPROG1`, specify the following `LINK` command:

```
LINK FROM=MYOBJ1; TO=MYPROG1; RL=LIBCINIT.LIB.SYS,
      LIBCANSI.LIB.SYS
```

To use the `rand` function with the RL version of the standard library in a program called `MYPROG2`, specify the following `LINK` command:

```
LINK FROM=MYOBJ2;TO=MYPROG2;RL=LIBCRAND.LIB.SYS,LIBC.LIB.SYS,
     LIBCANSI.LIB.SYS
```

For a complete specification of available functions, see chapter 5.

The Math Library

The math library consists of additional mathematical functions, such as trigonometric and logarithmic functions, that perform floating-point operations.

Two relocatable versions of the math library are provided. The library in `LIBMANSI.LIB.SYS` conforms to the ANSI standard for C and is the version you should use. The library in `LIBM.LIB.SYS` is compatible with older, pre-ANSI releases of the library, and is for programs that may be relying on non-ANSI behaviors.

The primary difference between the two libraries is in the way they handle an error, such as attempting to compute the square root of a negative value. The ANSI version of the library calls a user-written function named `_matherr` if one is provided; no error message is displayed. The older version of the library calls a user-written function named `matherr` if one is provided. Otherwise, an error message is displayed.

To use the math library, you must add the library file to your RL list when linking your program. The math library must precede the standard library (`LIBC.LIB.SYS`) in the RL list if the RL form of the standard library is used. The ordering of the files is significant because of the interdependencies between the libraries. The ordering is not significant if the XL form of the standard library is linked.

The following example links the math library into a program. In the example, assume the object file `MYOBJ` was created by the HP C/iX compiler and defines the function `main`. To use the ANSI-conforming math library and the RL version of the standard library in a program called `MYMATH`, specify the following link command:

```
LINK FROM=MYOBJ;TO=MYMATH;RL=LIBMANSI.LIB.SYS,LIBC.LIB.SYS,
     LIBCANSI.LIB.SYS
```

For a complete specification of math library functions, see chapter 5.

Other Library Functions

In addition to the standard library functions and the math library functions, HP C/iX provides another set of functions that perform miscellaneous tasks.

For a complete specification of available functions, see chapter 5.

Library Header Files

To use many of the facilities of the HP C/iX library, your C source code should include the preprocessing directive:

```
#include incfile.h>
```

Enclosing *incfile*.h in angle brackets tells the preprocessing phase of the compiler to look for that file in a standard location on the system, the H group of the SYS account for HP C/iX. For example, if you want to use the `fprintf` function, your program should specify:

```
#include stdio.h>
```

This includes the declaration of `fprintf`, as well as various types and variables used by the functions found in the `stdio.h` header file. The standard include file or files that are needed for each function are specified in the syntax descriptions provided in chapter 5. The order of inclusion of the header files using the `#include` directive makes no difference, and an error does not occur if you include the same header file more than once.

The following table lists and describes the HP C/iX library header files:

Table 1-2. HP C/iX Library Header Files

Header	Description
`<assert.h>`	Defines the `assert` macro.
`<ctype.h>`	Declares macros and external functions useful for testing and mapping characters.
`<errno.h>`	Declares error variables and defines macros useful for obtaining a more detailed description of a library function error.
`<fcntl.h>` [a]	Defines arguments to the `open` function.
`<float.h>`	Defines macros that describe the floating-point types.
`<limits.h>`	Defines several macros that represent basic C data type limits.
`<locale.h>`	Used for localization. Contains macro definitions, function, and type declarations needed to select the desired locale.
`<malloc.h>`	Declares memory management functions, `mallopt` argument functions, and a structure returned by the `mallinfo` function. Memory management functions are also declared in `<stdlib.h>`.
`<math.h>`	Contains declarations for the HP C/iX math library functions, as well as functions in the standard library that return floating-point values. Also defines the structure and constants used by the `matherr` error-handling mechanisms.
`<memory.h>`	Declares several functions useful for manipulating character arrays and other objects treated as character arrays. These functions are also declared in `<string.h>`.
`<mpe.h>`	Declares several types, constants, and functions that facilitate using the MPE/iX operating system interface. See chapter 4 for additional information.

Table 1-2. HP C/iX Library Header Files

Header	Description
`<search.h>`	Defines the types used with the `hsearch` and `tsearch` functions.
`<setjmp.h>`	Declares a type and several functions for bypassing the normal function call and return discipline.
`<signal.h>`	Contains declaration used in dealing with conditions that may be reported during program execution.
`<stdarg.h>`	Provides a standard method for dealing with variable arguments.
`<stddef.h>`	Defines several macros and types used widely in conjunction with the C library.
`<stdio.h>`	Defines a structure, several functions, and macros useful for I/O.
`<stdlib.h>`	Declares various general utility functions and macros.
`<string.h>`	Declares functions useful for manipulating character arrays and other objects treated as character arrays.
`<time.h>`	Declares types, global variables, and functions used for manipulating time.
`<times.h>` [b]	Contains the definition of the struct `tms`, which is used by some older non-ANSI library functions.
`<unistd.h>`	Defines macros that are used as arguments to the `lseek` function. These macros are also declared in `<stdarg.h>`.
`<values.h>`	Contains a set of manifest constants, conditionally defined for particular processor architectures.
`<varargs.h>`	Declares types and macros for declaring variable argument functions. See also `<stdarg.h>`.

a. These headers are not defined by the ANSI C standard. Programs using these headers are likely to be less portable.
b. These headers are not defined by the ANSI C standard. Programs using these headers are likely to be less portable.

Some of the ANSI-defined header files, as implemented in the HP C/iX library, contain declarations for entities beyond those required by ANSI C. For example, the header `<math.h>` contains a macro definition for `M_PI`, the value of the mathematical constant `pi`. Because `M_PI` is not a reserved identifier in C, it is possible that a legal C program might use that identifier for a different purpose. Consequently, it is important that the compiler not process such declarations when compiling in ANSI mode.

If you have a program that relies on a non-ANSI declaration in one of the standard header files, and if you want to compile in ANSI mode, you must explicitly request such declarations to be visible. You do so by adding the following directive to your source file before including the standard header file:

```
#define  _MPEXL_SOURCE
```

This tells the preprocessor that your source program needs the extensions present in MPE/iX. Alternatively, you could specify this using the compiler option `-D_MPEXL_SOURCE` when you invoke the compiler. For compatibility with previous releases, this directive is automatically issued for you if you are not compiling in ANSI mode.

2 HP C/iX Library Input and Output

This chapter discusses HP C/iX library input/output functions and streams.

HP C/iX Library Input and Output

The C language does not provide any direct facility to perform input or output. Instead, C implementations usually provide a set of functions that perform I/O. Care must be used when calling the I/O functions because the compiler does not ensure that the arguments to the functions are correct. Most errors in calls to I/O functions may not provide the correct results and may cause addressing violations and abnormal terminations.

HP C/iX provides two I/O facilities. You can call MPE intrinsics directly or call a set of supplied C functions that provide an interface that is transportable to other systems. Because most of the C functions are built *on top of MPE*, they may not execute as quickly as direct MPE calls. The added time in most cases is small and the resulting portability is usually well worth the extra execution time.

You should use either HP C/iX I/O functions or MPE/iX intrinsics, but not both, in the same program. Using HP C/iX I/O functions with MPE/iX intrinsics to operate on the same file can result in unpredictable program behavior. The HP C/iX I/O routines use data and file control buffers that are different from the ones used by the MPE/iX I/O intrinsics.

HP C/iX I/O and POSIX I/O functions utilize a **byte stream** model. The data is treated as a continuous stream of bytes. There are conceptually no record boundaries.

The HP C/iX I/O functions allow C programs to access any file type supported by MPE. These functions emulate stream I/O when accessing MPE (non-byte stream) files.

The performance of stream emulation suffers when accessing variable-length records with certain I/O functions. For example, the HP C I/O function `fseek` enables the caller to position a file pointer to any given byte number. In the case of files that are composed of fixed-sized records, the positioning operation is direct because the address of the record that contains the interesting byte can be calculated and the file pointer can be positioned to it. If a file has variable-length records, the `fseek` function still works, but the implementation requires the file to be rewound and then all records are read until the required record is reached. The `fseek` function is much slower with variable-length record MPE files than with fixed-length record MPE files.

Basic Stream Usage

Using a stream is similar to using an MPE file. A stream is opened by calling the C library function `fopen()`. The `fopen` function creates a data structure that contains descriptive data about a stream and returns a pointer to this structure. This pointer designates the stream in all further transactions.

When you use the `fopen` function to open an existing file, the fixed attributes of the file take precedence over the mode requested by `fopen()`. In particular, an existing ASCII file is opened as a text stream, and an existing binary file is opened as a binary stream, regardless of the mode requested by `fopen()`.

Three constant pointers that designate standard streams opened automatically by the C startup routines can also be used in further transactions. Refer to the section called "Standard Files" for details.

After a stream has been opened, you may read from it or write to it in several ways. Reading or writing can be done on a character-by-character basis using the inline macros `getc` and `putc`, or on a block-by-block basis using functions such as `fread` or `fwrite`. The macros `getchar` and `putchar` and the higher-level functions `fgetc`, `fgets`, `fprintf`, `fputc`, `fputs`, `fread`, `fscanf`, `fwrite`, `gets`, `printf`, `puts`, and `scanf` use, or act as if they use, `getc` and `putc`; they may be freely intermixed.

An open stream can also be controlled by functions such as `fseek` and `rewind`. These functions allow you to position the stream position indicator to an arbitrary byte.

When you are finished using a stream, the stream should be closed. This may be accomplished by issuing a call to `fclose`, implicitly by calling `exit`, or by returning from the `main` function. It is important to close all files because the `fclose` function causes information that is buffered in memory to be written out to the physical file. Calling an MPE termination routine does not properly close open streams.

Stream Types

The HP C/iX library supports two stream types: text and binary streams. The stream type dictates how special characters are to be processed and how records are to be padded. Streams created by the HP C/iX library default to text streams.

Text Streams

A **text stream** is an ordered sequence of characters composed into lines with each line consisting of zero or more characters plus a terminating newline (`\n`) character.

On MPE/iX, text streams are line-oriented fixed record length ASCII files. Text streams are usually the product of editor programs and are read directly without any interpretation by other functions. The newline character is not actually written to the file, but is used by the HP C/iX I/O functions to indicate when a buffer is full of information and should be posted to the file.

On input, newline characters are added to the records read from the file to make it appear as if the newline character is actually in the file. This is done to allow programs, such as `EDITOR`, to produce files that can be read by C functions in a manner compatible with other systems. ASCII files managed by MPE do not actually contain `\n` characters, but appear to when read by C functions. Further, the type of record structure used in the file has impact on what is seen when a file is read. Assume that there is an ASCII MPE file with variable-length records. If the following 5 bytes are written:

```
'A', 'B', 'C', 'D', '\n'
```

only 4 bytes are actually output to the MPE file. The record in the file appears as:

```
ABCD
```

The \n is used by the C functions to indicate that a record should be written, but the newline character is not actually written. When the same record is read back in, the \n is added to the end of the buffer. The result is that successive getc operations return the same five characters originally written out:

```
'A', 'B', 'C', 'D', '\n'
```

If the same example is examined with an ASCII file composed of fixed-length records, each of which is six characters in length, the result is different. Assume the same five characters are written. When the C I/O system encounters the `\n`, it gets ready to write out a record. The record contains "`ABCD`" at this point. However, since the record is a fixed-length record with a length of six characters, the two characters after the `'D` are padded with spaces (040). The record written to the MPE file is:

```
ABCD
```

where   indicates a space. As before, the `\n` triggers the write, but the actual `\n` character is not written out.

When the record is read back into the program, the `\n` is restored at the end of the record, but the I/O functions have no way of knowing whether the trailing spaces are pad characters written by the MPE/iX file system, or actual data characters written by a C program. This ambiguity is resolved in one of two ways (described below), depending on whether ANSI-conformant behavior has been requested.

The standard ANSI conformant interpretation, which is requested by including `LIBCANSI.LIB.SYS` in the `RL` list when linking (as described in chapter 1), is to discard all trailing spaces in fixed-length records when reading text streams. In the example given above, the result is that the following five characters are read back:

```
`A', `B', `C', `D', `\n'
```

However, for compatibility with previous releases of the HP C/iX library, the default behavior is that trailing spaces in fixed-length records in text files are not stripped. Thus, if the program in the example is not linked with `LIBCANSI`, it reads the following seven characters:

```
`A', `B', `C', 'D', space, space, `\n'
```

Because of the special meaning of the `\n` character, in text streams, you should avoid writing binary data to a text stream. If the binary data happens to contain a byte with the same numeric value as the newline character (ASCII code 10), the result is an unexpected record break.

Binary Streams

Like text streams, binary streams are also ordered sequence of bytes. Binary streams, however, transparently record data. No special attention is given to `\n` characters or any other characters. No padding is performed for binary streams.

If a `\n` character is written to a binary stream, it is actually written. Binary streams return the same number of characters originally written except in one special case.

On MPE/iX, if `fopen()` opens a binary stream it is a fixed-length record format file. If the file is closed, the last record in the file, if incomplete, is filled with trailing zeros. The end-of-file is located on a record boundary, regardless of the last byte written to the file.

By default, if you are using POSIX on MPE/iX, `fopen()` creates a byte stream file. If the file is closed, the last byte written is the end-of-file.

File Descriptors

To perform I/O operations, you must associate a stream with a file or device. For the unbuffered I/O operations (the ones in the standard I/O library), you do this by declaring a pointer to a structure type called FILE. The FILE structure, which is defined in <stdio.h>, contains several fields to hold information about the pointer to the buffer, the file descriptor, and the file access mode.

The FILE structures provide the operating system with bookkeeping information, but your only means of access to the stream is the pointer to the FILE structure (called a **file pointer**). The file pointer, which you must declare in your program, holds the stream identifier returned by the fopen function. You use the file pointer to read from, write to, or close the stream.

For unbuffered functions, you must associate a file with a file descriptor by using the open function. A **file descriptor** is a unique integer that identifies a particular file. This file descriptor is also contained in the FILE structure returned by fopen.

Standard Files

There are three constant pointers defined in stdio.h> that designate standard C streams. These streams are automatically opened by the C language startup routines. The standard stream designators are:

Table 2-1. Standard Stream Designators

Stream	Function	Default
stdin	Standard Input	$STDINX
stdout	Standard Output	$STDLIST
stderr	Standard Error	$STDLIST

The stdin stream is opened for reading. Your program only receives data from the stdin stream. It cannot write data to this stream. The stdin stream defaults to the standard MPE file $STDINX. If you run your program in interactive mode, the input device is normally a keyboard.

The stdout stream is opened for writing. Your program only outputs data to the stdout stream. It cannot read data from this stream. The stdout stream defaults to the standard MPE file $STDLIST. If you run your program in an interactive mode, the output device is normally your terminal.

The stderr stream is also opened for writing. Your program cannot read data from this stream. Like stdout, this stream defaults to the standard MPE file $STDLIST. For interactive programs, this file is normally your terminal. The stderr stream is used to print error and warning messages when an erroneous condition is detected in your program. The stderr stream is unbuffered by default. An unbuffered stream transfers data to its destination one byte at a time.

Reading from stdin in Interactive and Batch Modes

When reading from stdin in interactive mode using fgets, gets, fscanf, scanf, or fread, the input text stream is not padded with trailing blanks.

When reading from stdin in batch mode using fgets, gets, fscanf, scanf, or fread, the input stream may or may not be padded with trailing blanks before being terminated with a null character. Whether or not padding is applied depends on the file type of the input batch stream file.

If the batch stream file is a variable record length ASCII file, no padding is applied and reading from stdin in batch behaves the same as reading in interactive mode.

If the batch stream file has fixed record lengths, the input records are padded with trailing blanks. When reading from fixed record length batch stream files, be sure to use large enough buffers to accommodate the entire record, including the null character appended to the string by the I/O system.

Whether using fixed or variable record length ASCII files, insert the EOD command in the batch stream file between the embedded program data and the next MPE/iX command. This prevents the program from accidentally reading command lines from the file.

For example, given the following program

```
#include stdio.h>
main (void)
{
        char iobuff[81];
        printf("\n Please enter your name:");
        gets(iobuff);
        printf("%s\n",iobuff);
}
```

a batch job to run this program is:

```
!JOB WALTER.JONES
!RUN ECHONAME
Walter Morgan
!EOD
!SHOWTIME
!EOJ
```

Restrictions

Due to the implementation of the HP C/iX library and the MPE/iX file system, operations on certain types of files may be restricted. Refer to appendix B, "Restrictions and Special Considerations," for more information.

3 Interfacing with MPE/iX

The `mpe.h>` header file provides several facilities that allow you to more easily interface with the MPE/iX operating system. Note that calling MPE/iX directly from C programs makes the C programs less portable than using the standard C library.

For information about interfacing with MPE/iX intrinsics from the POSIX/iX library, refer to the *MPE/iX Developer's Kit Reference Manual*.

Foptions

The structure tag fop names a structure that describes the bit positions of the MPE/iX FOPEN intrinsic's foptions. The structure is:

```
struct fop {
    unsigned short reserved:2;      /* for MPE/iX */
    unsigned short typer:3;         /* file type */
    unsigned short no_f_equ:1;      /* no file equations */
    unsigned short label:1;         /* labeled tape option */
    unsigned short carriage:1;      /* carriage control needed */
    unsigned short format:2;        /* record format */
    unsigned short designator:3;    /* default designator */
    unsigned short ascii:1;         /* ASCII(1)/binary(0) */
    unsigned short domain:2;        /* file domain */
};
```

In addition to the fop structure, mpe.h> contains a typedef called *foptions* that is the union of an unsigned short and an fop structure. The typedef is:

```
typedef union {
    struct fop fs;
    unsigned short fv;
} foptions;
```

This typedef is useful for declaring regions of storage that are to serve as foptions. If a variable f is declared as being type foptions, then f.fv accesses the unsigned short version of the foptions while f.fs accesses the structural definition of the foptions. For example:

```
#include mpe.h>
#pragma intrinsic FOPEN MPE_FOPEN

main
{
  foptions f;         /* declare f as an foption variable */
   . . .
  f.fv = 0;                   /* clear all options to 0 */
  f.fs.ascii = 1;             /* set ASCII foption to true */
  f.fs.no_f_equ = 1;          /* disallow file equations */
  MPE_FOPEN(. . , f.fv, . . .); /* pass foptions */
   . . .
}
```

Note, in the above example, the foptions variable can be accessed as named bit-fields using the f.fs construct or as a 16-bit unsigned
short value using the f.fv construct. Also, notice the FOPEN intrinsic has been given the name MPE_FOPEN in this example to avoid confusion with the C library function fopen.

Aoptions

Similar to the fop structure, the aop structure provides access to the aoptions used in FOPEN. It is defined as:

```
struct aop {
    unsigned short reserved:3;      /* reserved for MPE/iX */
    unsigned short copy:1;          /* copy open mode */
    unsigned short no_wait:1;       /* I/O without wait */
    unsigned short multi:2;         /* multi-access mode */
    unsigned short no_buf:1;        /* no buffering */
    unsigned short exclusive:2;     /* exclusive access flag */
    unsigned short locking:1;       /* allow locking */
    unsigned short multirecord:1;   /* multi-record flag */
    unsigned short access:4;        /* mode of access */
};
```

Also, a typedef that defines a union type named aoptions is provided:

```
typedef union {
    struct aop as;
    unsigned short av;
} aoptions;
```

If variable a is declared as being type aoptions, then a.av accesses the unsigned short version of the aoptions while a.as accesses the structural definition of the aoptions.

Condition Codes

Many MPE/iX intrinsics return condition code information upon their completion. The condition codes are defined with macros in the `mpe.h` header file. The macros are as follows:

Table 3-1. Standard Stream Designators

`CCG`	Condition Code Greater
`CCE`	Condition Code Equal
`CCL`	Condition Code Less

These macros may be used with the `ccode` function also declared in this header file.

MPE/iX File Numbers

The values returned by HP C/iX library functions such as `fopen` and `open` do not represent values that are meaningful to the MPE/iX file system intrinsics. The function `_mpe_fileno`, declared in `mpe.h`, maps a file descriptor returned by a C library function such as `open` into an MPE/iX file number that can be passed directly to MPE/iX file system intrinsics. If `fd` is a file descriptor returned by the C library function `open`, `_mpe_fileno(fd)` returns the associated MPE/iX file number. For a description of `_mpe_fileno`, see chapter 5.

The `_mpe_fileno` function is not supported in the POSIX/iX library. However, equivalent functionality is provided by the `_MPE_FILENO` macro. The `_MPE_FILENO` macro is described in the *MPE/iX Developer's Kit Reference Manual*.

4 HP C/iX Library Header Descriptions

This chapter describes the contents of the header files provided with the HP C/iX library.

Some of the header files described in this chapter contain extensions required for conformance with the POSIX standard. These POSIX extensions are not documented in this manual. Refer to the *MPE/iX Developer's Kit Reference Manual* for additional information about these extensions. The POSIX extensions require the `_POSIX_SOURCE` preprocessor macro definition. This macro should not be defined if you are using the HP C/iX library.

Available Header Files

The HP C/iX library is divided into sections. Each section has a header file that defines the objects found in that section of the library.

The standard ANSI C headers are

```
<assert.h>     <locale.h>    <stddef.h>
<ctype.h>      <math.h>      <stdio.h>
<errno.h>      <setjmp.h>    <stdlib.h>
<float.h>      <signal.h>    <string.h>
<limits.h>     <stdarg.h>    <time.h>
```

The non-ANSI HP C/iX header files are

```
<fcntl.h>      <search.h>
<malloc.h>     <unistd.h>
<memory.h>     <values.h>
<mpe.h>        <varargs.h>
```

Referencing Library Header Files

To reference the HP C/iX library header files, place the `#include` preprocessor directive in your source code. The order of inclusion of the header files is of no significance. The same header file may be included more than once in the same program without generating errors.

The syntax for including a HP C/iX library header file is:

```
#include <libraryname.h>
```

By enclosing *`libraryname`* in angle brackets, you instruct the HP C/iX preprocessor to look for that header file in the `H` group of the `SYS` account.

For example, if you want to use the `fprintf` function, which is in the standard I/O library, your program must specify:

```
#include stdio.h>
```

The declaration of `fprintf,` various types, and variables used by the I/O function, are found in the `stdio.h` header file.

Header file identifiers beginning with an underscore (_) are reserved for library use. You should not create identifiers that begin with an underscore within your own source code.

Library Functions and Header File Macros

The HP C/iX library contains both functions and macros. Macros improve the execution speed of certain frequently used operations. One drawback to using macros is that they do not have an address. For example, if a function expects the address of (a pointer to) another function as an argument, you cannot use a macro name in that argument. The following example illustrates the drawback:

```
#define add1(x) ((x)+=1)
extern f(void some_function( ));
main(void)
{
   &vellip;
      f(add1); /* This construct is illegal. */
   &vellip;
}
```

Using `add1` as an argument causes an error. To override a possible macro and ensure that a library function is referenced as a true function, you can do any of the following:

- Use the `#undef` directive, which causes the function name to no longer be defined as a macro.
- Enclose the function name in parentheses to suppress macro expansion.
- Take the address of the function using the & operator.

There are three ways in which a function can be declared:

- In a header file (which might generate a macro). This is the safest method to declare a standard library function.

```
#include <stdlib.h>
m=abs(n);
```

- By explicit declaration. Make sure that your declaration matches the one in this manual.

```
extern int abs(int j);
m=abs(n);
```

- By implicit declaration, if the function return type is `int`.

```
m=abs(n);
```

Header File Contents

This section describes the contents of the HP C/iX library header files. All header files provided by HP C/iX are listed. This includes those headers files that comply with the ANSI C standard, and those that do not comply with this standard. The non-ANSI headers files are provided for compatibility with other UNIX-based [1] systems and for interfacing with the MPE operating system.

Diagnostics <assert.h>

The header `assert.h>` defines the macro `assert`. If the expression passed to this function is false (equal to 0), a message is written and the program is terminated.

Character Handling <ctype.h>

The header `ctype.h>` declares several macros and external functions useful for testing and mapping characters. The functions enable you to convert between uppercase and lowercase and to classify characters as digits, nonprintable characters, uppercase, or lowercase. In all cases, the argument is an `int` that must be representable as an unsigned character or the value of the macro `EOF`.

Because their syntaxes are identical, the following example can be used for all character classification macros:

```
for  (i=0; array[i] != 0; i++))  {
      if (isprint(array[i]))
           putchar(array[i]);
      else putchar(' ');
}
```

The following identifiers are defined in `<ctype.h>`:

Table 4-1. Character Handling Macros and Functions <ctype.h>

Name	Type	Description
`isalnum()`	function	Tests whether an argument is a letter or a decimal digit.
`isalpha()`	function	Tests whether an argument is a letter.
`isascii()` [a]	function	Tests whether an argument is in the ASCII character set.
`iscntrl()`	function	Tests whether an argument is a control character.
`isdigit()`	function	Tests whether an argument is a decimal digit.
`isgraph()`	function	Tests whether an argument is any printable non-space character.
`islower()`	function	Tests whether an argument is a lowercase letter.

1. UNIX is a registered trademark of The Open Group.

Table 4-1. Character Handling Macros and Functions <ctype.h>

Name	Type	Description
`isprint()`	function	Tests whether an argument is any printable character including the space character (octal values 040 through 0176).
`ispunct()`	function	Tests whether an argument is any printable character that is not a space, a digit, or a letter.
`isspace()`	function	Tests whether an argument is a white space character.
`isupper()`	function	Tests whether an argument is an uppercase letter.
`isxdigit()`	function	Tests whether an argument is a hexadecimal digit.
`toascii()` [b]	function	Converts an integer to 7-bit ASCII.
`_tolower()`	macro	Converts an uppercase letter to lowercase.
`_toupper()`	macro	Converts a lowercase letter to uppercase.
`tolower()`	function	Converts an uppercase letter to lowercase.
`toupper()`	function	Converts a lowercase letter to uppercase.

a. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.
b. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.

Error Handling <errno.h>

The header `<errno.h>` defines several macros, all relating to the reporting of error conditions.

When an error occurs in a HP C/iX library function, an error is usually signalled to the caller through the function return value. The error is signalled by an otherwise impossible return value, usually `-1` or `NULL`. To provide more information about the cause of the error, several functions in the standard library and math library set the external variable `errno` to a non-zero value when an error occurs.

The external variable `errno` is declared in this header file. This file also defines many macro expressions for the various possible values of `errno`. The value of `errno` is zero at program startup, but is never reset to zero by the library functions.

Programs that use `errno` for error checking should reset `errno` to zero before a library function call, then inspect `errno` after the function call.

Some functions in the standard library call one or more underlying functions to perform specific tasks. For example, the `fopen` function calls the `open` function. In these cases, the underlying functions may set `errno`. The `errno` return values of the underlying functions are not documented here.

The following is a list of the common error names and their values. The POSIX/iX library provides additional error names and values. These errors are described in the *MPE/iX Developer's Kit Reference Manual*.

NOTE The values associated with symbolic names are subject to change. It is suggested that the symbolic names in `errno.h>` be used rather than the actual numeric values.

`ENOENT (2)`	`No such file.` This error occurs when a file name is specified and the file should exist but doesn't.
`EBADF (9)`	`Bad file number.` Either a file descriptor does not refer to an open file, a read (respectively write) request is made to a file that is open only for writing (respectively reading), or the file descriptor is not in the legal range of file descriptors.
`ENOMEM (12)`	`Not enough space.` A `brk` or `sbrk` call requested more space than the system is able to supply.
`EACCES (13)`	`Permission denied.` An attempt was made to access a file in a way forbidden by the protection system.
`EFAULT (14)`	`Bad address.` A bad address argument was detected, such as a null pointer.
`EINVAL (22)`	`Invalid argument.` An invalid argument, such as a bad `oflag open` argument or a bad `lseek` offset argument. This can also be set by the math functions.
`EMFILE (24)`	`Too many open files.` No process may have more than a system-defined number of file descriptors open at a time. See `_NFILE` in `stdio.h>`.
`EFBIG (27)`	`File too large.` The current output request would exceed the file limit.
`ESEEK (29)`	`Illegal seek.` An attempt to seek in a file that does not support seeking was detected, such as a seek on a terminal file.
`EDOM (33)`	`Math argument.` The argument of a function in the math package is out of the domain of the function.
`ERANGE (34)`	`Result too large.` The value of a function in the math package is not representable within machine precision.
`ENOBUFS (49)`	`No buffer space available.` An operation on a file was not performed because the system lacked sufficient buffer space.
`ESYSERR (50)`	System error. A call from an HP C/iX library function to a system intrinsic, or to the heap manager, has failed. When the ESYSERR occurs, three global variables are set:

`_mpe_intrinsic`   `_mpe_errno`   `_mpe_status`

The _mpe_intrinsic variable returns a numeric value denoting the intrinsic that has failed. The _mpe_errno variable contains the error number given by the failing intrinsic or, for a file system error, the number received by an FCHECK intrinsic call. The _mpe_status variable contains an MPE STATUS value: the first (high-order) 16 bits contain the error number and the second 16 bits contain the subsystem number. The errno.h> header file lists the symbolic names used for intrinsics and gives a type definition for _mpe_status(t_mpe_status).

The following is a list of the `_mpe_intrinsic` symbolic names and their values.

`I_CREATEPROCESS`	`(1)`	CREATEPROCESS system intrinsic
`I_FCLOSE`	`(3)`	FCLOSE system intrinsic
`I_FCONTROL`	`(4)`	FCONTROL system intrinsic
`I_FFILEINFO`	`(5)`	FFILEINFO system intrinsic
`I_FPOINT`	`(7)`	FPOINT system intrinsic
`I_FREAD`	`(8)`	FREAD system intrinsic
`I_FRENAME`	`(9)`	FRENAME system intrinsic
`I_FWRITE`	`(10)`	FWRITE system intrinsic
`I_HPCIGETVAR`	`(11)`	HPCIGETVAR system intrinsic
`I_HPFOPEN`	`(14)`	HPFOPEN system intrinsic
`I_PRINT`	`(15)`	PRINT system intrinsic
`P_HEAP`	`(16)`	PASCAL HEAP manager

The three global variables, `_mpe_intrinsic`, `_mpe_errno`, and `_mpe_status`, are set only when an error occurs and only when `errno` is set to ESYSERR. These variables are *not* cleared on a successful intrinsic call. The `_mpe_status` variable is provided to give you direct access to the error text of a failing intrinsic. You can obtain this information by calling the intrinsic HPERRMSG with `_mpe_status.word` as the input. The HPERRMSG intrinsic can write the error message to the screen or to a buffer.

NOTE When a P_HEAP error occurs, the value of `_mpe_status` does not return a valid error text when used as input to HPERRMSG because the PASCAL HEAP error messages are not in the system catalog. The `_mpe_status.decode.subsys_num` variable is set to the Pascal subsystem number, and `_mpe_status.decode.error_num` is the error number returned by the failing HEAP routine.

Example

```
#pragma intrinsic HPERRMSG
#include <errno.h>
/*****************************************************************/
/*  These are the definitions included from errno.h:            */

     struct _status_word {
        short error_num;             /* error number             */
        short subsys_num;            /* subsystem number        */
        };

     typedef union {
        struct _status_work decode; /* for individual part access */
        int    word;                /* for complete struct access */
        } t_mpe_status              /* for type definition        */

     extern t_mpe_status _mpe_status;
/******************************************************************/
```

```
if (fclose(unopened_file))
   if (errno == ESYSERR)
      HPERRMSG(2, , , _mpe_status.word, , , status);
   else
      perror(0);
```

To close a file, the HP C/iX library function `fclose` calls the system intrinsic FCLOSE. If this intrinsic call fails, `errno` is set to ESYSERR, `_mpe_errno` is set to the value returned by the FCHECK intrinsic, and `_mpe_status` is set to a value consisting of two parts. The two parts are as follows: `_mpe_status.decode.subsys_num` is the file system subsystem number (143), and `_mpe_status.decode.error_num` is set to the value returned by the FCHECK intrinsic (the same as `_mpe_errno`).

The call to HPERRMSG as shown above writes the proper error message to the terminal screen. If the function `fclose` fails for a reason other than a failing intrinsic, the C/iX library function `perror` supplies the appropriate messages.

File Control <fcntl.h>

The header <`fcntl.h`> defines arguments to the open function. The macros define constant values for file access options. See the open function in chapter 5 for more detailed information.

NOTE This header file is not defined by the ANSI C standard. Programs using this header are likely to be less portable.

Floating Types <float.h>

The header <`float.h`> defines macros that specify the characteristics of floating-point types. The following macros are defined in this header file:

Table 4-2. Floating Types <float.h>

Name	Description
FLT_RADIX	Radix of exponent representation.
FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG	Number of base-2 digits in the floating-point significand, *p*.

Table 4-2. Floating Types <float.h>

Name	Description
`FLT_DIG`, `DBL_DIG`, `LDBL_DIG`	Number of decimal digits, *q*, such that any floating-point number with *q* decimal digits can be rounded into a floating-point number with *p* radix *b* digits and back again without change to the *q* decimal digits.
`FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`	The difference between 1.0 and the last value greater than 1.0 that is representable in the given floating-point type.
`FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`	Minimum negative integer such that 2 raised to that power minus 1 is a normalized floating-point number.
`FLT_MIN`, `DBL_MIN`, `LDBL_MIN`	Minimum normalized positive floating-point number.
`FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`	Minimum negative integers such that 10 raised to that power is in the range of normalized floating-point numbers.
`FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDBL_MAX_EXP`	Maximum integers such that 2 raised to that power minus 1 is a representable finite floating-point number.
`FLT_MAX`, `DBL_MAX`, `LDBL_MAX`	Maximum representable finite floating-point number.
`FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`	Maximum integer such that 10 raised to that power is in the range of representable floating-point numbers.

Limits <limits.h>

The header `<limits.h>` defines several macros that represent basic C data type limits. The following macros are defined in this header file:

Table 4-3. Integral Type Limits <limits.h>

Name	Description
`CHAR_BIT`	The number of bits in a `char`.
`CHAR_MAX`	The maximum value stored in a `char`.
`CHAR_MIN`	The minimum value stored in a `char`.
`INT_MAX`	The maximum value stored in an `int`.
`INT_MIN`	The minimum value stored in an `int`.
`LONG_MAX`	The maximum value stored in a `long`.
`LONG_MIN`	The minimum value stored in a `long`.
`MB_LEN_MAX`	The maximum length of a multibyte character.

Table 4-3. Integral Type Limits <limits.h>

Name	Description
`SCHAR_MAX`	The maximum value stored in a signed `char`.
`SCHAR_MIN`	The minimum value stored in a signed `char`.
`SHRT_MAX`	The maximum value stored in a `short`.
`SHRT_MIN`	The minimum value stored in a `short`.
`UCHAR_MAX`	The maximum value stored in an `unsigned char`.
`UINT_MAX`	The maximum value stored in an `unsigned int`.
`ULONG_MAX`	The maximum value stored in an `unsigned long`.
`USHRT_MAX`	The maximum value stored in an `unsigned short`.

Localization <locale.h>

The header `<locale.h>` contains a structure definition, several macro definitions, and the declarations for two functions. These allow you to select the desired locale at run time. This header is used for Native Language Support (NLS).

The macro definitions are constants that define the various categories of objects that can be localized, such as the collating sequence used in sorting and the monetary symbol of the local currency. The structure `lconv` defines a record used for holding values associated with locale-specific numeric formatting (monetary and otherwise). The functions set and retrieve the current locale and manipulate the numeric formatting values.

The following identifiers are declared in `<locale.h>`:

Table 4-4. Localization <locale.h>

Name	Type	Description
`LC_ALL`	macro	A constant used to define a *`category`* of localizable objects. Sets and gets the current heterogeneous locales used by all categories of localizable objects.
`LC_COLLATE`	macro	The *`category`* that controls the current locale of the `strcoll` and `strxfrm` functions.
`LC_CTYPE`	macro	The *`category`* that controls the current locale of the character handling functions.
`LC_MONETARY`	macro	The *`category`* that controls the current locale for monetary formatting by the `localeconv` function.
`LC_NUMERIC`	macro	The *`category`* that controls the current locale for decimal-point, digit separator, and monetary formatting by the `localeconv` function.
`LC_TIME`	macro	The *`category`* that controls the current locale for time formatting by the `strftime` function.

Table 4-4. Localization <locale.h>

Name	Type	Description
`lconv`	type definition	A structure type definition for a record containing numeric and monetary formatting values. See the description of the `localeconv` function for more details.
`localeconv()`	function	Returns information about the editing symbols of a numeric quantity specific to a locale.
`NULL`	macro	The constant 0.
`setlocale()`	function	Controls locale-specific features of the library.

Memory Management <malloc.h>

The header `<malloc.h>` declares several memory management functions, several `mallopt` argument macros and a structure returned by the `mallinfo` function. The following identifiers are declared in this header file:

Table 4-5. Memory Management <malloc.h>

Name	Type	Description
`calloc()`	function	Allocates a block of memory.
`free()`	function	Frees a block of allocated memory.
`mallinfo`	type definition	A data type definition used when declaring parameters and return values. See the function descriptions for more information.
`mallinfo()`	function	Returns information describing space usage.
`malloc()`	function	Allocates a block of memory.
`mallopt()`	function	Provides control over the memory allocation algorithm.
`realloc()`	function	Changes the size of a block of allocated memory.

NOTE This header file is not defined by the ANSI C standard. Programs using this header are likely to be less portable.

Math Library <math.h>

The header `<math.h>` contains declarations for all functions in the HP C/iX math library. It contains various function declarations for routines in the standard library that return floating-point values.

The header also defines structures and constants used by the `matherr` error-handling mechanisms, and macros provided for compatibility with other implementations.

The following identifiers are declared in this header file:

Table 4-6. Math Library Functions <math.h>

Name	Type	Description
`abs()`	function	Computes the absolute value of an integer.
`acos()`	function	Returns the arc cosine in radians of an input value.
`asin()`	function	Returns the arc sine of an input value.
`atan2()`	function	Returns the arc tangent of the input Cartisian coordinates.
`atan()`	function	Returns the arc tangent of the input value.
`ceil()`	function	Computes the ceiling function that finds the smallest integer that is greater than or equal to a specified real number.
`cos()`	function	Returns a cosine value for an input angle.
`cosh()`	function	Computes the hyperbolic cosine of an angle.
`DOMAIN`	macro	An integral constant returned through `errno` when the result is outside the domain of the returned data type.
`erf()` [a]	function	Returns the statistical error function of the input value.
`erfc()` [b]	function	Returns the complementary error function of the input value.
`errno`	global variable	A global external `int` variable that provides additional information regarding errors encountered in library routines.
`exp()`	function	Returns a base number raised to the power of the argument.
`fabs()`	function	Computes the absolute value of a floating-point number.
`floor()`	function	Computes the largest integer value less than or equal to its argument.
`fmod()`	function	Returns the floating-point remainder of a division operation.
`frexp()`	function	Breaks a floating-point number into a normalized fraction and an integral power of 2.
`gamma()`	function	Returns the log gamma of the input value.
`hypot()`	function	Returns the length of the hypotenuse of a right triangle.
`HUGE`	macro	The identifier for the maximum value of a single-precision floating-point number. Provided for System V compatibility.
`HUGE_VAL`	macro	The ANSI C identifier for the maximum value (+ infinity) of a single-precision floating-point number. Returned from math library functions when the result is too large.
`j0()`	function	Returns Bessel functions of *x* of the first kind of order zero.
`j1()` [c]	function	Returns Bessel functions of *x* of the first kind of order one.

Table 4-6. Math Library Functions <math.h>

Name	Type	Description
`jn()`	function	Return Bessel functions of *x* of the first kind of order i.
`ldexp()`	function	Accepts a double value and an integer exponent and returns a double quantity equal to N * 2^{exp}.
`log()`	function	Returns the natural logarithm of a positive number.
`log10()`	function	Returns the logarithm base ten of a positive number.
`exception`	type definition	A data type definition used with the non-standard `matherr` function.
`M_E`	macro	The base of natural logarithms (e).
`M_LOG2E`	macro	The base-2 logarithm of e.
`M_LOG10E`	macro	The base-10 logarithm of e.
`M_LN2`	macro	The natural logarithm of 2.
`M_LN10`	macro	The natural logarithm of 10.
`M_PI`	macro	The ratio of the circumference of a circle to its diameter.
`M_PI_2`	macro	Half of the ratio of the circumference of a circle to its diameter.
`M_PI_4`	macro	One quarter of the ratio of the circumference of a circle to its diameter.
`M_1_PI`	macro	The reciprocal of the ratio of the circumference of a circle to its diameter.
`M_2_PI` [d]	macro	Two times the reciprocal of the ratio of the circumference of a circle to its diameter.
`M_2_SQRTPI`	macro	The square root of the ratio of the circumference of a circle to its diameter.
`M_SQRT2`	macro	The positive square root of 2.
`M_SQRT1_2`	macro	The positive square root of 1/2.
`MAXFLOAT`	macro	The largest floating-point number allowed with this architecture.
`matherr()`	function	A user written call-back routine that can be invoked by many functions in the math library when errors are detected.
`modf()`	function	Accepts a double value and splits the value into its integer and fractional parts.
`OVERFLOW`	macro	An integral constant returned through `errno` when an arithmetic overflow has occurred.
`PLOSS`	macro	An integral constant returned through `errno` when a significant loss of precision has occurred.

Table 4-6. Math Library Functions <math.h>

Name	Type	Description
`pow()`	function	Returns the value of a number raised to the power of an exponent.
`signgam`	external integer	Contains the sign of the value returned by the `gamma` function.
`sin()`	function	Computes a sine value.
`SING`	macro	Expands to an integral constant.
`sinh()`	function	Computes the hyperbolic sine of an angle.
`sqrt()`	function	Computes the square root of an input value.
`tan()`	function	Computes a tangent value.
`tanh()`	function	Computes the hyperbolic tangent value of an angle.
`TLOSS` [e]	macro	An integral constant returned through `errno` when an arithmetic error has occurred.
`y0()`	function	Return Bessel functions of *x* of the second kind of order zero.
`y1()`	function	Return Bessel functions of *x* of the second kind of order one.
`yn()`	function	Return Bessel functions of *x* of the second kind of order *i*.

a. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.
b. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.
c. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.
d. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.
e. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.

Memory Handling <memory.h>

The header `<memory.h>` declares several functions useful for manipulating character arrays and other objects treated as character arrays. The following functions are declared by this header file:

Table 4-7. Memory Handling <memory.h>

Name	Type	Description
`memccpy()`	function	Copies characters from one object to another until a specified character is found or until the specified count is reached.
`memchr()`	function	Searches memory for a specified character.
`memcmp()`	function	Compares the first *n* characters of two objects.

Table 4-7. Memory Handling <memory.h>

Name	Type	Description
`memcpy()`	function	Copies a specified number of characters from one object to another.
`memset()`	function	Initializes an object with a supplied character value.

NOTE This header file is not defined by the ANSI C standard. Programs using this header are likely to be less portable. Use `<string.h>` for ANSI compliance.

MPE Aids <mpe.h>

The `mpe.h>` header file declares several types, constants, and functions that allow you to more easily interface with the MPE/iX operating system. Refer to chapter 3, "Interfacing With MPE/iX," for additional information.

NOTE This header file is not defined by the POSIX or ANSI C standards. Programs using this header are likely to be less portable.

Searching Utilities <search.h>

The header `<search.h>` contains identifiers used for creating and searching binary trees and hash tables. The following identifiers are declared:

Table 4-8. Search Utilities <search.h>

Name	Type	Description
`ACTION`	type definition	An enumerated type with the enumerated constants `FIND` and `ENTER`.
`ENTRY`	type definition	A structured type defining a record with a search key and data.
`hcreate()`	function	Allocates sufficient space for a hash table used by `hsearch()`.
`hdestroy()`	function	Destroys a search table created by `hcreate()`.
`lsearch()`	function	Performs a linear search and update.
`tdelete()`	function	Deletes a specified node from a binary search tree.
`tfind()`	function	Searches for a specified entry in a binary search tree.
`tsearch()`	function	Returns a pointer into a hash table indicating the location of a specified entry.
`twalk()`	function	Traverses a binary search tree and returns the value at the specified node.
`VISIT`	type definition	An enumerated type with the enumerated constants `preorder`, `postorder`, `endorder`, and `leaf`.

NOTE This header file is not defined by the ANSI C standard. Programs using this header are likely to be less portable.

Non-local Jumps <setjmp.h>

The header <setjmp.h> declares a type and several functions for bypassing the normal function call and return discipline. The following identifiers are declared in this header file:

Table 4-9. Nonlocal Jumps <setjmp.h>

Name	Type	Description
`jmp_buf`	type definition	An array type used by `setjmp` to save and by `longjmp` to restore a program's environment.
`longjmp()`	function	Restores an environment previously saved by `setjmp()`.
`setjmp()`	function	Saves the current environment.

Signal Handling <signal.h>

The header <signal.h> declares a type, functions, and macros for handling various signals that can be raised during program execution.

The following identifiers are defined in <signal.h>:

Table 4-10. Signal Handling <signal.h>

Name	Type	Description
`raise`	function	Causes a signal to be raised.
`sig_atomic_t`	type definition	An integral type, such that an object of this type can be accessed in an atomic fashion (in one operation), even in the presence of external interrupts.
`SIG_DFL`	macro	Passed as the second parameter to `signal()`. Specifies default signal handling.
`SIG_ERR`	macro	Returned by `signal()` to indicate an error when calling the `signal` function.
`SIG_IGN`	macro	Passed as the second parameter to `signal()`. Specifies that exceptions should be ignored.
`SIGABRT`	macro	The signal raised by the `abort` function indicating abnormal termination.
`SIGFPE`	macro	A signal indicating that a floating-point exception or erroneous arithmetic operation has occurred (for example, divide by 0).
`SIGILL`	macro	A signal indicating that an illegal instruction was executed (possibly after a jump).

Table 4-10. Signal Handling <signal.h>

Name	Type	Description
SIGINT	macro	A signal indicating that an interactive interrupt has been received.
signal	function	Specifies how a signal is to be handled.
SIGSEGV	macro	A signal indicating that an invalid address to storage has been requested.
SIGTERM	macro	A signal indicating that a termination request was sent to the program.

Variable Number of Arguments <stdarg.h>

The header `<stdarg.h>` contains a type definition and three macros. These can be used to determine the arguments of a function that can be called with a variable number of arguments. The variable number of arguments is indicated by an ellipsis in the function declaration.

NOTE The header `<varargs.h>` also contains the same type definitions and macros described in this section. However, `<varargs.h>` is not defined by the ANSI C standard.

The following identifiers are defined in `<stdarg.h>`:

Table 4-11. Variable Number of Arguments <stdarg.h>

Name	Type	Description
va_arg	macro	Returns the type and value of the next argument in the argument list ap.
va_end	macro	Terminates access to the variable argument list by making ap unusable.
va_list	type definition	A pointer to a double used to store information needed by the `<stdarg.h>` macros.
va_start	macro	Initializes the ap pointer (of type va_list) to the argument list for subsequent use by va_arg and va_end.

Common Definitions <stddef.h>

The following identifiers are defined in `<stddef.h>`:

Table 4-12. Common Definitions <stddef.h>

Name	Type	Description
NULL	macro	The constant 0.

Table 4-12. Common Definitions <stddef.h>

Name	Type	Description
`offsetof(`*`type, identifier`*`)`	macro	Expands to an integral constant that has type `size_t`, the value of which is the offset in bytes, from the beginning of a structure designated by *`type`*, of the member designated by *`identifier`*.
`ptrdiff_t`	type definition	The signed integral type of subtracting two pointers.
`size_t`	type definition	The unsigned integral type of the `sizeof` operator.
`wchar_t`	type definition	The integral data type large enough to represent all members of the largest extended character set among the supported locales.

Input/Output <stdio.h>

The header `<stdio.h>` defines a structure, functions, and macros that are used for input and output.

The following identifiers are defined in `<stdio.h>`:

Table 4-13. Input/Output <stdio.h>

Name	Type	Description
`BUFSIZ`	macro	Specifies the size of the buffers used by `setbuf`.
`clearerr()`	macro, function	Clears the end-of-file and error indicator of a stream.
`ctermid()`[a]	function	Returns `$stdlist` as the file name for the terminal.
`EOF`	macro	Returned upon end-of-file or upon error by most integer functions that deal with streams.
`fclose()`	function	Closes an open file.
`fdopen()`	function	Opens a stream on a file descriptor.
`feof()`	macro, function	Tests whether the end-of-file indicator for a stream has been set.
`ferror()`	macro, function	Tests whether the error indicator for a stream has been set.
`fflush()`	function	Flushes an I/O buffer to a file.
`fgetc()`	function	Returns the next character from an open stream.
`fgetpos()`	function	Returns the current file position of an open stream.
`fgets()`	function	Reads a string from an open stream.
`FILE`	type definition	A type definition for a file descriptor. This type defines a data structure used internally by the I/O routines to identify open files and maintain context when accessing files.

Table 4-13. Input/Output <stdio.h>

Name	Type	Description
`FILENAME_MAX`	macro	Specifies the maximum number of characters allowed in a file name.
`fileno()` [b]	macro, function	Maps a stream pointer to a file descriptor.
`fopen()`	function	Opens a file.
`FOPEN_MAX`	macro	Specifies the minimum number of files that the operating system guarantees may be opened simultaneously.
`fpos_t`	type definition	A type definition for an object capable of defining all unique locations within a file.
`fprintf()`	function	Writes data in formatted form to an open stream.
`fputc()`	function	Writes a character to an output stream.
`fputs()`	function	Writes a string to an output stream.
`fread()`	function	Reads data items from an open stream.
`freopen()`	function	Closes and reopens a stream.
`fscanf()`	function	Reads externally formatted data from an open stream.
`fseek()`	function	Positions the next I/O operation on an open stream to an new position.
`fsetpos()`	function	Sets the file position for the stream.
`ftell()`	function	Returns the current file position indicator for the next I/O operation on an open stream.
`fwrite()`	function	Writes data items to an open stream.
`getc()`	macro, function	Reads a character from an open stream.
`getchar()`	macro, function	Reads a character from the standard input stream `stdin`.
`gets()`	function	Reads a string from the standard input stream `stdin`.
`getw()` [c]	function	Reads a word (4 bytes) from an open stream.
`L_tmpnam`	macro	Specifies the number of bytes needed to hold a temporary file name generated by the `tmpnam` function.
`NULL`	macro	The constant 0.
`perror()`	function	Prints an error message corresponding to the `errno` global variable.
`printf()`	function	Writes data in formatted form to the standard output stream `stdout`.
`putc()`	macro, function	Writes a character to an open stream.

Table 4-13. Input/Output <stdio.h>

Name	Type	Description
`putchar()`	macro, function	Writes a character to the standard output stream `stdout`.
`puts()`	function	Writes a string to the standard output stream `stdout`.
`putw()`	function	Writes a word (4 bytes) to an open stream.
`remove()`	function	Purges an existing file.
`rename()`	function	Renames an existing file.
`rewind()`	function	Sets the file position indicator for a stream to the beginning of the file.
`scanf()`	function	Reads externally formatted data from the standard input stream `stdin`.
`SEEK_CUR`	macro	A constant value that may be used as the *`ptrname`* parameter to `fseek`. Seek relative to the current location in the file.
`SEEK_END`	macro	A constant value that may be used as the *`ptrname`* parameter to `fseek`. Seek relative to the end of file.
`SEEK_SET`	macro	A constant value that may be used as the *`ptrname`* parameter to `fseek`. Seek relative to the beginning of file.
`setbuf()`	function	Assigns a buffer to an open stream.
`setvbuf()`	function	Assigns a buffer and buffering method to an open stream.
`size_t`	type definition	The unsigned integral type of the `sizeof` operator.
`sprintf()`	function	Writes formatted data to a character string in memory.
`sscanf()`	function	Reads formatted data from a character string in memory.
`stderr`	macro	The standard error file.
`stdin`	macro	The standard input file.
`stdout`	macro	The standard output file.
`TMP_MAX`	macro	The maximum number of unique file names that can be generated by the `tmp_name` function.
`tmpfile()`	function	Creates a temporary file.
`tmpnam()`	function	Creates a name for a temporary file.
`ungetc()`	function	Pushes back a single character onto an open stream.
`vfprintf()`	function	Writes data in formatted form to an open stream using a variable argument list.
`vprintf()`	function	Writes data in formatted form to an open stream using a variable argument list.

Table 4-13. Input/Output <stdio.h>

Name	Type	Description
`vsprintf()`	function	Writes formatted data to a character string in memory using a variable argument list.
`_IOFBF`, `_IOLBF`, `_IONBF`	macro	Constant expressions with values suitable for use as the third argument to the `setvbuf` function.
`_NFILE` [d]	macro	Defines the maximum number of open files allowed per process.

a. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.
b. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.
c. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.
d. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.

General Utilities <stdlib.h>

The header `<stdlib.h>` contains a number of general-purpose declarations and definitions. It defines functions used for:

- string data type conversion
- multibyte character and string manipulation
- memory management
- array searching and sorting
- integer arithmetic
- communicating with the environment

The following identifiers are defined in `<stdlib.h>`:

Table 4-14. General Utilities <stdlib.h>

Name	Type	Description
`abort()`	function	Terminates a program abnormally.
`abs()`	function	Computes the absolute value of an integer.
`atexit()`	function	Specifies a function to call when a program terminates.
`atof()`	function	Converts a string to a double floating-point value.
`atoi()`	function	Converts a string to an integer.
`atol()`	function	Converts a string to a long integer.
`bsearch()`	function	Performs a binary search of a sorted array.

Table 4-14. General Utilities <stdlib.h>

Name	Type	Description
calloc()	function	Allocates a block of memory.
div()	function	Computes the quotient and remainder of two integers.
div_t	type definition	A data type definition used when declaring the return value for div().
exit()	function	Terminates the calling process normally.
EXIT_FAILURE	macro	A value that can be passed to the exit function to indicate unsuccessful program termination.
EXIT_SUCCESS	macro	A value that can be passed to the exit function to indicate successful program termination.
free()	function	Frees a block of allocated memory.
getenv()	function	Returns the value of an environment variable.
labs()	function	Computes the absolute value of a long integer.
ldiv()	function	Computes the quotient and remainder of two long integers.
ldiv_t	type definition	A data type definition used when declaring the return value for ldiv().
malloc()	function	Allocates a block of memory.
mblen()	function	Determines the number of characters in a multibyte character.
mbstowcs()	function	Converts a sequence of multibyte characters in a null-terminated string to a sequence of wide character codes.
mbtowc()	function	Converts a single multibyte character to its wide character representation.
MB_CUR_MAX	macro	Maximum size in bytes of a multibyte character.
NULL	macro	The constant 0.
qsort()	function	Sorts an array of objects.
rand()	function	Returns a random number.
RAND_MAX	macro	The maximum value returned by the rand function.
realloc()	function	Changes the size of a block of allocated memory.
size_t	type definition	The unsigned integral type of the sizeof operator.
srand()	function	Sets a starting point for calls to the rand function.
strtod()	function	Converts a string to a double-precision, floating-point number.
strtol()	function	Converts a string to a long integer value.

Table 4-14. General Utilities <stdlib.h>

Name	Type	Description
`strtoul()`	function	Converts a string to an unsigned integer representation.
`system()`	function	Executes an MPE/iX command.
`wchar_t`	type definition	A data type definition used for wide characters.
`wcstombs()`	function	Converts a sequence of wide character codes to a sequence of multibyte characters.
`wctomb()`	function	Converts a single wide character value to its multibyte character representation.

String Handling <string.h>

The header `<string.h>` declares several functions for manipulating character arrays and other objects treated as character arrays. A *`string`* is a sequence of characters terminated by and including the first null character. A *`pointer to`* a string is a pointer to its first character. The *`length`* of the string is the number of characters preceding the first null character.

The following identifiers are declared by `<string.h>`:

Table 4-15. String Handling <string.h>

Name	Type	Description
`memchr()`	function	Searches memory for a specified character.
`memcmp()`	function	Compares the first *`n`* characters of two objects.
`memcpy()`	function	Copies a specified number of characters from one object to another.
`memmove()`	function	Copies a specified number of characters from one object to another. Allows source and destination objects to overlap.
`memset()`	function	Initializes an object with a supplied character value.
`NULL`	macro	The constant 0.
`size_t`	type definition	The unsigned integral type of the `sizeof` operator.
`strcat()`	function	Appends one string to another.
`strchr()`	function	Locates the first occurrence of a specified character within a string.
`strcmp()`	function	Compares two strings and returns an integer indicating the result of the comparison.
`strcpy()`	function	Copies the contents of one string to another string.
`strcspn()`	function	Returns the length of the first substring in one string composed entirely of non-members of the character set of another string.

Table 4-15. String Handling <string.h>

Name	Type	Description
`strerror()`	function	Maps an error number to a message string.
`strlen()`	function	Computes the length of the string pointed to by *s*.
`strncat()`	function	Appends a copy of one string to another string.
`strncmp()`	function	Compares two strings up to a maximum of *n* characters and returns the result of the comparison.
`strncpy()`	function	Copies all or part of one string into another string.
`strpbrk()`	function	Returns a pointer to the location in one string of the first occurrence of any member of the character set in another string.
`strrchr()`	function	Locates the last occurrence of a supplied character within a string.
`strspn()`	function	Returns the length of the first substring in one string composed entirely of members of the character set in another string.
`strstr()`	function	Locates the first occurrence in one string of the sequence of characters specified by another string.
`strtok()`	function	Divides one string into zero or more tokens. The token separators consist of any characters contained in another string.
`strxfrm()`	function	Transforms a string in a manner appropriate for the current locale.

Date and Time <time.h>

The header `<time.h>` declares data types, global variables, and functions for storing and manipulating time values.

The date and time functions enable you to access the date and time maintained by the system clock. The functions handle daylight savings time, and automatically convert between standard time and daylight savings time when appropriate.

Most of the functions require the calendar time returned by `time()`, that is the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.

The following identifiers are declared in this header file:

Table 4-16. Date and Time <time.h>

Name	Type	Description
`asctime()`	function	Converts a `tm` structured time variable into a null terminated 26-character string.
`clock()`	function	Reports CPU time used.
`clock_t`	type definition	Return values from the `clock` function.

Table 4-16. Date and Time <time.h>

Name	Type	Description
`CLOCKS_PER_SEC`	macro	The number of clock ticks per second, as counted by the `clock` function.
`ctime()`	function	Converts a calendar time into a 26-character ASCII string.
`daylight` [a]	global variable	Communicates with functions in this library. See the function descriptions in chapter 5 for more information.
`difftime()`	function	Computes the difference between two times.
`gmtime()`	function	Converts time to Coordinated Universal Time (UTC) in the structured `tm` type format.
`localtime()`	function	Converts time to the local time zone.
`mktime()`	function	Converts a broken-down time of type `struct tm` to a calendar time of of type `time_t`.
`NULL`	macro	The constant 0.
`size_t`	type definition	The data type used to return values from the `sizeof` operator.
`strftime()`	function	Creates a formatted time string.
`time()`	function	Returns the current calendar time.
`time_t`	type definition	A data type definition used to return values from the `time` function. It is also used to declare return values and parameters of other `time.h>` functions. A time value represented using type `time_t` is referred to as a calendar time.
`timezone`	macro	A constant containing the offset of the local time zone to GMT. The local time zone defaults to EST. This value can be changed by setting the environment variable TZ using the SETVAR command.
`tm`	type definition	A structure data type definition used to declare parameters and return values for `time.h>` functions. Contains the components of a calendar time value, broken down into individual fields for year, month, day, hour, and so on. A time value represented using type `struct tm` is referred to as a broken-down time.
`tzname` [b]	global variable	An external variable used to communicate with functions in this library. See the time function descriptions in chapter 5 for more information.
`tzset()`	function	Sets time zone conversion information.

a. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.

b. These identifiers are not defined by the ANSI C standard. Programs using these identifiers are likely to be less portable.

The `tm` structure, used by several of the `<time.h>` functions, is shown below:

```
struct tm {
   int tm_sec;    /* seconds after the minute (0 through 59) */
   int tm_min;    /* minutes after the hour (0 through 59) */
   int tm_hour;   /* hours since midnight (0 through 23) */
   int tm_mday;   /* day of the month (1 through 31) */
   int tm_mon;    /* month of the year (0 through 11) */
   int tm_year;   /* years since 1900 */
   int tm_wday;   /* days since Sunday (0 through 6) */
   int tm_yday;   /* day of the year (0 through 365) */
   int tm_isdst;  /* daylight savings time flag (1 = dst) */
};
```

Standard Macros <unistd.h>

The header `<unistd.h>` defines several macros that are used as arguments to the `lseek` function.

Table 4-17. Standard Macros <unistd.h>

Name	Type	Description
SEEK_CUR	macro	A constant value that may be used as the whence parameter to `lseek`. Seek relative to the current location in the file.
SEEK_END	macro	A constant value that may be used as the whence parameter to `lseek`. Seek relative to the end of file.
SEEK_SET	macro	A constant value that may be used as the whence parameter to `lseek`. Seek relative to the beginning of file.

NOTE This header file is not defined by the ANSI C standard. Programs using this header are likely to be less portable.

Machine-Dependent Values <values.h>

The header `<values.h>` contains a set of manifest constants, conditionally defined for particular processor architectures. The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit. The following macros are defined in this header file:

Table 4-18. Machine-Dependent Values <values.h>

Name	Type	Description
BITS	macro	The number of bits in a specified type, such as `int`.

Table 4-18. Machine-Dependent Values <values.h>

Name	Type	Description
HIBITS	macro	The value of a short integer with only the high-order bit set.
HIBITL	macro	The value of a long integer with only the high-order bit set.
HIBITI	macro	The value of a regular integer with only the high-order bit set.
MAXSHORT	macro	The maximum value of a signed short integer.
MAXLONG	macro	The maximum value of a signed long integer.
MAXINT	macro	The maximum value of a signed regular integer.
MAXFLOAT, LN_MAXFLOAT	macros	The maximum value of a single-precision floating-point number and its natural logarithm.
MAXDOUBLE, LN_MAXDOUBLE	macros	The maximum value of a double-precision floating-point number and its natural logarithm.
MINFLOAT, LN_MINFLOAT	macros	The minimum positive value of a single-precision floating-point number and its natural logarithm.
MINDOUBLE, LN_MINDOUBLE	macros	The minimum positive value of a double-precision floating-point number and its natural logarithm.
FSIGNIF	macro	The number of significant bits in the mantissa of a single-precision floating-point number.
DSIGNIF	macro	The number of significant bits in the mantissa of a double-precision floating-point number.

NOTE This header file is not defined by the ANSI C standard. Programs using this header are likely to be less portable.

Variable Arguments (old form) <varargs.h>

The header <varargs.h> declares several types and macros for calling variable argument functions.

Table 4-19. Variable Arguments <varargs.h>

Name	Type	Description
va_arg	macro	Returns the next argument in an argument list.
va_alist	type definition	A type definition used when declaring the variable used as the ap parameter to the va_arg, va_end, and va_start macros.
va_start	macro	Initializes a variable to the beginning of an argument list.

Table 4-19. Variable Arguments <varargs.h>

Name	Type	Description
va_end	macro	Terminates access to a variable argument list.

NOTE This header file is for non-ANSI mode only. Use `stdarg.h>` in ANSI mode. Using this header is likely to make a program less portable.

5 HP C/iX Library Function Descriptions

This chapter provides descriptions of HP C/iX library functions arranged in alphabetical order.

If a function conforms to the ANSI C or POSIX standard, a cross-reference to the standard is given at the end of the function description.

a64l

Converts a base-64 ASCII string to a long integer.

Syntax

[long a64l (char *s*);]

Parameters

s	A pointer to a null terminated base-64 ASCII string. Maximum length is 6 bytes; not counting the null terminator.

Return Values

x	A long integer containing the binary value of the base-64 ASCII string.

Description

This function maintains numbers stored in base-64 ASCII characters. Long integers can be represented by up to six characters. Each character represents a *digit* in a radix-64 notation.

The characters used to represent digits are:

Characters	Digits
`.`	0
`/`	1
`0` through `9`	2 through 11
`A` through `Z`	12 through 37
`a` through `z`	38 through 63

The leftmost character is the least significant digit. For example:

$$a0 = (38 \times 64^0) + (2 \times 64^1) = 166$$

The `a64l` function is passed a pointer to a null-terminated base-64 representation and returns a corresponding long value. If the string pointed to by *s* contains more than six characters, `a64l()` uses the first six (leftmost) characters.

See Also

`l64a()`

abort

Terminates a program abnormally.

Syntax

```
#include <stdlib.h>
void abort (void);
```

Parameters

None.

Return Values

None.

Description

The `abort` function causes abnormal program termination to occur unless the signal `SIGABRT` is being caught and the signal handler does not return.

The `abort` function closes all open files if possible and then terminates the process. Temporary files under MPE/iX are not saved.

Process termination is achieved by calling the system intrinsic `QUIT`. This intrinsic transmits an abort message to the list device of the calling process and sets the job control word (JCW) to indicate that the program terminated in an error state. The C language job control word (CJCW) is also set to a non zero value to indicate the error condition.

See Also

`exit()`, `raise()`, `signal()`, ANSI C 4.10.4.1, POSIX.1 8.2.3.12, *MPE/iX Intrinsics Reference Manual*

abs

Computes the absolute value of an integer argument.

Syntax

```
#include <stdlib.h>
int abs (int x);
```

Parameters

x	An integer value whose absolute value is to be computed.

Return Values

x	The absolute value of the integer specified in *x*.

Example

The following program calculates integer absolute values until a zero is entered from the keyboard:

```
#include <stdlib.h>
#include <stdio.h>
main(void)
{
int value;
do
  {
  printf("Enter value: ");
  scanf("%d", &value);
  if (value == 0)
     exit (0);
  printf("Absolute value of %d is %d.\n, value, abs(value));
  }
  while (value !=0);
}
```

See Also

`fabs()`, `labs()`, `floor()`, ANSI C 4.10.6.1, POSIX.1 8.1

access

Determines the accessibility of a file.

Syntax

```
#include <unistd.h>
int access (char *fname, int amode);
```

Parameters

fname	A pointer to a character string containing a file name.
amode	An integer indicating whether read or write access to a file is requested.

Return Values

0 Requested access is permitted.

–1 Requested access is denied; `errno` is set to one of the following values:

`ENOENT`	Read, write, or execute (search) permission is requested for a null path name, or the named file does not exist.
`EACCES`	The requested access is denied.
`ESYSERR`	A call to a system intrinsic failed.

Description

The `access` function checks for read or read/write access for the file referenced by *fname*. The bit pattern contained in *amode* is constructed as follows:

`04`	Read access
`02`	Write access

Other values of *amode* are not supported.

The `access` function is not supported in the POSIX/iX library. If called, `access()` returns a -1 and sets `errno` to `ENOSYS`.

acos

Returns the arc cosine in radians of the input value.

Syntax

```
#include <math.h>
double acos (double x);
```

Parameters

x	A real number.

Return Values

n	The arc cosine of *x*.
0	The magnitude of the argument of `acos` is greater than one or less than negative one. In addition, `errno` is set to `EDOM`.

Description

The `acos` function returns the arc cosine of *x*, in the range of zero to pi. A message indicating a `DOMAIN` error is printed on the standard error output if *x* is greater than one or less than negative one.

Error-handling can be changed by a user-written `matherr` function.

See Also

`matherr()`, ANSI C 4.5.2.1, POSIX.1 8.1

asctime

Converts a tm structured time variable into a null-terminated 26-character string.

Syntax

```
#include <time.h>
char *asctime (const struct tm *timeptr);
```

Parameters

timeptr	A pointer to a structure of type tm that contains the broken-down time.

Return Values

x	A pointer to the string.

Description

The asctime function provides a way for you to get the current time, modify it in some way, and then print the result in ASCII form.

The *timeptr* parameter points to a structure of type tm whose members were assigned values with localtime(), gmtime(), or explicitly by you. The asctime function returns a character pointer to a null terminated string with a maximum length of 26 characters. This string is the same type as the string returned by ctime(). Because asctime() returns a pointer to a static character array, it is overwritten by subsequent calls to asctime().

Example

The date command shown in the section on ctime() can be rewritten using localtime() and asctime():

```
   #include <stdio.h>
   #include <time.h>
   main()
   {
      int time(), nseconds;
      struct tm *ptr, *localtime();
      char *string, *asctime();

      nseconds = time(NULL);

/* you may modify the current time in tm here */

      string = asctime(ptr);
      printf("%s", string);
   }
```

This program illustrates an indirect way to obtain the date, but it does enable you to modify the date stored in tm before you print the data. If you only want to print the date, use the time/ctime combination.

Of all the ctime functions, the localtime function is the most useful. The localtime function enables you to break up the current time into chunks that can be easily referenced and examined for such applications as personal calendar programs and program schedulers. Many of the tm values can be used as indices into arrays containing strings identifying months and days. For example, declaring an external array like

```
   char *month[ ] = { "January", "February", "March", "April",
                     "May", "June", "July", "August", "September",
                     "October", "November", "December"
                   };
```

enables you to use tm_mon as an index into this array to obtain the actual month name. The same thing can be done with tm_wday if you initialize an array containing the names of the days of the week.

See Also

clock(), mktime(), localtime(), time(), ANSI C 4.12.3.1, POSIX.1 8.1.1

asin

Returns the arc sine of the input value in radians.

Syntax

```
#include math.h>
double asin (double x);
```

Parameters

x	A real number.

Return Values

n	The arc sine of *x*.
0	The magnitude of the argument of `asin` is greater than one or less than negative one. `errno` is set to `EDOM`.

Description

The `asin` function returns the arc sine of *x*, in the range of –pi/2 to pi/2.

A `DOMAIN` error is printed on the standard error output if *x* is greater than one or less than negative one.

Error-handling can be changed by a user-written `matherr` function.

See Also

`matherr()`, ANSI C 4.5.2.2, POSIX.1 8.1

assert

Terminates the program if the assertion is false.

Syntax

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

Parameters

expression	An integer value to be evaluated.

Return Values

None.

Description

The assert macro terminates the program if the assertion is false. The assert macro takes a single integer (expression) argument. If the expression evaluates to 0 (false), assert() writes a message containing the expression that tested false and the line number where the assert occurred. The program then terminates. The macro NDEBUG is referenced but not defined in <assert.h>. If NDEBUG is defined at the point when <assert.h> is included, the assert macro calls have no effect. The NDEBUG macro enables the operation of the assert macro:

NDEBUG Definition	**assert macro effect**
Defined	Calls are ignored (no debugging done)
Undefined	Calls are processed (debugging is done)

See Also

abort(), ANSI C 4.2.1.1, POSIX.1 8.1

atan2

Returns the arc tangent of the input Cartisian coordinates *x* and *y*.

Syntax

```
#include math.h>
double atan2 (double y, double x)
```

Parameters

y	A real number indicating the Cartisian coordinate `y`.
x	A real number indicating the Cartisian coordinate `x`.

Return Values

n	The arc tangent of (*x*, *y*).
0	Indicates both arguments are zero and `errno` is set to `EDOM`. A `DOMAIN` error is also printed on the standard error output device.

Description

The `atan2` function returns the arc tangent of *y/x*, in the range of –pi to pi. It uses the signs of both arguments to determine the quadrant of the return value.

Error handling can be changed by a user-written `matherr` function.

See Also

`matherr()`, ANSI C 4.5.2.4, POSIX.1 8.1

atan

Returns the arc tangent of the input value *x*.

Syntax

```
#include math.h>
double atan (double x);
```

Parameters

x	A real number.

Return Values

y	The arc tangent of *x* in the range of –`pi/2` to `pi/2`.

Description

The `atan` function returns the arc tangent of *x*, in the range –`pi/2` to `pi/2`. No range or domain errors are possible.

See Also

`matherr()`, ANSI C 4.5.2.3, POSIX.1 8.1

atexit

Specifies a function to call when a program terminates.

Syntax

```
#include <stdlib.h>
int atexit (void (*func) (void));
```

Parameters

func	A pointer to a function to be registered.

Return Values

0	The function is successfully registered.
≠0	An error occurred.

Description

The `atexit` function registers a function pointed to by *func* that will be called at normal program termination. The function is called without arguments. Up to 32 functions can be registered.

See Also

`exit()`, ANSI C 4.10.4.2

atof

Converts a string to a double floating-point number.

Syntax

```
#include <stdlib.h>
double atof (const char *str);
```

Parameters

str	A pointer to a character string to be converted to a double floating-point number.

Return Values

x	A double floating-point number.

Description

The `atof` function converts the string of characters that the *`str`* argument points to into a double floating-point number. The `atof` function skips over white space before looking for the start of the number. The format of the input string is the same as that accepted by the `%lf scanf` format conversion.

This function converts any numeric and numeric formatting characters up to, but not after, any non-numeric character that it encounters. In this case, `atof()` returns the number that has been converted up to that point.

See Also

`atoi()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`, ANSI C 4.10.1.1, POSIX.1 8.1

atoi

Converts a string to an integer.

Syntax

```
#include <stdlib.h>
int atoi (const char *str);
```

Parameters

str	A pointer to a character string to convert to an integer.

Return Values

x	An integer value upon successful completion.
0	An error occurred. The *str* argument may have started with an unrecognized character.

Description

The `atoi` function converts the string of characters pointed to by the *str* argument into an integer. The `atoi` function skips over white space before looking for the start of the number. The format of the input string is the same as that accepted by the `%d scanf` format conversion.

This function converts as many characters as possible until it encounters an unrecognized character. For example, if the received string is `"19A1"`, `atoi()` returns `19`.

See Also

`atof()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`, `scanf()`, ANSI C 4.10.1.2, POSIX.1 8.1

atol

Converts a string to a long integer.

Syntax

```
#include
long int atol (const char *str);
```

Parameters

str	A pointer to a character string to be converted to an object of type `long int`.

Return Values

x	A long integer upon successful completion.
0	An error occurred. The *str* argument may have started with an unrecognized character.

Description

The `atol` function converts the string of characters that *str* points to into a long integer (`unsigned long int`) representation. The `atol` function skips over white space before looking for the start of the number. The format of the input string is the same as that accepted by the `%ld scanf` format conversion.

This function converts any characters up to, but not after, any unrecognized character it encounters. In this case, `atol` returns the number that has been converted up to that point.

See Also

`atof()`, `atoi()`, `strtod()`, `strtol()`, `strtoul()`, `scanf()`, ANSI C 4.10.1.3, POSIX.1 8.1

Bessel Functions

The Bessel functions are `j0`, `j1`, `jn`, `y0`, `y1`, and `yn`.

Syntax

```
#include <math.h>

double j0 (double x);

double j1 (double x);

double jn (int i, double x);

double y0 (double x);

double y1 (double x);

double yn (int i, double x);
```

Parameters

x	A real number input to the Bessel functions.
i	An integer value indicating the order to use when calculating the Bessel functions.

Return Values

n	The result of the Bessel function.
`-HUGE`	The input arguments are non-positive.
0	The input argument is too large in magnitude. In addtion, `errno` is set to `ERANGE`.

Description

The `j0` and `j1` functions return Bessel functions of *x* of the first kind of orders zero and 1, respectively. The `jn` function returns the Bessel function of *x* of the first kind of order *i*.

The `y0` and `y1` functions return the Bessel functions of *x* of the second kind of orders zero and 1, respectively. The `yn` function returns the Bessel function of *x* of the second kind of order *i*. The value of *x* must be positive.

Non-positive arguments cause `y0`, `y1`, and `yn` to return the value `-HUGE` and sets `errno` to `EDOM`. They also cause a message indicating a `DOMAIN` error to be printed on the standard error output, but the process continues.

Arguments too large in magnitude cause `j0`, `j1`, `jn`, `y0`, `y1` and `yn` to return zero and to set `errno` to `ERANGE.` In addition, a message indicating `TLOSS` error is printed on the standard error output.

Error handling can be changed by a user-written `matherr` function.

See Also

`matherr()`

bsearch

Performs a binary search of a sorted array.

Syntax

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar) (const void *, const void *));
```

Parameters

key	A pointer to the search pattern to be found in the table.
base	A pointer to the beginning of a table of items to be searched.
nmemb	The number of elements in the array.
size	The total size, in bytes, of each element of the array.
compar	A pointer to the comparison function.

Return Values

x	A pointer to an array element that matches the specified search pattern.
NULL	No match found.

Description

The bsearch function searches an array of *nmemb* objects for a member that matches the object pointed to by *key*. The size of each member of the array is specified by *size*.

The contents of the array must be sorted in ascending order according to the comparison function pointed to by *compar*. The comparison function is called with two arguments that point to the *key* object and to an array member, in that order.

The function must return an integer less than, equal to, or greater than zero indicating if the first argument is to be considered less than, equal to, or greater than the second.

If two search keys in the array are equal to the specified object, the element matched is unspecified.

See Also

hsearch(), lsearch(), qsort(), tsearch(), ANSI C 4.10.5.1, POSIX.1 8.1

calloc

Allocates a block of memory.

Syntax

```
#include <stdlib.h>
void *calloc (size_t nelem, size_t elsize);
```

Parameters

nelem	The number of elements, each of size *elsize*, to be found in the block of allocated memory.
elsize	The size, in bytes, of each element specified in *nelem*.

Return Values

x	A pointer to the allocated space.
`NULL`	There is not enough available memory or *elsize* is zero.

Description

The `calloc` function allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to all bits zero. It is suitably aligned for any use.

See Also

`malloc()`, `free()`, `realloc()`, ANSI C 4.10.3.1, POSIX.1 8.1

catread

Returns a message from a message catalog file in HP-UX format.

Syntax

```
int catread (int fd, int set_num, int msg_num, char *msg_buf,
             int buflen [,char *arg]....);
```

Parameters

fd	An integer containing a file descriptor of the message catalog.
set_num	An integer containing the message set number where the message to be read is located.
msg_num	An integer containing the message number within the set to read from the message catalog.
msg_buf	A pointer to a character array in which the message is returned.
buflen	An integer containing the length of buffer pointed to by *msg_buf*.
arg1..n	Optional pointers to character strings that can be inserted into the error message.

Return Values

≥0	The number of non-null bytes placed in the *msg_buf*. Indicates success.
<0	Indicates *set_num* or *msg_num* is not found in the catalog.

Description

The catread function retrieves messages from message catalogs created on HP-UX or formatted according to the HP-UX message catalog conventions. The catread function is layered on getmsg.

This function provides interoperability support for message catalogs ported to MPE/iX from HP-UX systems. For information on how to read message catalogs created on MPE/iX, refer to the descriptions of the MPE/iX intrinsics CATOPEN, CATCLOSE, and CATREAD which are documented in the *MPE/iX Intrinsics Reference Manual*.

The message read from the catalog may have embedded formatting information in the form ![n], where n is a digit. An exclamation mark followed by n is replaced by the nth argument string. If exclamation marks are not numbered, they are replaced by the arguments in serial order. Either all or none must be numbered.

See Also

getmsg()

ccode

Retrieves the condition code for the calling process.

Syntax

```
#include <mpe.h>
int ccode();
```

Parameters

None.

Return Values

The general meanings of the values returned by ccode are described below. The specific meaning depends upon the intrinsic called. Refer to the individual intrinsic descriptions in the *MPE/iX Intrinsics Reference Manual* for details on the specific meaning.

Value	Condition Code	Description
0	Condition Code Greater Than (CCG)	A special condition occurred but may not have affected the execution of the request.
1	Condition Code Less Than (CCL)	The request was not granted because an error condition occurred.
2	Condition Code Equal (CCE)	This usually indicates that a request was granted.

Description

The ccode function retrieves the two bit condition code for the calling process. A condition code is a process-specific value that provides information about the completion status of system intrinsic functions calls. Many intrinsics use the condition code to signal success, warning, or failure. From the condition code value, you can learn some basic information about what happened during execution of the intrinsic.

ceil

Computes the ceiling function that finds the smallest integer that is greater than or equal to the specified real number.

Syntax

```
#include <math.h>
double ceil (double x);
```

Parameters

x	A real number.

Return Values

n	An integer value of type `double`.

Description

The `ceil` function returns the smallest integer not less than the argument *x*.

See Also

`floor()`, `fmod()`, ANSI C 4.5.6.1, POSIX.1 8.1

clearerr

Clears the end-of-file and error indicators of a stream.

Syntax

```
#include <stdio.h>
void clearerr (FILE *stream);
```

Parameters

stream	A pointer to an open stream.

Return Values

None.

Description

The `clearerr` function clears the end-of-file and error indicators to zero for the file pointed to by *stream*.

See Also

`fopen()`, ANSI C 4.9.10.1, POSIX.1 8.1

clock

Reports CPU time used.

Syntax

```
#include <time.h>
clock_t clock (void)
```

Parameters

None.

Return Values

x	The number of clock ticks consumed by the program.

Description

The clock function returns the amount of CPU time, in microseconds, used since the first call to `clock()`. The time reported is the sum of the user and system times of the calling process.

The resolution of the clock varies, depending on the hardware and on the software configuration. On MPE/iX, the clock resolution is 10 milliseconds.

The value returned by `clock()` is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating only 2147 seconds of CPU time (about 36 minutes).

See Also

ANSI C 4.12.2.1

close

Closes a file.

Syntax

```
int close (int fildes);
```

Parameters

fildes	An open file descriptor.

Return Values

0	A successful close.	
–1	An unsuccessful close and `errno` is set to one of the following values:	
	EBADF	The *fildes* parameter is not a valid open file descriptor.
	ESYSERR	A call to a system intrinsic failed.

Description

The `close` function closes the file indicated by *fildes*. The *fildes* parameter is an open file descriptor obtained from a call to `dup()` or `open()`.

NOTE If linking with the POSIX/iX library, refer to the description of `close()` located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

`dup()`, `open()`, `read()`, `write()`

cos

Computes a cosine value for a given angle.

Syntax

```
#include <math.h>
double cos (double x);
```

Parameters

x	A real number giving the angle measured in radians.

Return Value

n	The cosine of the angle.
0	A complete loss of significance. A TLOSS error message is printed on the standard error output. The external variable errno is set to ERANGE.

Description

The cos function returns the cosine of its argument, *x*, measured in radians.

This function loses accuracy when its argument is far from zero. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed and errno is set to ERANGE.

Error handling can be changed by a user-written matherr function.

See Also

sin(), tan(), matherr(), ANSI C 4.5.2.5, POSIX.1 8.1

cosh

Computes the hyperbolic cosine of an angle.

Syntax

```
#include <math.h>
double cosh (double x);
```

Parameters

x	A real number giving the angle measured in radians.

Return Values

`HUGE_VAL`	An overflow condition occurred, and `errno` is set to `ERANGE`.
n	The hyperbolic cosine of the given angle.

Description

The `cosh` function returns the hyperbolic cosine of the given angle. Error handling can be changed by a user-written `matherr` function.

See Also

`tanh()`, `cos()`, `matherr()`, ANSI C 4.5.3.1, POSIX.1 8.1

creat

Creates a new file or rewrites an existing file.

Syntax

```
#include <fcntl.h>
creat (char *pathname, int mode)
```

Parameters

pathname	A pointer to a string containing the pathname of a file to be created or rewritten. The *pathname* must be terminated by a null character.
mode	The *mode* parameter is ignored. This parameter is provided for compatibility with other systems.

Return Values

≥0	Success. A non-negative integer value is returned representing the lowest unused file descriptor.
−1	An error occurred. No file has been created or modified and `errno` is set to one of the following values:

`EACCES`	A file access permission violation is associated with one of the following: • Search permission is denied within the accessed group or account. • The file does not exist and the group in which the file is to be created does not permit writing. • The file exists and write permission is denied.
`EMPFILE`	More than the maximum number of file descriptors are currently open.
`ENOENT`	The *pathname* is `NULL`.
`ESYSERR`	A call to a system intrinsic failed.

Description

The `creat` function opens for write-only access a file whose pathname is specified in the string pointed to by *pathname*. The file offset is set to the beginning of the file. Upon success, `creat` returns a file descriptor used by other I/O functions to refer to the file.

The function call below:

```
creat (path, mode);
```

is equivalent to the following:

```
open (path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

NOTE If linking with the POSIX/iX library, refer to the description of `creat()` located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

`open()`

crypt

Provides one-way encryption of passwords.

Syntax

```
char *crypt (char *key, char *salt);
```

Parameters

key	A pointer to a character string to be encrypted.
salt	A pointer to a character string used as the initial value in the hashing algorithm.

Return Values

x	A pointer to a character string containing the encrypted password.

Description

The `crypt` function is the password encryption function based on the NBS Data Encryption Standard (DES). It is a one-way algorithm that produces a scrambled character string based upon the input string. It includes variations from the DES intended to frustrate the use of hardware implementations of the DES for key search.

The *key* parameter is your typed password. The *salt* parameter is a two-character string chosen from the set `[a-zA-Z0-9./]`; this string is used to set the hashing algorithm in one of 4096 different ways, after which *key* is used to encrypt repeatedly a constant string. The returned value points to the encrypted password; the first two characters of the password is the *salt* itself.

The return value points to static data that are overwritten by each call.

See Also

`setkey()`, `encrypt()`

ctime

Converts the current time into a 26-character ASCII string of the form

```
Fri May 11 09:53:03 1984\n\0
```

where `\n` is a newline character and `\0` is a terminating null character.

Syntax

```
#include <time.h>
char *ctime(const time_t *timer);
```

Parameters

timer	A pointer to the time to be converted.

Return Value

x	A pointer to a 26-byte character string containing the converted time.

Description

The `ctime` function converts a Coordinated Universal Time (UTC) value (a value representing the number of elapsed seconds since 00:00:00 UTC January 1, 1970) into a character string. The returned 26 character time value is adjusted to the time zone specified by the TZ (Time Zone) environment variable.

By default, `ctime` adjusts the returned value to the Eastern Standard Time (EST) zone. You may override this default behavior by using the MPE/iX command SETVAR TZ *name*. Time zone names, and the format of TZTAB.LIB.SYS file containing time zone offsets from GMT are listed in appendix A, "Time Zones."

Example

Using `time` and `ctime`, you can write a simple `date` command:

```
#include <stdio.h>
#include <time.h>
main()
{
   char *str, *ctime();
   time_t time(), nseconds;
   nseconds = time(NULL);
   str = ctime(&nseconds);
   printf("%s\n", str);
}
```

See Also

`time()`, ANSI C 4.12.3.2, POSIX.1 8.1

difftime

Computes the difference between two times.

Syntax

```
#include <time.h>
double difftime (time_t time2, time_t time1);
```

Parameters

time2	A time, in `time_t` format.
time1	A time, in `time_t` format.

Return Values

x	Returns *time2* - *time1* in seconds as a `double`.

Description

The `difftime` function computes the time difference between *time2* and *time1* in seconds. The *time2* parameter should be the later of the two times.

See Also

`time()`, ANSI C 4.12.2.2

div

Computes the quotient and remainder of two integers.

Syntax

```
#include <stdlib.h>
div_t div (int numer, int denom);
```

Parameters

numer	The numerator.
denom	The denominator.

Return Values

Returns a structure of type `div_t`, comprising the quotient and the remainder. The structure contains the following:

```
int quot;     /* quotient */
int rem;      /* remainder */
```

Description

The `div` function computes and returns the quotient and the remainder of the division of *numer* by *denom*.

If the division is inexact, the sign of the resulting quotient and the algebraic quotient are the same, and the magnitude of the resulting quotients is the largest integer less than the magnitude of the algebraic quotient.

If the result cannot be represented, the behavior is undefined; otherwise, *quot* × *denom* + *rem* equals *numer*.

See Also

`ldiv()`, ANSI C 4.10.6.2

dup

Duplicates an open file descriptor.

Syntax

```
#include <fcntl.h>
int dup (int fildes);
```

Parameters

fildes	A file descriptor.

Return Values

n	A non-negative integer representing the new file descriptor.	
–1	An error occurred and errno is set to one of the following values:	
	EBADF	The *fildes* parameter is not a valid open file descriptor.
	EMFILE	The maximum number of file descriptors are currently open.

Description

The dup function returns the lowest-numbered available file descriptor. The new file descriptor returned by dup( ) refers to the same open file description as *fildes*. The data in the file is not duplicated; only the file descriptor is duplicated.

Using dup( ) to create two file descriptors that point to the same file is different from opening the file twice with open( ). With dup( ), both file descriptors use the same file table entry, and the same file offset is used for reads and writes. With open( ), multiple file descriptors and file table entries are created, and multiple file offset variables are used with reads and writes.

The new file descriptor has the following in common with the original file descriptor:

- Both share the same open file description.
- Both share the same file position indicator.
- Both share the same access mode.

NOTE If linking with the POSIX/iX library, refer to the description of dup( ) located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

open( )

ecvt

Converts a floating-point number to a string.

Syntax

```
char *ecvt (double value, int ndigit, int *decpt, int sign);
```

Parameters

value	The floating-point number to be converted to a character string.
ndigit	The number of digits to convert.
decpt	A pointer to an integer to which the position of the decimal point relative to the beginning of the string is returned.
sign	A pointer to an integer to which a flag indicating the sign of the number is returned.

Return Values

x	A pointer to a character array containing the results of the conversion.

Description

The `ecvt` function converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The resulting numeric string is rounded and left-justified without leading zeros. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero. Otherwise, the word pointed to by *sign* is zero.

The values returned by `ecvt()` point to a single static data array whose content is overwritten by each call.

See Also

`fcvt()`, `gcvt()`

encrypt

Encrypts a block of data.

Syntax

```
void encrypt (char *block, int edflag);
```

Parameters

block	A pointer to the character array that is encrypted.
edflag	An integer that is ignored by the function.

Return Values

None.

Description

The `encrypt` function scrambles the data in *block* using the same hashing algorithm used by `crypt()`. The `encrypt` function performs a one-way encryption on the supplied data in *block* using an encryption key previously defined to the encryption algorithm using `setkey`.

The argument to `encrypt` is an 8-byte character array. The array is treated as a binary number. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by `setkey`.

See Also

`crypt()`, `setkey()`

erf

Returns the statistical error function of the input value.

Syntax

```
#include <math.h>
double erf (double x);
```

Parameters

x	A real number defining the upper limit of the integral.

Return Values

n	The integral given by the error function from 0 to *x*.

Description

The `erf` function returns the error function of *x*, defined as:

Figure 5-1. erf function

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

See Also

`exp()`, `erfc()`

erfc

Returns the complementary error function of the input value.

Syntax

```
#include <math.h>
double erfc (double x);
```

Parameters

x	A real number defining the upper limit of the integral.

Return Values

n	The complement of the integral given by the error function from 0 to *x*.

Description

The `erfc` function returns the complementary error function, `1- erf(x)`. This function is provided because of the extreme loss of relative accuracy when `erf(`*x*`)` is called for large values of *x*. If `erf(5)` is called and the return value subtracted from 1, 12 places of accuracy are lost when compared to calling `erfc(5)`.

See Also

`exp()`, `erf()`

exit

Terminates the calling process normally.

Syntax

```
#include <stdlib.h>
void exit (int status);
```

Parameters

status	A value passed to the environment upon program termination.

Return Values

None.

Description

The `exit` function terminates the calling process. The parameter *status* is returned to the MPE/iX command interpreter using the `CJCW` job control word. By convention, a *status* value of zero (0) indicates `EXIT_SUCCESS`, and a value of one (1) indicates `EXIT_FAILURE`. You may establish additional return values as required.

Using the `return` *expression* statement from `main()` in a C program has the same effect as using `exit()`, where *status* is equivalent to *expression*. This value returned using `CJCW` is undefined if `main()` does not return a value or explicitly call `exit()`.

The `exit` function causes all functions registered by the `atexit` function to be called in the reverse order of their registration. The `exit` function then triggers the system-level clean-up procedures. All output streams are flushed. All stream are closed. All files created by the `tmpfile` function are deleted.

See Also

`abort()`, `atexit()`, ANSI C 4.10.4.3, POSIX.1 8.1

exp

Returns the base *e* raised to the power of the argument.

Syntax

```
#include <math.h>
double exp (double x);
```

Parameters

x	A real number used as the exponent of *e*.

Return Values

n	*e* raised to the power of *x*.
HUGE_VAL	An overflow condition has occurred and `errno` is set to `ERANGE`.
0	An underflow condition has occurred.

Description

`exp` returns e^x.

This function sets `errno` to `ERANGE` when an underflow or overflow occurs.

Error handling can be changed by a user-written `matherr` function.

See Also

matherr(), ANSI C 4.5.4.1, POSIX.1 8.1

fabs

Computes the absolute value of a floating-point argument.

Syntax

```
#include <math.h>
double fabs (double x);
```

Parameters

x	A floating-point value whose absolute value is to be computed.

Return Values

n	The absolute value of the floating-point value specified in *x*.

Example

The following program calculates floating-point absolute values until a zero is entered from the keyboard:

```
#include <math.h>
main()
{
  double value, fabs();
  do
  {
  printf("Enter value: ");
  scanf("%lf", &value);
  if (value == 0)
     exit
  printf("Absolute value of %.12g is %.12g.\n", value, fabs(value));
  }
  while (value !=0);
}
```

See Also

abs(), labs(), ANSI C 4.5.6.2, POSIX.1 8.1

fclose

Closes an open file.

Syntax

```
#include <stdio.h>
int fclose (FILE *stream);
```

Parameters

stream	A pointer to the file to be closed.

Return Values

0	The file is successfully closed.
≠0	An error occurred. The file is not closed.

Description

The `fclose` function flushes the buffer associated with the specified stream, and, if the buffer was allocated automatically by the standard I/O system, frees the space allocated to that buffer. The stream is then closed, breaking the connection between your file pointer and the stream. The `fclose` function closes files opened by the `fopen()`, `fdopen()`, or `freopen()` functions.

The `fclose` function takes a pointer to `FILE` as its argument (returned from a call to `fopen()`, `fdopen()`, or `freopen()`). The function posts any information written to the file that is still in the stream's buffer, and it then closes the file. This disassociates the file and the stream. If the buffer was automatically allocated, it is deallocated.

There are two reasons why you can open a file, but might never explicitly close the file. First, notice that all programs in this chapter that open files end with a call to `exit()`. The `exit()` call automatically performs an `fclose()` operation for every open file in that program. Second, when a C program is compiled, an `exit()` call is normally compiled with your code, so that if you return from `main()` or reach the `}` that terminates `main()`, it is equivalent to calling `exit()`.

See Also

`exit()`, `fdopen()`, `fopen()`, `freopen()`, `setbuf()`, ANSI C 4.9.5.1, POSIX.1 8.1

fcvt

Converts a floating-point number to a string.

Syntax

```
char *fcvt (double value, int ndigit, int *decpt, int *sign);
```

Parameters

value	The floating-point number to be converted to a character string.
ndigit	The number of digits to convert.
decpt	A pointer to an integer to which the position of the decimal point relative to the beginning of the string is returned.
sign	A pointer to an integer to which a flag indicating the sign of the number is returned.

Return Values

x	A pointer to a character array containing the resulting numeric character string.

Description

The `fcvt` function converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The resulting numeric string is rounded and left-justified without leading zeros. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero; otherwise, the word pointed to by *sign* is zero.

This function is identical to `ecvt()`, except that the correct digit has been rounded for `printf %f` (FORTRAN F-format) output of the number of digits specified by *ndigit*.

The `fcvt` function points to a single static data array whose content is overwritten by each call.

See Also

`ecvt()`, `gcvt()`

fdopen

Opens a stream on a file descriptor.

Syntax

```
#include <stdio.h>
FILE *fdopen (int fildes, const char *type);
```

Parameters

fildes	An open file descriptor.
type	A pointer to a character string containing a new access mode. Following are valid strings and their meanings:

`r`	Open or create file for reading.
`w`	Open or create file for writing.
`a`	Open or create file in append mode. All writes are at end-of-file.
`r+`	Open or create file for update (reading and writing).
`w+`	Open or create file for update.
`a+`	Open or create file for append update (read anywhere, but all writes are at end-of-file).

Return Values

x	A pointer to an open stream.
`NULL`	An error occurred. There may be too many open files, or the arguments may have been incorrectly defined.

Description

The `fdopen` function associates a stream with an open file descriptor. File descriptors are obtained from the `open` or `dup` functions; however, streams are the required form of file reference for many of the standard I/O library functions. The type of stream must agree with the mode of the open file.

Opening a file in read mode fails if the file does not exist or cannot be read.

When a file is opened for update, both input and output may be done on the resulting stream. Do not directly follow output with input without an intervening call to `fflush()` or to a file positioning function (`fseek()`, `fsetpos()`, or `rewind()`). Do not directly follow input with output without an intervening call to a file positioning function unless the input operation encounters end-of-file.

When a file is opened for append, it is impossible to overwrite information already in the file. The `fseek` function may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file, and the file pointer is repositioned at the end of the output. When opening a binary file, the file position indicator may, in some cases, be positioned beyond the last data written because of blank or null padding.

When opened, a stream is fully buffered only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for a stream are cleared.

See Also

`open()`, `dup()`, `close()`, POSIX.1 8.1

feof

Tests whether the end-of-file indicator for a stream has been set.

Syntax

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

Parameters

stream	A pointer to a file to be tested.

Return Values

=0	End-of-file has not been set.
≠0	End-of-file has been set.

Description

The `feof` function is intended to clarify ambiguous return values from standard I/O functions.

The `feof` function returns a nonzero value if the end-of-file indicator was set on the specified *stream*. It does not reset the indicator. You need to use the `clearerr` function to reset it.

Because I/O functions return `EOF` for end-of-file and error conditions, you can use `feof()` and `ferror()` to distinguish between them. Also, some systems support I/O functions that take integer data instead of characters. For these functions, you need to use `feof()` and `ferror()` to differentiate between valid data and the `EOF` flag.

Example

The following program uses `feof()`:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[ ];
{
   int c;
   FILE *dfile, *datale, *datagt;

   if(argc != 2) {
      fprintf(stderr, "usage:  intsort filename\n");
      exit(1);
   }
   dfile = fopen(argv[1], "r");
   if(dfile == NULL) {
         fprintf("Can't open %s.\n", argv[1]);
         exit(1);
      }
      datale = fopen("dfle", "w");
      if(datale == NULL) {
         fprintf("Can't create dfle file.\n");
         exit(1);
      }
      datagt = fopen("dfgt", "w");
      if(datagt == NULL) {
         fprintf("Can't create dfgt file.\n");
         exit(1);
      }
      for(;;) {
         if((c = fgetc(dfile)) != EOF) {
            if(c <= 'Z'  c >= 'A')
               fputc(c, datale);
            else
               fputc(c, datagt);
          } else {
               if(feof(dfile))
                  break;
               else
                  fprintf(stderr, "error in reading file \n");
                  exit(1);
          }
      }
      exit(0);
   }
```

Whenever `fgetc()` returns an integer equal to `EOF`, the `feof()` function checks whether the end-of-file has been reached. If the end-of-file has been reached, the loop and the program terminate; if not, an error message is displayed and the program terminates.

See Also

`fopen()`, `ferror()`, ANSI C 4.9.10.2, POSIX.1 8.1

ferror

Tests whether the error indicator for a stream has been set.

Syntax

```
#include <stdio.h>
int ferror (FILE *stream);
```

Parameters

stream	A pointer to a file to be tested.

Return Values

0	Error indicator has not been set.
≠0	Error indicator has been set.

Description

The `ferror` function is intended to clarify ambiguous return values from standard I/O functions.

The `ferror` function returns a nonzero value if the error indicator was set on the specified *stream*. It does not reset the indicator. You need to use the `clearerr` function to reset it.

Because I/O functions return `EOF` for end-of-file and error conditions, you can use `ferror()` and `feof()` to distinguish between them. Also, some systems support I/O functions that take integer data instead of characters. For these functions, you need to use `ferror()` and `feof()` to differentiate between valid data and the `EOF` flag.

See Also

`fopen()`, `feof()`, ANSI C 4.9.10.3, POSIX.1 8.1

fflush

Flushes an I/O buffer to a file.

Syntax

```
#include <stdio.h>
int fflush (FILE *stream);
```

Parameters

stream	A file pointer to an output stream.

Return Values

0	Success.
EOF	An error occurred.

Description

The fflush function causes any information that was buffered by the stream pointed to by the *stream* argument to be flushed out to the associated file. The fflush function returns an EOF if the flush operation caused a write error. It returns a zero if there was no error.

The fclose and exit functions automatically perform fflush(). Therefore, there is often no need to call fflush() explicitly before closing a file or terminating a program. However, it might be necessary to manually fflush() a stream.

For example, data written to a terminal is line buffered by default. This means the system waits for a newline character before writing the buffer onto the terminal screen. There are times when you want whatever has been written so far to be written to the screen without waiting for the newline character. In such situations, you must use fflush.

Another situation when explicit use of the fflush function is needed is when you have written less than a full buffer of data to a file, and you want the contents of that file processed by another function. Because less than a full buffer was written, the data is still in the buffer; the file is still empty. Performing an fflush() causes the buffered data to be written out to the file, enabling other functions or commands to utilize the file's contents.

See Also

fopen(), exit(), setbuf(), ANSI C 4.9.5.2, POSIX.1 8.1

fgetc

Reads a character from an open stream.

Syntax

```
#include <stdio.h>
int fgetc (FILE *stream);
```

Parameters

stream	Pointer to an open stream.

Return Values

x	The character read, expressed as an integer.
EOF	No more input, or an error occurred.

Description

The `fgetc` function reads the next character from the specified stream and advances the file position. The character is returned as an integer. When there are no more input characters, the value `EOF` is returned.

See Also

`fclose()`, `ferror()`, `fopen()`, `fread()`, `getc()`, `gets()`, `putc()`, `fputc()`, `scanf()`, ANSI C 4.9.7.1, POSIX.1 8.1

fgetpos

Returns the current file position of an open stream.

Syntax

```
#include <stdio.h>
int fgetpos (FILE *stream, fpos_t *pos);
```

Parameters

stream	A pointer to an open stream.
pos	A pointer to an object of type `fpos_t`, where the current file position indicator is returned.

Return Values

0	Success.
≠0	An error occurred, and `errno` is set to indicate the error condition.

Description

The `fgetpos` function gets the current value of the file position indicator of the open stream and returns it to the object pointed to by *pos*. The value returned in *pos* contains unspecified data usable by `fsetpos()`.

See Also

`ftell()`, `fsetpos()`, `rewind()`, ANSI C 4.9.9.1

fgets

Reads a string from an open stream.

Syntax

```
#include <stdio.h>
char *fgets (char *string, int n, FILE *stream);
```

Parameters

string	A pointer to a character array.
n	The maximum number of characters to read, plus one.
stream	A pointer to an open stream.

Return Values

x	If successful, a pointer to a character array.
NULL	An error occurred.

Description

The `fgets` function reads a string from an open stream. The *string* parameter is a pointer to a character string, and *stream* is a file pointer to the input stream.

The `fgets` function reads *n*-1 characters or up to a newline character, whichever comes first. If a newline character is encountered, that character is retained as part of the string. (Contrast this with the `gets` function, which replaces the newline character with a null character.)

The `fgets` function appends a null character to the string.

The function returns the pointer to the *string* argument if the read is successful. If an end-of-file is encountered and no characters were read into the array, the contents of the array remain unchanged and a null pointer is returned. A null pointer is also returned if there is a read error. In this case, the contents of the array pointed to by *string* are undefined.

Examples

The following program uses `fgets()` and `fputs()` to copy a file.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[ ];
{
   char c, line[256], *fgets();
   FILE *from, *to;
```

```
      if(argc != 3) {
         printf("Usage:  cp fromfile tofile\n");
         exit(1);
      }
      from = fopen(argv[1], "r");
      if(from == NULL) {
         printf("Can't open %s.\n", argv[1]);
         exit(1);
      }
      to = fopen(argv[2], "w");
      if(to == NULL) {
         printf("Can't create %s.\n", argv[2]);
         exit(1);
      }
      while(fgets(line, 256, from) != NULL)
         fputs(line, to);
      exit(0);
   }
```

The program above accepts two arguments: the first is the name of the file to be copied, and the second is the name of the file to be created. The first file is opened for reading, and the second file is created for writing. The data from the first file is copied directly to the newly created file.

In this program, the return value of `fgets()` is compared to `NULL` in the `while` loop, because `fgets()` returns the null pointer when it reaches the end of its input. You can easily convert this program to a file print command by doing the following:

1. Change the `argc` comparison to

   ```
   if(argc != 2) . . .
   ```

2. Remove the `to` file pointer.
3. Remove the block of code that uses `fopen()` to open the new file, and assign a value to `to`.
4. Change the `fputs()` call to

   ```
   fputs(line, stdout);
   ```

The new file print program should look like this:

```
   #include <stdio.h>
   main(argc, argv)
   int argc;
   char *argv[ ];
   {
      char c, line[256], *fgets();
      FILE *from;
      if(argc < 2) {
         printf("Usage:  cat file\n");
         exit(1);
      }

      from = fopen(argv[1], "r");
      if(from == NULL) {
```

```
         printf("Can't open %s.\n", argv[1]);
         exit(1);
      }
      while(fgets(line, 256, from) != NULL)
         fputs(line, stdout);
      exit(0);
   }
```

See Also

`ferror()`, `fopen()`, `fread()`, `getc()`, `puts()`, `scanf()` ANSI C 4.9.7.2, POSIX.1 8.1

fileno

Maps a stream pointer to a file descriptor.

Syntax

```
#include <stdio.h>
int fileno (FILE *stream);
```

Parameters

stream	A pointer to an open stream.

Return Values

≥0	An open file descriptor associated with *stream*.

Description

The `fileno` function returns the file descriptor associated with *stream*.

The following symbolic values, located in `<unistd.h>`, define the file descriptors associated with `stdin`, `stdout`, and `stderr` streams when the application is started:

File Descriptor Symbolic Value	Stream Description	Value
STDIN_FILENO	Standard input stream `stdin`	0
STDOUT_FILENO	Standard output stream `stdout`	1
STDERR_FILENO	Standard error stream `stderr`	2

This routine is implemented as a macro in `<stdio.h>` and as a function.

See Also

`fdopen()`, `open()`, POSIX.1 8.2.1

floor

Computes the largest integer value that is less than or equal to its argument.

Syntax

```
#include <math.h>
double floor (double x);
```

Parameters

x	A real number.

Return Values

n	An integer value stored as a `double`.

Description

The `floor` function returns the largest integer not greater than *x*.

See Also

ANSI C 4.5.6.3, POSIX.1 8.1

fmod

Returns the floating-point remainder of *x* divided by *y*.

Syntax

```
#include <math.h>
double fmod (double x, double y);
```

Parameters

x	The numerator.
y	The divisor.

Return Values

f	The remainder of *x*/*y*.
`NaN`	Neither *x* or *y* is a number, *x* is `+INFINITY`, or *y* is zero. In addition, `errno` is set to `EDOM`.
x	An underflow condition has occurred; *y* may be ±infinity.
0	An overflow condition has occurred.

Description

The `fmod` function returns the floating-point remainder of the division of *x* by *y*. Zero is returned if *y* is zero or if *x*/*y* overflows. Otherwise the number `f` with the same sign as *x* is returned, such that *x* =`i`*y* + `f` for some integer `i` and |`f`| |*y*|.

See Also

`floor()`, `ceil()`, `fabs()`, ANSI C 4.5.6.4, POSIX.1 8.1

fopen

Opens a stream.

Syntax

```
#include <stdio.h>
FILE *fopen (const char *fname, const char *mode);
```

Parameters

fname	A pointer to a character string containing the name of the file.
mode	A pointer to a character string defining the mode of the file open.

Return Values

x	If successful, a pointer to the `FILE` structure associated with the stream.
`NULL`	The file open operation failed.

Description

The `fopen` function opens the file named by *fname* and associates a stream with it. This function returns a pointer to the `FILE` structure associated with the stream.

Opening a file in read mode fails if the file does not exist or cannot be read.

When a file is opened for update, both input and output may be done on the resulting stream. Do not directly follow output with input without an intervening call to `fflush()` or to a file positioning function (`fseek()`, `fsetpos()`, or `rewind()`). Do not directly follow input with output without an intervening call to a file positioning function unless the input operation encounters end-of-file.

When a file is opened for appending, it is impossible to overwrite information already in the file. The `fseek` function can be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and the file pointer is repositioned at the end of the output. When opening a binary file the file position indicator may, in some cases, be positioned beyond the last data written because of blank or null padding.

When opened, a stream is fully buffered only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for a stream are cleared.

The *mode* parameter points to a character string beginning with one of the following sequences:

`r`	Open or create text stream for reading.
`w`	Open or create text stream for writing. Truncate to zero length.
`a`	Open or create text stream in append mode. All writes are at end-of-file.

`rb`	Open or create binary stream for reading.
`wb`	Open or create binary stream for writing. Truncate to zero length.
`ab`	Open or create binary stream in append mode. All writes are at end-of-file.
`r+`	Open or create text stream for update (reading and writing).
`w+`	Open or create text stream for update. Truncate to zero length.
`a+`	Open or create text stream for append update (read anywhere but all writes at end-of-file).
`r+b` or `rb+`	Open or create binary stream for update (reading and writing).
`w+b` or `wb+`	Open or create binary stream for update. Truncate to zero length.
`a+b` or `ab+`	Open or create binary stream for append update (reading anywhere but all writes to end-of-file).

NOTE If you are linking with the POSIX/iX library, the `fopen` function only creates or opens byte stream files. Attempting to open any other file type results in an error.

When `fopen()` parses *mode*, all cases of `b` are ignored in the standard options, and an MPE byte stream format file is opened and a binary stream is associated with it. In addition, all other *mode* options specified below are invalid.

If you are linking with the HP C/iX library, there are several enhancements that provide greater control in the MPE file environment. These options should follow the standard options in the *mode* string. Spaces may be used in the *mode* string to improve the readability of the file's open mode. Notice that the case of the option is important. An upper case `B` is different from a lower case `b`.

`Bl`*n*	The `Bl` option specifies the blocking factor to use if this call to `fopen()` creates the file. The option character is followed by an integer that indicates the blocking factor. If the `Bl` option is not specified, then the default is one record per block.
`Bs`	If the `Bs` option is specified, the file is opened or created as a byte stream file. This is the only required option for opening byte stream files. The maximum file size for a byte stream file is two gigabytes. If specified, the `Rn` option is ignored. The `S`*n* option can be used to reset the file size. This option is mutually exclusive with the `V` option. If the `Bs` or `V` options are not specified, the file is created with an MPE fixed-length record format.
`Bu`*n*	The `Bu` option specifies the number of buffers to be allocated to this file. If the `Bu` option is not specified, the default is 2.
`C`	If the `C` option is specified, then the file will accept carriage control information. The default is to not have carriage control.
`Df`*n*	The `Df` option specifies the final disposition of the file after the file is

closed. The affect of each value of *n* is defined as follows:

0	Don't change the disposition.
1	Save the file as a permanent file.
2	Save the file as a temporary file.
3	Don't rewind on close.
4	Purge the file on close.

If the `Df` option is not specified and the file is a new file, then the default is to save the file as a permanent file. If the file is old, the default is not to change the disposition.

`Ds`*n* The `Ds` option specifies the disk space disposition of the file after the file is closed for fixed, undefined, and variable format files. The effect of each value of *n* is defined as follows:

0	Don't return any disk space allocated beyond the end-of-file indicator.
1	Return to the system any disk space allocated beyond the end-of-file indicator. The EOF becomes the file limit. No records may be added to the file beyond this new limit.
2	Return to the system any disk space allocated beyond the end-of-file indicator, but do not set the file limit to EOF, and allow records to be added to the file up to the file limit.

If the `Ds` option is not specified, the default is not to return any disk space allocated beyond the end-of-file indicator.

`E`*n* The `E` option specifies the maximum number of extents that can be allocated to the file.

The maximum value is 32. The default value, if the `E` option is not specified, is 8 extents.

`F`*n* The `F` option indicates the value used as the file code if this call to `fopen()` creates the file. If the `F` option is not specified, the file code is zero.

`L` If specified, the `L` option indicates that dynamic locking should be allowed on this file.

`M`*n* The `M` option controls multi-access. The option character is followed by an integer that indicates the level of multi-access for this open request. The levels are specified in the *MPE/iX Intrinsics Reference Manual* under the `FOPEN` intrinsic description.

`Q` If the `Q` option is specified, file equations are disallowed. The default is to allow file equations.

`R`*n* The `R` option specifies the size of the record if the file is created by this open request. If the `V` option is also used, this option specifies the maximum size of the variable-sized records. The option letter is followed by a decimal number that is equal to the number of bytes in the record size. Notice that the number must be positive. A byte count is always

	specified. The default for text and binary streams is 256 bytes. The default for byte streams is 1 byte.
`S`*n*	The `S` option specifies the maximum size of the file. The value of *n* is the maximum size of the file in records for text and binary streams, and in bytes for byte streams. Notice that if the `S` parameter is not specified, the default is 4095.
`Te`	If the `Te` option is specified, the file is saved in the temporary file domain. If the `Te` option is not specified and the file is a new file, the default is to save the file as a permanent file. If the file is old, the default is to not change the disposition.
`Tm`	If the `Tm` option is specified, disk read functions trim editor line numbers, if they exist, and trailing blanks from each record of an ASCII fixed record length file before returning file data to the reader. This option is used on files opened with read only access. Random access to file data using `fseek()` and `lseek()` is not permitted. The default is to not trim editor line numbers and blanks.
`U`*n*	If the `U` option is specified, the file is created with *n* user-label records. If this option is not specified, the default is no user-label records.
`V`	If the `V` option is specified, the file is created with an MPE variable-length record format. If the `V` or `Bs` options are not specified, then the file is created with an MPE fixed-length record format. This option is mutually exclusive with the `Bs` option.
`X`*n*	The `X` option controls exclusive access ability for the file. The option character is followed by an integer that indicates the level of exclusivity for this open request. The levels are specified in the *MPE Intrinsics Reference Manual* under the `FOPEN` specification.

The following example creates or opens a fixed record binary file for writing with 256 byte records, a file size of 1000 records, and a file code of 1030:

```
#include stdio.h>

FILE *stream;

stream = fopen("filename","wb R256 S1000 F1030");
```

See Also

`fclose()`, `freopen()`, `fflush()`, ANSI C 4.9.5.3, POSIX.1 8.1

fprintf

Writes data in formatted form to an open stream.

Syntax

```
#include <stdio.h>
int fprintf (FILE *stream, const char *format
            [,item [,item]...] );
```

Parameters

stream	A pointer to an open stream where data is to be written.
format	A pointer to a character string defining the format of the data to be written (or the character string itself enclosed in double quotes).
item,...	Each *item* is a variable or expression specifying the data to write.

Return Values

≥0	The number of characters written.
<0	An error occurred.

Description

The `fprintf` function enables you to output data in formatted form to an open stream. In the `fprintf` function, *string* is a pointer to an open stream, and *format* is a pointer to a character string (or the character string itself enclosed in double quotes) that specifies the format and content of the data to be written. Each *item* is a variable or expression specifying the data to write.

The `fprintf` format is similar to the `printf` function. It is made up of conversion specifications and literal characters. Literal characters are all characters that are not part of a conversion specification. Literal characters are written to the open stream exactly as they appear in the format.

Conversion Specifications

The following list shows the different components of a conversion specification in their correct sequence:

1. A percent sign (`%`), which signals the beginning of a conversion specification; to output a literal percent sign, you must type two percent signs (`%%`).
2. Zero or more flags, which affect the way a value is written (see below).
3. An optional decimal digit string, which specifies a minimum `field width`.
4. An optional *precision* consisting of a dot (`.`) followed by a decimal digit string.

5. An optional `l`, `h`, or `L` indicating that the argument is of an alternate type. When used in conjunction with an integer conversion character, an `l` or `h` indicates a long or short integer argument, respectively. When used in conjunction with a floating-point conversion character, an `L` indicates a long double argument.
6. A conversion character, which indicates the type of data to be converted and printed.

A one-to-one correlation must exist between each specification encountered and each item in the item list.

The available flags are:

`-`	Causes the data to be left-justified within its output field. Normally, the data is right-justified.
`+`	Causes all signed data to begin with a sign (`+` or `-`). Normally, only negative values have signs.
`blank`	Causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the `blank` and `+` flags both appear, the `blank` flag is ignored.
`#`	Causes the data to be written in an alternate form. Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.
`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 `-` flag s both appear, the `0` flag is ignored. The `0` flag is also ignored for `d`, `i`, `o`, `u`, `x`, and `X` conversions if a precision is specified.

A *`field width`*, if specified, determines the minimum number of spaces allocated to the output field for the particular piece of data being printed. If the data happens to be smaller than the field width, the data is blank-padded on the left (or on the right, if the `-` flag is specified) to fill the field. If the data is larger than *`field width`*, the *`field width`* is simply expanded to accommodate the data. An insufficient *`field width`* never causes data to be truncated. If *`field width`* is not specified, the resulting field is made just large enough to hold the data.

The *`precision`* is a value that means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

NOTE A *`field width`* or *`precision`* may be replaced by an asterisk (`*`). If so, the next item in the item list is fetched, and its value is used as the *`field width`* or *`precision`*. The *`item`* fetched must be an integer.

Conversion Characters

Conversion characters specify the type of data to expect in the item list and cause the data

to be formatted and printed appropriately. The integer conversion characters are:

`d`, `i`	An integer *`item`* is converted to signed decimal. The *`precision`*, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the *`precision`*, the value is expanded with leading zeros. The default *`precision`* is 1. A null string results if a zero value is printed with a zero *`precision`*. The # flag has no effect.
`u`	An integer *`item`* is converted to unsigned decimal. The effects of the *`precision`* and the # flag are the same as for `d`.
`o`	An integer *`item`* is converted to unsigned octal. The # flag, if specified, causes *`precision`* to be expanded, and the octal value is printed with a leading zero (a C convention). The *`precision`* parameter behaves the same as in `d` above, except that writing a zero value with a zero *`precision`* results in only the leading zero being written, if the # flag is specified.
`x`	An integer *`item`* is converted to hexadecimal. The letters `abcdef` are used in writing hexadecimal values. The # flag, if specified, causes the *`precision`* to be expanded, and the hexadecimal value is written with a leading `"0x"` (a C convention). The *`precision`* behaves as in `d` above, except that writing a zero value with a zero *`precision`* results in only the leading `"0x"` being written, if the # flag is specified.
`X`	Same as `x` above, except that the letters `ABCDEF` are used to write the hexadecimal value, and the # flag causes the value to be written with a leading `"0X"`.

The character conversion characters are as follows:

`c`	The character specified by the `char` *`item`* is written. The *`precision`* is meaningless, and the # flag has no effect.
`s`	The string pointed to by the character pointer *`item`* is written. If a *`precision`* is specified, characters from the string are written until the number of characters indicated by the *`precision`* is reached or until a null character is encountered, whichever comes first. If the *`precision`* is omitted, all characters up to the first null character are written. The # flag has no effect.

The floating-point conversion characters are:

`f` The `float` or `double` *`item`* is converted to decimal notation in style `f`; that is, in the form

```
[-]ddd.ddd
```

where the number of digits after the decimal point is equal to the *`precision`*. If *`precision`* is not specified, six digits are written after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. If the # flag is specified, a decimal point always appears, even if no digits follow the decimal point.

`e` The `float` or `double` *`item`* is converted to scientific notation in style `e`; that is, in the form

`[-]d.ddde±ddd`

where there is always one digit before the decimal point. The number of digits after the decimal point is equal to *precision*. If *precision* is not given, six digits are written after the decimal point. If the *precision* is explicitly zero, the decimal point is eliminated entirely. The exponent always contains exactly three digits. If the # flag is specified, the result always contains a decimal point, even if no digits follow the decimal point.

`E`	Same as `e` above, except that `E` is used to introduce the exponent instead of `e` (style `E`).
`g`	The `float` or `double` *item* is converted to either style `f` or style `e`, depending on the size of the exponent. If the exponent resulting from the conversion is less than -4 or greater than the *precision*, style `e` is used. Otherwise, style `f` is used. The *precision* specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the # flag is specified, the result always has a decimal point, even if no digits follow the decimal point, and trailing zeros are *not* removed.
`G`	Same as the `g` conversion above, except that style `E` is used instead of style `e`.

Other conversion characters are:

`p`	The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters.
`n`	The argument is a pointer to an integer into which is written the number of characters written to the output stream so far by this call to `fprintf()`. No argument is converted.
`%`	A % is written. No argument is converted. The complete conversion specification is &%&%.

The *item*s in the item list may be variable names or expressions. Note that, with the exception of the `s` conversion, pointers are *not* required in the item list. If the `s` conversion is used, a pointer to a character string must be specified.

Example

The following program illustrates the use of the `fscanf()` and `fprintf()` functions:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[ ];
{
   int count = 0;
   FILE *file;

   if(argc != 2) {
      fprintf(stderr, "Usage:  wdcnt filename\n");
      exit(1);
      }

      file = fopen(argv[1], "r");
      if(file == NULL) {
         fprintf(stderr, "Can't open %s.\n", argv[1]);
         exit(1);
      }

      while(fscanf(file, "%*s") != EOF)
         count;

      fprintf(stdout, "Number of words found:  %d\n", count);

      exit(0);
   }
```

This program counts the number of "words" in the file specified as its only argument. A word is defined as a string of nonspace characters.

In this program, `fprintf()` directs error messages to `stderr.` Error output is written on a different stream than normal output; the error output (or the normal output) can be redirected to another destination. For example, invoking the program with `stderr` set to the file `errmsgs` causes all output from erroneous conditions to be collected in the `errmsgs` file. In this example, this capability is trivial because the program terminates on any error. However, this is a very useful capability for programs that output any number of warnings without terminating. Not only does this command keep the desired output uncluttered with error messages, but it also enables you to save the output. Thus, it is good programming practice to write error messages and warnings on `stderr` and to use `stdout` (or any destination file) to output normal data.

See Also

`putc()`, `scanf()`, `printf()`, `vprintf()`, `vfprintf()`, `fscanf()`, ANSI C 4.9.6.1, POSIX.1 8.1

fprintmsg

Prints formatted output with numbered arguments.

Syntax

```
#include <stdio.h>
int fprintmsg (file *stream, char *format [, arg] ...);
```

Parameters

stream	A stream file pointer to which the output is directed.
format	A pointer to the string containing formatting information to be output. *format* contains optional placeholders and formatting specifications where *arg1* thru *argn* are to be substituted.
arg1 ... argn	A character, character pointer, or integer value giving the parameter to be converted, formatted, and merged with *format* prior to output.

Return Values

x	The number of characters transmitted.
EOF	Indicates failure.

Description

The `fprintmsg` function places formatted output on the file or device indicated by *stream* after performing the parameter substitution. This function is derived from `printf`. In `fprintmsg`, the conversion character `%` is replaced by the sequence `%n$`. The `n` is a decimal digit in the range 1-9, and indicates that this conversion should be applied to the `n`th argument, rather than to the next unused one. All other aspects of formatting are unchanged. All conversion specifications must contain the `%n$` sequence, and you should make sure the numbering is correct. All parameters must be used exactly once. See `printf` for more details on formatting and conversion specifications.

See Also

`printf()`, `printmsg()`, `sprintmsg()`

fputc

Writes a character to an output stream.

Syntax

```
#include <stdio.h>
int fputc (int c, FILE *stream);
```

Parameters

`c`	A character, expressed as an integer, to be written to the output stream.
`stream`	A pointer to an open stream.

Return Values

x	The character written, expressed as an integer.
`EOF`	An error occurred.

Description

The `fputc` function writes the character specified in *`c`* to the specified stream and advances the file position. The character is returned as an integer.

If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

See Also

`fclose()`, `ferror()`, `fopen()`, `fwrite()`, `fprintf()`, `getc()`, `gets()`, `putc()`, `puts()`, `fputc()`, `setbuf()`, ANSI C 4.9.7.3, POSIX.1 8.1

fputs

Writes a string to an output stream.

Syntax

```
#include <stdio.h>
int fputs (const char *string, FILE *stream);
```

Parameters

string	A pointer to an array of characters.
stream	A pointer to an output stream.

Return Values

≥0	Success.
`EOF`	An error occurred.

Description

The `fputs` function writes the character string pointed to by *string* to the specified stream, stopping when a null character is encountered. The terminating null character is not written.

Examples

Refer to the examples located in the `fgets` function description.

See Also

`ferror()`, `fopen()`, `fread()`, `printf()`, `fgets()`, `putc()`, ANSI C 4.9.7.4, POSIX.1 8.1

fread

Reads data items from an open stream.

Syntax

```
#include <stdio.h>
size_t fread (void *fileptr, size_t size, size_t nitems,
   FILE *stream);
```

Parameters

fileptr	A pointer to a buffer to hold the data. The type of the buffer is determined by the type of the data being read.
size	The size of each data item, in bytes.
nitems	The number of data items to read.
stream	A pointer to an open stream.

Return Values

>0	The number of items actually read.
0	Either `EOF` was detected or an error occurred.

Description

The `fread` function reads *nitems* times *size* number of bytes into the buffer pointed to by the *fileptr* argument. The bytes are read from the stream pointed to by the *stream* argument.

This function returns the number of items actually read. This may be less than the number of requested items if an error occurs or an end of file is encountered.

NOTE You must use special care when using `fread()` to access ASCII files from the MPE/iX file system. These files are record-oriented text streams. The `fread` function returns a new line character, `\n`, whenever it reaches an end of record. You must take the `\n` character into account when reading, writing, or repositioning within an MPE/iX fixed length ASCII file.

Binary MPE files behave differently from ASCII MPE files. See chapter 2, "HP C/iX Library Input and Output," for more information.

Example

The following program keeps track of employee data. Each employee is described in a single structure.

```
#include <stdio.h>
struct emp {
          char       name[40]; /* name */
          char       job[40];  /* job title */
          long       salary;   /* salary */
          char       hire[6]   /* hire date */
          char       curve[2]  /* pay curve */
          int        rank;     /* percentile ranking */
}
#define EMPS 400               /* no.  of employees */
main()
{
     int items;
     struct emp staff[EMPS];
     FILE *data;

     data = fopen("empdata", "r");
     if(data == NULL) {
        fprintf(stderr, "Can't open employee data file.\n");
        exit(1);
     }

     items = fread((char *)staff, sizeof(staff[0]), EMPS, data);
     if(items != EMPS) {
         fprintf(stderr, "Insufficient data found.\n");
        exit(1);
     }

     fclose(data);
     archive("empdata");

/* Employee information processing goes here. */
        …
/* Processing is done.  Write out new employee records. */

     data = fopen("empdata", "w");
     if(data == NULL) {
        fprintf(stderr, "Can't create new employee file.\n");
        exit(1);
     }

     items = fwrite((char *)staff, sizeof(staff[0]), EMPS, data);
     if(items != EMPS) {
        fprintf(stderr, "Write error!\n");
        exit(1);
     }

     exit(0);
}
archive(filename)
char *filename;
{
    …
}
```

This program reads the employee information contained in the binary file `empdata.` The data in this file consists of concatenated streams of bytes describing each employee in a 400-employee company. The bytes are written such that, when read correctly, they correspond exactly with the `emp` structure defined in the program. The `staff` array is an array of structures containing one structure for each employee.

In the `fread()` call, the `sizeof(staff[0])` expression returns the number of bytes in the `emp` structure. Because the same number of bytes are in each employee structure, any element of the `staff` array can be specified as the `sizeof` argument; `staff`[0] is used in this example. By counting the number of bytes in each structure member, you can approximate the number of bytes returned by the `sizeof` operator (in this example, 40 + 40 + 8 + 6 + 2 + 4 = 100 bytes). This might vary due to padding performed by a programming language or by machine architecture. Specifying EMPS as the *`nitems`* argument tells `fread` to read 400 structures. Thus, 100 x 400 = 40000 bytes are read, filling in the information for the members of each structure contained in the `staff` array. The `fread()` and `fwrite()` functions can read or write any type of data.

The following examples show some `fread()` calls that read different types of data:

To read a long integer:

```
long nint;
fread((void *)&nint, sizeof(nint), 1, stream);
```

To read an array of 100 long integers:

```
long nint[100];
fread((void *)nint, sizeof(nint[0]), 100, stream);
```

To read a double-precision floating-point value:

```
double fpoint;
fread((void *)&fpoint, sizeof(fpoint), 1, stream);
```

To read an array of 50 floating-point values:

```
float fpoint[50];
fread((void *)fpoint, sizeof(fpoint[0]), 50, stream);
```

See Also

`fgetc()`, `getc()`, `gets()`, `getchar()`, `fscanf()`, `scanf()`, ANSI C 4.9.8.1, POSIX.1 8.1

free

Frees a block of allocated memory.

Syntax

```
#include <stdlib.h>
void free (void *ptr);
```

Parameters

ptr	A pointer to a block of memory previously allocated by a call to `calloc()`, `malloc()`, or `realloc()`.

Return Values

None.

Description

The `free` function frees the block of memory pointed to by *ptr*, making the space available for further allocation. The contents of the block are destroyed. The argument to `free()` is a pointer to a block previously allocated by `calloc()`, `malloc()`, or `realloc()`.

The `malloc` and `free` functions provide a simple generalized memory allocation package.

Undefined results occur if some random pointer is handed to `free()`.

See Also

`calloc()`, `malloc()`, `realloc()`, ANSI C 4.10.3.2, POSIX.1 8.1

freopen

Closes and reopens a stream.

Syntax

```
#include <stdio.h>
FILE *freopen (const char *fname, const char *type,
               FILE *stream);
```

Parameters

fname	A pointer to a character string that contains the name of the file to be opened.
type	A pointer to a character string defining the mode of the file open.
stream	A pointer to an open stream.

Return Values

x	If successful, a pointer to the `FILE` structure associated with the stream.
`NULL`	The file open operation failed.

Description

The `freopen` function substitutes the named file in place of the open *stream*. The original stream is closed, regardless of whether the open succeeds (close errors are ignored). This function returns a pointer to new *stream*.

This function is typically used to attach the preopened streams associated with `stdin`, `stdout` and `stderr` to other files.

Opening a file in read mode fails if the file does not exist or cannot be read.

When a file is opened for update, both input and output may be done on the resulting stream. Do not directly follow output with input without an intervening call to `fflush()` or to a file positioning function (`fseek()`, `fsetpos()`, or `rewind()`). Do not directly follow input with output without an intervening call to a file positioning function unless the input operation encounters end-of-file.

When a file is opened for appending, it is impossible to overwrite information already in the file. `fseek()` may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and the file pointer is repositioned at the end of the output. When opening a binary file the file position indicator may, in some cases, be positioned beyond the last data written, because of blank or null padding.

When opened, a stream is fully buffered only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for a stream are cleared.

The *type* parameter points to a character string beginning with one of the following sequences:

`r`	Open or create text stream for reading.
`w`	Open or create text stream for writing. Truncate to zero length.
`a`	Open or create text stream in append mode. All writes are at end-of-file.
`rb`	Open or create binary stream for reading.
`wb`	Open or create binary stream for writing. Truncate to zero length.
`ab`	Open or create binary stream in append mode. All writes are at end-of-file.
`r+`	Open or create text stream for update (reading and writing).
`w+`	Open or create text stream for update. Truncate to zero length.
`a+`	Open or create text stream for append update (read anywhere but all writes at end-of-file).
`r+b` or `rb+`	Open or create binary stream for update (reading and writing).
`w+b` or `wb+`	Open or create binary stream for update. Truncate to zero length.
`a+b` or `ab+`	Open or create binary stream for append update (reading anywhere but all writes to end-of-file).

NOTE If you are linking with the POSIX/iX library, `freopen()` parses *type* and ignores all cases where `b` is specified. An MPE byte stream format file is opened and a binary stream is associated with it. In addition, all other *type* options specified below are invalid.

If you are linking with the HP C/iX library, there are several enhancements that provide greater control in the MPE file environment. These options should follow the standard options in the *type* string. Spaces can be used in the *type* string to improve the readability of the file's open type. Notice that the case of the option is important. An uppercase `B` is different from a lowercase `b`.

These options are the same options that are used by the `fopen()` function. For a detailed description of these options, refer to the description of `fopen()`.

`Bl`*n*	The `Bl` option specifies the blocking factor to use if this call to `freopen()` creates the file. The option character is followed by an integer that indicates the blocking factor. If the `Bl` option is not specified, then the default is one record per block.
`Bs`	If the `Bs` option is specified, the file is opened or created as a byte stream file. This is the only required option for opening byte stream files. The maximum file size for a byte stream file is two gigabytes. If specified, the `Rn` option is ignored. The `S`*n* option can be used to reset the file size. This option is mutually exclusive with the `V` option. If the `Bs` or `V` options are not specified, the file is created with an MPE fixed-length record format.

`Bu`*`n`*	The `Bu` option specifies the number of buffers to be allocated to this file. If the `Bu` option is not specified, the default is 2.
`C`	If the `C` option is specified, then the file accepts carriage control information. The default is to not have carriage control.
`Df`*`n`*	The `Df` option specifies the final disposition of the file after the file is closed. The affect of each value of *`n`* is defined as follows:

0	Don't change the disposition.
1	Save the file as a permanent file.
2	Save the file as a temporary file.
3	Don't rewind on close.
4	Purge the file on close.

If the `Df` option is not specified and the file is a new file, then the default is to save the file as a permanent file. If the file is old, the default is not to change the disposition.

`Ds`*`n`*	The `Ds` option specifies the disk space disposition of the file after the file is closed for fixed, undefined, and variable format files. The affect of each value of *`n`* is defined as follows:

0	Don't return any disk space allocated beyond the end-of-file indicator.
1	Return to the system any disk space allocated beyond the end-of-file indicator. The EOF becomes the file limit. No records may be added to the file beyond this new limit.
2	Return to the system any disk space allocated beyond the end-of-file indicator, but do not set the file limit to EOF, and allow records to be added to the file up to the file limit.

If the `Ds` option is not specified, the default is not to return any disk space allocated beyond the end-of-file indicator.

`E`*`n`*	The `E` option specifies the maximum number of extents that is allocated to the file. The maximum value is 32. The default value, if the `E` option is not specified, is 8 extents.
`F`*`n`*	The `F` option indicates the value used as the file code if this call to `freopen()` creates the file. If the `F` option is not specified, the file code is zero.
`L`	If specified, the `L` option indicates that dynamic locking should be allowed on this file.
`M`*`n`*	The `M` option controls multi-access. The option character is followed by an integer that indicates the level of multi-access for this open request. The levels are specified in the *MPE Intrinsics Reference Manual* under the `FOPEN` specification.
`Q`	If the `Q` option is specified, file equations are disallowed. The default is to allow file equations.

`R`*n*	The `R` option specifies the size of the record if the file is created by this open request. If the `V` option is also used, this option specifies the maximum size of the variable sized records. The option letter is followed by a decimal number that is equal to the number of bytes in the record size. Notice that the number must be positive. A byte count is always specified. The default for text and binary streams is 256 bytes. The default for byte streams is 1 byte.
`S`*n*	The `S` option specifies the maximum size of the file. The value of *n* is the maximum size of the file in records for text and binary streams, and in bytes for byte streams. The default for text and binary streams is 4095 records. The default for byte streams is 2 gigabytes.
`Te`	If the `Te` option is specified, the file is saved in the temporary file domain. If the `Te` option is not specified and the file is a new file, the default is to save the file as a permanent file. If the file is old, the default is to not change the disposition.
`Tm`	If the `Tm` option is specified, disk read functions trim editor line numbers, if they exist, and trailing blanks from each record of an ASCII fixed record length file before returning file data to the reader. This option is used on files opened with read only access. Random access to file data using `fseek()` and `lseek()`, is not permitted. The default is not to trim editor line numbers and blanks.
`U`*n*	If the `U` option is specified, the file is created with *n* user-label records. If this option is not specified, the default is no user-label records.
`V`	If the `V` option is specified, the file is created with an MPE variable-length record format. This option is mutually exclusive with the `Bs` option. If the `V` or `Bs` options are not specified, then the file is created with an MPE fixed-length record format.
`X`*n*	The `X` option controls exclusive access ability for the file. The option character is followed by an integer that indicates the level of exclusivity for this open request. The levels are specified in the *MPE/iX Intrinsics Reference Manual* under the `FOPEN` intrinsic description.

The following example creates or opens a stream associated with a fixed record ASCII file for writing with 80-byte records and a file size of 1000 records:

```
#include <stdio.h>

FILE *stream;

stream = freopen("filename","w R80 S1000",stdout);
```

See Also

`fclose()`, `fopen()`, `fflush()`, ANSI C 4.9.5.4, POSIX.1 8.1

frexp

Breaks a floating-point number into a normalized fraction and an integral power of 2.

Syntax

```
#include <math.h>
double frexp (double value, int *eptr);
```

Parameters

value	A real number input to the function.
eptr	A pointer to the integer exponent returned by the function.

Return Values

x	A real number between 0.5 and 1.

Description

The `frexp` function accepts a `double` *value*, and returns two values, *x* and *n*, such that

value == *x* * 2^n

where *x* is a `double` quantity in the range 0.5 < *x* < 1, and *n* is an integer exponent. In the `frexp` function, *value* is the value to be processed, and *eptr* is a pointer to an integer variable where the exponent *n* is to be stored. The quantity *x* is the return value of `frexp`.

Example

The following program accepts a number argument and uses `frexp` to output that number's representation in the form shown above:

```
main(argc, argv)
int argc;
char *argv[ ];
{
   double value, x, frexp();
   int eptr;


   printf("%g = %g * 2^%d\n", value, x, eptr);
}
```

See Also

`ldexp()`, `modf()`, ANSI C 4.5.4.2, POSIX.1 8.1

fscanf

Reads externally formatted data from an open stream.

Syntax

```
#include <stdio.h>
int fscanf (FILE *stream, const char *format
            [,item [,item]...] );
```

Parameters

stream	A pointer to an open stream from which data is to be read.
format	A pointer to a character string defining the format of the data to be read (or the character string itself enclosed in double quotes).
item	Each *item* is the address of a variable into which the data will be placed. Refer below to descriptions of conversion specifications.

Return Values

≥0	The number of successfully matched and assigned input items.
EOF	An error occurred on input (no input characters, or a matching error occurred before any conversion).

Description

The `fscanf` function reads externally formatted data from an open stream, converts the data to internal format, and stores the results in a group of arguments. The format consists of white-space characters, conversion specifications, and literal characters.

This function behaves identically to the `scanf` function except that `fscanf()` reads data from an open input stream instead of from `stdin`.

White-Space Characters

White-space characters (blanks, tabs, newlines, or form feeds) cause input to be read up to the next non-white-space character.

Conversion Specifications

A conversion specification is a character sequence that tells `fscanf()` how to interpret the data received at that point in the input.

In the format, a conversion specification is introduced by a percent sign `(%)`, optionally followed by an asterisk `(*)` (called the **assignment suppression** character), optionally followed by an integer value (called the **field width**). The conversion specification is terminated by a character specifying the type of data to expect; the terminating characters are called **conversion characters**. The integer and floating-point conversion characters may be optionally preceded by a character indicating the size of the receiving variable.

When a conversion specification is encountered in a format, it is matched up with the corresponding item in the item list. The data formatted by that specification is then stored in the location pointed to by that item. For example, if there are four conversion specifications in a format, the first specification is matched up with the first item, the second specification with the second item, and so on.

The number of conversion specifications in the format is directly related to the number of items specified in the item list. With one exception, there must be at least as many items as there are conversion specifications in the format. If there are too few items in the item list, an error occurs; if there are too many items, the excess items are ignored. The one exception occurs when the assignment suppression character (`*`) is used. If an asterisk occurs immediately after the percent sign (before the *`field width`*, if any), the data formatted by that conversion specification is discarded. No corresponding item is expected in the item list; this is useful for skipping over unwanted data in the input.

Conversion Characters

There are 14 conversion characters: five format integer data, three format character data, three format floating-point data, and three special characters.

The integer conversion characters are:

`d`	A decimal integer is expected.
`i`	A signed integer is expected.
`o`	An octal integer is expected.
`u`	An unsigned decimal integer is expected.
`x`	A hexadecimal integer is expected.

The character conversion characters are:

`c`	A single character is expected, normal skip over leading white space is suppressed.
`s`	A character string is expected.
`[`	A character string is expected, normal skip over leading white space is suppressed.

The floating-point conversion characters are:

`e, f, g`	A floating-point number is expected (the capitalized forms of these characters are also accepted).

The special characters are:

`p`	Matches an implementation-defined set of sequences.
`n`	No input is consumed. The corresponding argument is a pointer to an integer into which is written the number of characters read from the input stream so far by this call to `fscanf()`.

`%`	Matches a single `%`. No conversion or assignment occurs. The complete conversion specification is &%&%

Integer Conversion Characters

The `d`, `o`, and `x` conversion characters read characters from the stream until an inappropriate character is encountered, or until the number of characters specified by the *`field width`*, if given, is exhausted (whichever comes first).

For `d`, an inappropriate character is any character except +, -, and 0 through 9. For `o`, an inappropriate character is any character except +, -, and 0 through 7. For `x`, an inappropriate character is any character except +, -, 0 through 9, and the characters `a` through `f` and `A` through `F`. Note that negative octal and hexadecimal values are stored in their twos complement form with sign extension. Thus, they might look unfamiliar if you print them out later using `printf()`.

These integer conversion characters can be preceded by a `l` to indicate that a `long int` should be expected rather than an `int.` They can also be preceded by `h` to indicate a `short int.` The corresponding items in the item list for these conversion characters must be pointers to integer variables of the appropriate length.

Character Conversion Characters

The `c` conversion character reads the next character from the open stream no matter what that character is. The corresponding item in the item list must be a pointer to a character variable. If a *`field width`* is specified, the number of characters indicated by the *`field width`* are read. In this case, the corresponding item must refer to a character array large enough to hold the characters read.

Note that strings read using the `c` conversion character are not automatically terminated with a null character in the array. Because all C library functions that use strings assume the existence of a null terminator, be sure to add the `'\0'` character yourself. If you do not, library functions are not able to tell where the string ends, and you get unexpected results.

The `s` conversion character reads a character string from the open stream, which is delimited by one or more space characters (blanks, tabs, or newlines). If *`field width`* is not given, the input string consists of all characters from the first nonspace character up to (but not including) the first space character. Any initial space characters are skipped over. If a *`field width`* is given, characters are read, beginning with the first nonspace character, up to the first space character, or until the number of characters specified by the *`field width`* is reached (whichever comes first). The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating null character, which is added automatically.

The `s` conversion character cannot be made to read a space character as part of a string. Space characters are always skipped over at the beginning of a string, and they terminate reading whenever they occur in the string. For example, suppose you want to read the first character from the following input line:

```
"          Hello, there!"
```

(Ten spaces followed by "Hello, there!"; the double quotes are added for clarity). If you use `%c`, you get a space character. However, if you use `%1s`, you get "H" (the first nonspace character in the input).

The `[` conversion character also reads a character string from the open stream. However, you should use this character when a string is not to be delimited by space characters. The left bracket is followed by a list of characters, and is terminated by a right bracket. If the first character after the left bracket is a circumflex `(^),` characters are read from the open stream until a character is read that matches one of the characters between the brackets. If the first character is not a circumflex, characters are read from the open stream until a character not occurring between the brackets is found. The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating null character which is added automatically. In some implementations, a minus sign (-) may specify a range of characters.

The three string conversion characters provide you with a complete set of string-reading capabilities. The `c` conversion character can be used to read any single character or to read a character string when the exact number of characters in the string is known beforehand. The `s` conversion character enables you to read any character string that is delimited by space characters and is of unknown length. Finally, the `[` conversion character enables you to read character strings that are delimited by characters other than space characters and that are of unknown length.

Floating-Point Conversion Characters

The `e, f,` and `g` (or `E, F, and G`, respectively) conversion characters read characters from the open stream until an inappropriate character is encountered, or until the number of characters specified by the *`field width`*, if given, is exhausted (whichever comes first).

The `e, f,` and `g` characters expect data in the following form: an optionally signed string of digits (possibly containing a decimal point), followed by an optional exponent field consisting of an `E` or `e` followed by an optionally signed integer. Thus, an inappropriate character is any character except `+, -, .,` `0` through `9, E,` or `e.`

These floating-point conversion characters may be preceded by a lowercase `L` (`l`), to indicate that a `double` value is expected rather than a `float`, or by an uppercase `L` (in ANSI C) to indicate that a long double value is expected rather than a `float`. The corresponding items in the item list for these conversion characters must be pointers to floating-point variables of the appropriate length.

Literal Characters

Any characters included in the format that are not part of a conversion specification are literal characters. A literal character is expected to occur in the input at exactly that point. Note that since the percent sign is used to introduce a conversion specification, you must type two percent signs (%%) to get a literal percent sign.

Examples

Refer to the example located in the `fprintf` function description.

See Also

`getc()`, `setlocale()`, `scanf()`, `fprintf()`, `printf()`, ANSI C 4.9.6.2, POSIX.1 8.1

fseek

Positions the next I/O operation on an open stream to a new position relative to the current position.

Syntax

```
#include <stdio.h>
int fseek (FILE *stream, long int offset, int ptrname);
```

Parameters

stream	A pointer to an open stream.
offset	The number of bytes to skip over. The *offset* parameter can be negative or positive, indicating backward or forward movement in the file, respectively.
ptrname	The reference point in the file from which *offset* bytes are measured.

Return Values

0	Success.
–1	An error occurred, and `errno` is set to indicate the error condition.

Description

The `fseek` function sets the file position indicator for the stream pointed to by *stream*.

For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*. The specified position is:

- The beginning of the file if *whence* is SEEK_SET.
- The current value of the file position indicator if *whence* is SEEK_CUR.
- The end-of-file if *whence* is SEEK_END.

For a text stream, either *offset* is zero, or *offset* is a value returned from a previous call to `ftell()` on the same stream, and *whence* is SEEK_SET.

A successful call to `fseek()` clears the end-of-file indicator for the stream and undoes any effect of `ungetc()` on the same stream. After an `fseek()` call, the next operation on an update stream can be either input or output.

NOTE If linking with the POSIX/iX library, `fseek()` allows the file offset to be set beyond the end of the existing data in the file. If data is written at this point, the gap between the old and new end-of-file is zero filled. However, `fseek()` cannot by itself extend the size of the file.

Example

The following program uses the `ftell()` and `fseek()` functions. The program prints each line of an `n`-line file in this order: line 1, line `n`, line 2, line `n`-1, line 3, and so on.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[ ];
{
   char line[256];
   int newlines;
   long front, rear, ftell();
   FILE *fp;

   front = 0;
   rear = 0;

   if(argc < 2) {
      fprintf(stderr, "Usage:  print filename\n");
      exit(1);
   }

   fp = fopen(argv[1], "r");
   if(fp == NULL) {
      fprintf(stderr, "Can't open %s.\n", argv[1]);
      exit(1);
   }

   newlines = countnl(fp) % 2;

   fseek(fp, 0, 2);
   rear = ftell(fp);

   while(front < rear) {
        fseek(fp, front, 0);
        fgets(line, 256, fp);
        fputs(line, stdout);
        front = ftell(fp);
        findnl(fp, rear);
        rear = ftell(fp);
        if(newlines == 1) {
             if(rear <= front)
                  break;
        }
        fgets(line, 256, fp);
        fputs(line, stdout);
   }

   exit(0);
}

countnl(fp)
FILE *fp;
```

```
{
   char c;
   int count = 0;

   while((c = getc(fp)) != EOF) {

if(c == '\n')

   count++      }
   rewind(fp);
   return(count);
}

findnl(fp, offset)
FILE *fp;
long offset;
{
   char c;

   fseek(fp, (offset-2), 0);
   while((c = getc(fp)) != '\n') {

fseek(fp, -2, 1);
   }
}
```

This program uses the `ftell()` and `fseek()` functions to print lines from a file starting at the beginning and the end of the file, and converging toward the center. The `countnl()` function counts the number of lines in the file so the program can decide whether or not to print a line in the final loop to prevent the middle line being printed twice in files with an odd number of lines. The `findnl()` function searches backwards in the file for the next newline character. When found, this positions the next I/O operation such that `fgets()` gets the next line back from the end of the file.

All three types of seeks are represented in this program. The first `fseek()` of the program is done relative to the end of the file. All other `fseek()` operations in the main program are done relative to the beginning of the file. Finally, `findnl()` contains an `fseek()` that is relative to the current position.

The example located in the `fread()` function description uses a structure that described each employee, as shown below:

```
struct emp {
          char      name[40]; /* name */
          char      job[40];  /* job title */
          long      salary;   /* salary */
          char      hire[6];  /* hire date */
          char      curve[2]; /* pay curve */
          int       rank;     /* percentile ranking */
}
```

This function reads the data for 400 employees all at once. Suppose you want the program to be selective, so that you can specify by employee number (1 through 400) which employee's information you want. The following program fragment shows how to use fseek to do this:

```
          …
       int empno, bytes;
       long total;
       FILE *data;
       struct emp empinfo;

       /* check for usage error and open data file */
          …
       sscanf(argv[1], “%d”, &empno);
       bytes = sizeof(empinfo);
       total = (empno - 1) * bytes;
       fseek(data, total, 0);
       fread((char *)&empinfo, sizeof(empinfo), 1, data);

       /* print out desired information */
          …
       exit(0);
   }
```

In this program, argv[1] contains, using a command-line argument, the employee number about whom information is desired. The employee number is converted to integer form using sscanf. The number of bytes per employee structure is obtained using sizeof, and is stored in bytes. The total number of bytes to skip in the data file is found by multiplying the employee number (minus one) times the number of bytes per employee structure. This is stored in total. Then, fseek() is used to seek past the specified number of bytes, relative to the beginning of the data file. This leaves the next I/O operation positioned at the start of the specified employee's information. The information is read using fread().

See Also

ftell(), rewind(), ANSI C 4.9.9.2, POSIX.1 8.1

fsetpos

Sets the file position for a stream.

Syntax

```
#include <stdio.h>
int fsetpos (FILE *stream, const fpos_t *pos);
```

Parameters

stream	Pointer to a file.
pos	A pointer to a structure that specifies the position of the file position indicator.

Return Values

0	Success.
≠0	An error occurred, and `errno` is set to indicate the error condition.

Description

The `fsetpos` function sets the file position indicator for the stream pointed to by *stream* according to the value of the object pointed to by *pos*. The object pointed to by *pos* must be an object obtained from a previous call to `fgetpos()` on the same stream.

A successful call to `fsetpos()` clears the end-of-file indicator for the stream and undoes any effect of `ungetc()` on the stream. After an `fsetpos()` call, the next operation on an update stream may be either input or output.

See Also

`fgetpos()`, `fseek()`, ANSI C 4.9.9.3

ftell

Returns the current file position indicator for the next I/O operation on an open stream.

Syntax

```
#include <stdio.h>
long int ftell (FILE *stream);
```

Parameters

stream	A pointer to an open stream.

Return Values

≥0	The current position for the next I/O operation, expressed as a byte offset relative to the beginning of the open file. (The first byte is byte 0.)
–1	An error occurred.

Description

The `ftell` function returns the current value of the file position indicator for the stream pointed to by *`stream`*. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, its file position indicator contains unspecified data, usable by `fseek()` for returning the file position indicator for the stream to its position at the time of the call to `ftell()`. The difference between two such return values is not necessarily a meaningful measure of the number of characters being read. The first byte of the file is byte 0.

Examples

Refer to the examples located in the `fseek` function description.

See Also

`rewind()`, `fseek()`, ANSI C 4.9.9.4 , POSIX.1 8.1

fwrite

Writes data items to an open stream.

Syntax

```
#include <stdio.h>
size_t fwrite (const void *ptr, size_t size,
               size_t nitems, FILE *stream);
```

Parameters

ptr	A pointer to a buffer that holds the data to be written to the open stream. The type of the buffer is determined by the type of the data being written.
size	The size of each data item, in bytes.
nitems	The number of data items to write.
stream	A pointer to an open stream.

Return Values

>0	The number of items actually written.

Description

The `fwrite` function is the output analog of the `fread` function. It writes a buffer pointed to by the *ptr* argument to the stream pointed to by the *stream* argument. The number of characters written is equal to the *size* argument times the *nitems* argument.

The file position indicator (if defined) is advanced by the number of characters successfully written.

The `fwrite` function returns the number of elements actually written. This is equal to the number requested unless `fwrite()` encounters an error. In this case, the file position indicator for the stream is indeterminate.

Examples

Refer to the examples located in the `fread` function description.

See Also

`fread()`, ANSI C 4.9.8.2, POSIX.1 8.1

gamma

Returns the log gamma of the input value.

Syntax

```
#include <math.h>
double gamma (double x);
extern int signgam;
```

Parameters

x	A real number.

Return Values

n — A real number giving the natural log of the absolute value of the gamma of *x*.

HUGE — Indicates one of the following:

- The parameter *x* is a non-positive integer, and `errno` is set to `EDOM`. A message indicating `SING` error is printed on the standard error output.
- An overflow condition has occurred, and `errno` is set to `ERANGE`.

Description

`gamma` returns:

`ln( | gamma(x) | )`

where:

`gamma(x)`

is defined as:

Figure 5-2.

$$\int_0^\infty e^{-t}\ t^{x-1}dt$$

The sign of `gamma(x)` is returned in the external integer `signgam`.

The argument *x* must be greater than or equal to zero. (The `gamma` function is defined over the reals excluding the non-positive integers.)

The following C program fragment can be used to calculate `gamma`:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error();
```

```
y = signgam * exp(y);
```

where `LN_MAXDOUBLE` is the lowest value that causes `exp` to return a range error, and is defined in the `<values.h>` header file.

Error handling can be changed by a user-written `matherr` function.

gcvt

Converts floating-point numbers to strings.

Syntax

```
char *gcvt (double value, int ndigit, char *buf);
```

Parameters

value	The floating-point number to be converted to a character string.
ndigit	The number of digits to convert.
buf	A pointer to a character string containing the numeric string to be formatted and to which the resulting formatted character string is returned.

Return Values

x	A pointer to a character array containing the resulting numeric character string (the same as *buf*).

Description

The `gcvt` function converts the floating-point number in *value* into a signed numeric character string. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible; otherwise, E-format is used. A minus sign or decimal point is included as part of the returned string. Trailing zeros are suppressed.

See Also

`ecvt()`, `fcvt()`

getc

Reads a character from an open stream.

Syntax

```
#include <stdio.h>
int getc (FILE *stream);
```

Parameters

stream	A pointer to an open stream.

Return Values

x	The character read, expressed as an integer.
EOF	No more data, or an error occurred.

Description

The `getc` function returns the next character from the input stream pointed to by *stream*. The `getc` function is equivalent to the `fgetc` function except that it is implemented as a macro. Because `getc()` can evaluate the stream more than once, the arguments should never be an expression with side effects.

Examples

A simple version of a command to print a file can be written using `getc()` and `putc()`:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[ ];
{
   int c;
   FILE *fp;

   if(argc != 2) {
      printf("Usage:  cat file\n");
      exit(1);
   }

   fp = fopen(argv[1], "r");
   if(fp == NULL) {
      printf("Can't open %s.\n", argv[1]);
      exit(1);
   }

   while((c = getc(fp)) != EOF)
      putc(c, stdout);
      putc('\n', stdout);

      exit(0);
   }
```

This program accepts a single argument that is assumed to be the name of a file whose contents are to be printed on the terminal. The specified file is opened for reading, and the resulting file pointer `fp` is used in `getc()` to read a character from the file. Each character read is written on `stdout` using `putc()`. (Note that `stdout`, `stdin`, and `stderr` are legal file pointers.) The reading and writing loop terminates when the constant `EOF` is returned from `getc()`, indicating that the end of the file has been reached. This constant is defined in <`stdio.h`>.

You can use the flexibility of `putc()` to send data somewhere other than to the user's terminal. For example, the file copy program from the previous example can be rewritten using `getc()` and `putc()`.

```
   #include <stdio.h>
   main(argc, argv)
   int argc;
   char *argv[ ];
   {
      int c;
      FILE *from, *to;

      if(argc != 3) {
         printf("Usage:  cp fromfile tofile\n");
         exit(1);
      }

      from = fopen(argv[1], "r");
      if(from == NULL) {
         printf("Can't open %s.\n", argv[1]);
         exit(1);
      }
      to = fopen(argv[2], "w");
      if(to == NULL) {
         printf("Can't create %s.\n", argv[2]);
         exit(1);
     }

     while((c = getc(from)) != EOF)
         putc(c, to);

     exit(0);
   }
```

See Also

`fclose()`, `ferror()`, `fopen()`, `fread()`, `fgetc()`, `gets()`, `putc()`, `fputc()`, `scanf()`, ANSI C 4.9.7.5, POSIX.1 8.1

getchar

Reads a character from the standard input stream stdin.

Syntax

```
include <stdio.h>
int getchar (void);
```

Parameters

None.

Return Values

x	The character read from stdin.
EOF	Either an end-of-file was detected or an error occurred.

Description

The getchar function reads one character from the standard input stream stdin.

The getchar function returns the next character in the currently defined stdin stream. It returns an EOF on end-of-file or file read error. The getchar function is identical to getc (stdin).

Examples

The following program reads stdin and echos the contents to stdout. The program ends when an end of file is encountered on stdin.

```
#include <stdio.h>
main()
{
   int c;

   while((c = getchar()) != EOF)
        putchar(c);
   putchar('\n');
}
```

The variable c is declared as an int type instead of char because sign extension, bit shifting, and similar operations can cause unexpected results with the char type. EOF is defined as a negative number. If EOF is assigned to a char variable, and chars are not signed in the implementation, the comparison for EOF will never be true. Therefore, all examples in this chapter use the int type.

The last putchar() statement in the program outputs a new line, so further output appears at the beginning of a new line instead of at the end of the last line of output.

The getchar and putchar functions are most useful in filters (programs that accept and modify data before passing it on). For example, the following program puts parentheses around each vowel encountered in the input:

```
#include <stdio.h>
main()
{
   int c;

   while((c = getchar()) != '\n') {
      if(vowel(c)) {
         putchar('(');
         putchar(c);
         putchar(')');

      }else
          putchar(c);
  }
}
vowel(c)
char c;
{
   switch(c) {
      case 'a':
      case 'A':
      case 'e':
      case 'E':
      case 'i':
      case 'I':
      case 'o':
      case 'O':
      case 'u':
      case 'U':
         return (1);
      default:
         return (0);
  }
}
```

The vowel test is placed in the function `vowel`; the program terminates when it encounters a new line.

The `getc` and `putc` functions can behave exactly like the `getchar` and `putchar` functions by specifying the appropriate file pointer. For example,

```
getc(stdin);
```

is identical to

```
getchar();
```

and

```
putc(c, stdout);
```

is identical to

```
putchar(c);
```

Thus, the `putchar()` call in the previous program can be stated as

```
    putc(c);
```

without altering the behavior of the program.

See Also

`fread()`, `getc()`, `gets()`, `fscanf()`, `scanf()`, `putchar()`, ANSI C 4.9.7.6, POSIX.1 8.1

getenv

Returns the value of an environment variable.

Syntax

```
#include <stdlib.h>
char *getenv (const char *name);
```

Parameters

name	The string to search. The string may be either the desired name, null-terminated, or of the form *name=value*, in which case `getenv()` uses the portion to the left of the = as the search key.

Return Values

x	A pointer to a string associated with the environment variable pointed to by *name*.
`NULL`	The value pointed to by *name* was not found in the environment list.

Description

The `getenv` function searches the environment list for a string that matches the string pointed to by *name*, and returns a pointer to the value in the current environment if such a string is present. If the string is not present, `getenv()` returns a null pointer.

The environment list consists of JCW variables and MPE/iX variables. Refer to the *MPE/iX Commands Reference Manual* for more information on MPE/iX variables.

NOTE If linking with the POSIX/iX library, refer to the description of `getenv()` located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

ANSI C 4.10.4.4

getmsg

Gets a message from a catalog. This function provides support for message catalogs that are created on HP-UX and moved to an MPE/iX system.

Syntax

```
char *getmsg (int fd, int set_num, int msg_num, char *buf,
     int buflen);
```

Parameters

fd	An integer containing a file descriptor of an open message catalog file.
set_num	An integer containing the message set number where the message to be read is located.
msg_num	An integer containing the message number within the set to read from the message catalog.
buf	A pointer to a character array in which the message is returned.
buflen	An integer containing the length of buffer pointed to by *buf*.

Return Values

x	A pointer to the returned string. This is the same value as *buf*.
NULL	Indicates failure. The file descriptor may be invalid, or the message indicated by *set_num* and *msg_num* may not be in the catalog.

Description

The getmsg function gets messages from an HP-UX message catalog. It provides interoperability support for message catalogs ported to MPE/iX from HP-UX systems. For information on how to read message catalogs created on MPE/iX with the GENCAT utility, refer to the descriptions of the MPE/iX intrinsics CATOPEN, CATCLOSE, and CATREAD which are documented in the *MPE/iX Intrinsics Reference Manual*.

The getmsg function attempts to read up to *buflen*-1 bytes of the specified message in the message catalog into the area pointed to by *buf*. A null byte is inserted to terminate the string placed in the buffer.

A **message catalog** is a specially formatted file containing numbered messages that are grouped together in message sets. The file contains an index allowing fast access to the messages. The calling program must open the message catalog before calling getmsg.

See Also

catread()

getopt

Gets ASCII characters from an argument vector.

Syntax

```
int getopt (int argc, char *argv, char *optstring);
extern char *optarg;
extern int optind, opterr;
```

Parameters

argc	An integer giving the length of the array *argv*.
argv	A pointer to the command line.
optstring	A string of recognized option letters.

Return Values

`'?'`	An option letter is not included in *optstring*. This error message can be disabled by setting `opterr` to zero.
`EOF`	All options have been processed.
n	The next option letter in *argv*, starting from `argv[1]` that matches a letter in *optstring*.
optarg	An external character pointer that is set to the start of the option argument, if any, on return from `getopt`.
optind	An external integer that should be initialized to one before the first call to `getopt`. The *optind* value is set to the *argv* index of the next argument to be processed on return from `getopt`.

Description

The `getopt` function returns the next option letter in *argv* that matches a letter in *optstring*. The *optstring* argument is a string of recognized option letters. If a letter in *optstring* is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space.

On return from `getopt`, *optarg* points to the start of the option argument.

The `getopt` function places in *optind* the *argv* index of the next argument to be processed. The external variable *optind* is initialized to 1 before the first call to `getopt()`.

The external integer `opterr` enables or disables printing error messages on the standard error device.

When all options have been processed (for example, up to the first non-option argument), `getopt` returns `EOF`. The special option can be used to delimit the end of the options. When this option is processed, `EOF` is returned and *optind* is incremented to the *argv* index beyond .

Options can be any ASCII characters except colon (`:`), question mark (`?`), or null (`\0`). It is impossible to distinguish between a `?` used as a legal option, and the character that `getopt` returns when it encounters an invalid option character in the input.

Set `opterr` to 0 to disable `getopt` from printing error messages on the standard error device. Otherwise, `getopt` prints an error message on the `stderr` and returns a question mark (`?`) when it encounters an option letter not included in *`optstring`*.

Example

The following code fragment shows how you can process the arguments for a command that can take the mutually exclusive options `a` and `b`, and the options `f` and `o`, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
   int c;
   extern char *optarg;
   extern int optind;
      …
   while ((c = getopt(argc, argv, "abf:o:")) != EOF)
   switch (c) {
      case 'a':
         if (bflg)
            errflg++;
         else
            aflg++;
         break;
      case 'b':
         if (aflg)
            errflg++;
         else
            bproc( );
         break;
      case 'f':
         ifile = optarg;
         break;
      case 'o':
         ofile = optarg;
         break;
      case '?':
         errflg++;
   }
   if (errflg) {
      (void) fprintf(stderr, "usage: .  .  .  ");
      exit (2);
   }
   for ( ; optind argc; optind++) {
      if (access(argv[optind], 4)) {
          …
}
```

gets

Reads a string from the standard input stream `stdin`.

Syntax

```
#include <stdio.h>
char *gets (char *s);
```

Parameters

s	A pointer to a character array where the string is to be returned.

Return Values

x	A pointer to the character array.
`NULL`	An error occurred. If any characters were read, the array contents are indeterminate.

Description

The `gets` function reads a string from the standard input stream, `stdin`, and stores the string in a character array pointed to by *s*. The string terminates by a new line in the input, which `gets()` replaces with a null character in the array or when end-of-file is reached.

Example

The following example uses `gets()` and `puts()`.

```
#include <stdio.h>
main()
{
   char line[80], *gets();

   while((gets(line)) != NULL)
      puts(line);
}
```

To terminate this program, generate an end of file on `stdin.` Using string comparison and string length functions, you can write a termination condition for this program.

See Also

`ferror()`, `fopen()`, `fread()`, `getc()`, `puts()`, `scanf()`, ANSI C 4.9.7.7, POSIX.1 8.1

getpid

Returns the process identification number.

NOTE	If linking with the POSIX/iX libraries, refer to the description of `getpid()` located in the *MPE/iX Developer's Kit Reference Manual*.

Syntax

```
int getpid (void)
```

Parameters

None.

Return Values

x	The process identification number (PIN) of the calling process.

See Also

MPE/iX intrinsics `FATHER` and `GETPROCID`, described in the *MPE/iX Intrinsics Reference Manual*.

getw

Reads a word from an open stream.

Syntax

```
#include <stdio.h>
int getw (FILE *stream);
```

Parameters

stream	A pointer to an open stream.

Return Values

x	The word read, expressed as an integer.
EOF	No more data, or an error occurred.

Description

The `getw` function returns the next word (`int` in C) from the named input *stream*. The `getw` function increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. The `getw` function assumes no special alignment in the file.

See Also

`fgetc()`, `ferror()`, `getc()`, `getchar()`

gmtime

Converts time to Coordinated Universal Time (UTC) in the structured `tm` type format.

Syntax

```
#include <time.h>
struct tm *gmtime (const time_t *timer);
```

Parameters

timer	A pointer to a variable of type `time_t`.

Return Values

x	A pointer to a variable of type `tm`.
`NULL`	The Coordinated Universal Time (UTC) is not available.

Description

The `gmtime` function converts a `time_t` variable, such as that returned by the `time` function, into a structured `tm` format. The converted value is expressed in Coordinated Universal Time (UTC).

See Also

`localtime()`, ANSI C 4.12.3.3, POSIX.1 8.1

hcreate

Allocates sufficient space for a hash table used by the `hsearch` function.

Syntax

```
#include <search.h>
int hcreate (unsigned nel);
```

Parameters

nel	An estimate of the maximum number of elements that the table contains. This number may be adjusted upward by the algorithm to obtain a mathematically favorable table size.

Return Values

≠0	Successful. The space was allocated.
0	The space sufficient to contain the number of entries specified in *nel* was not available. Therefore, no space was allocated.

Description

The `hcreate` function allocates space sufficient for a hash table that is to be searched by the `hsearch` function. The size of the space is determined by the *nel* parameter. The hash table itself is an array of pointers. The size of the data elements to be searched is not relevant to determining the amount of memory to be allocated for the hash table.

Only one hash search table may be active at any given time.

The `hcreate` function must be called before `hsearch()` to allocate sufficient space for the hash table.

NOTE The `hcreate` function and the header file `<search.h>` are not part of ANSI C. Using them may make your program less portable.

The `hdestroy` function may be used to deallocate the hash table when it is no longer needed.

Examples

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

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

struct info { /* this is the info stored in the table */
```

```
   int age, room; /* other than the key.  */
};

#define NUM_EMPL 5000 /* number of elements in search table */
#define END_FLAG -1 /* sentinel value for age to terminate
                                           table input */

main( )
{
   /* space to store strings */
   char string_space[NUM_EMPL*20];

   /* space to store employee information */
   struct info info_space[NUM_EMPL];

   /* next available space in string_space */
   char *str_ptr = string_space;

   /* next available space in info_space */
   struct info *info_ptr = info_space;
   ENTRY item, *found_item, *hsearch( );

   /* name to look for in table */
   char name_to_find[30];
   int i = 1;

   /* create table */
   (void) printf("Enter name, age, and room for table.  ");
   (void) printf("To terminate input, enter -1 for age.\n");
   (void) hcreate(NUM_EMPL);
   do {
       if (scanf("%s%d%d", str_ptr,info_ptr->age,
            info_ptr->room) == EOF) exit(0);
       if (info_ptr->age == END_FLAG) break;

      /* put information into structure */
      item.key = str_ptr;
      item.data = (char *)info_ptr;
      str_ptr += strlen(str_ptr) + 1;
      info_ptr++;

      /* put item into table */
      (void) hsearch(item, ENTER);
   } while (i++ NUM_EMPL);
   /* access table */
   item.key = name_to_find;
   while (scanf("%s", item.key) != EOF) {
      if ((found_item = hsearch(item, FIND)) != NULL) {
         /* if item is in the table */
            found_item->key,
            ((struct info *)found_item->data)->age,
            ((struct info *)found_item->data)->room);
      } else {
          (void) printf("No such employee %\n", name_to_find);
```

```
        }
    }
}
```

See Also

`hsearch()`, `hdestroy()`

hdestroy

Destroys a search table created by `hcreate()`.

Syntax

```
#include <search.h>
void hdestroy (void);
```

Parameters

None.

Return Values

None.

Description

The `hdestroy` function destroys the search table created by a previous call to `hcreate()`. A subsequent call to `hcreate()` can be made to create a new search table.

Only one hash search table may be active at any given time.

NOTE The `hdestroy` function and the header file `<search.h>` are not part of ANSI C. Using them may make your program less portable.

Examples

Refer to the example located in the `hcreate` function description.

See Also

`hsearch()`, `hcreate()`

hsearch

Returns a pointer into a hash table, indicating the location of a specified entry.

Syntax

```
#include <search.h>
ENTRY *hsearch (ENTRY item, ACTION action);
```

Parameters

item	A structure of type `ENTRY`, defined in the `<search.h>` header file. The *item* parameter contains two pointers: • *item.*`key` points to the comparison key. • *item.*`data` points to any other data to be associated with that key. Pointers to types other than character should be cast to pointer-to-character.
action	A member of an enumeration type `ACTION` which indicates the disposition of the entry if it cannot be found in the table. `ENTER` indicates that the item should be inserted in the table at an appropriate point. `FIND` indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a null pointer.

Return Values

x	A pointer to an object of type `ENTRY`, giving the location of the item in the table.
`NULL`	The table is full, or the item was not found.

Description

The `hsearch` function returns a pointer into a hash table, indicating the location at which a specified entry can be found. This function uses `malloc()` to allocate space.

Only one hash search table may be active at any given time. The `hcreate` function must be called before `hsearch()` to allocate sufficient space for the hash table.

The `hsearch` function is a hash-table search function generalized from Knuth Algorithm D (6.4) described in *The Art of Computer Programming, Vol.3 (Sorting and Searching)* by Donald Ervin Knuth (Reading, Mass.:Addison- Wesley, 1973).

NOTE The `hsearch` function and the header file `<search.h>` are not part of ANSI C. Using them may make your program less portable.

Example

Refer to the example located in the `hcreate` function description.

See Also

`hcreate()`, `hdestroy()`

hypot

Computes the length of the hypotenuse of a right triangle.

Syntax

```
#include <math.h>
double hypot (double x, double y);
```

Parameters

x	A real number indicating the length of one of the sides of the triangle adjacent to the right angle.
y	A real number indicating the length of the other side of the triangle adjacent to the right angle.

Return Values

`n`	The length of the hypotenuse of a right triangle.
`HUGE`	An overflow condition has occurred; `errno` is set to `ERANGE`.

Description

The `hypot` function returns `sqrt` (*x* * *x* + *y* * *y*), taking precautions to avoid overflow.

Error handling can be changed by a user-written `matherr` function.

See Also

`matherr()`

isalnum

Tests whether an argument is a letter or a decimal digit.

Syntax

```
#include <ctype.h>
int isalnum (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an unsigned char or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is a letter or a decimal digit.
0	The argument passed in *c* is not a letter or a decimal digit.

Description

The `isalnum` function returns a nonzero value if the argument passed in *c* is a letter or a decimal digit (ASCII characters `0` through `9`, `A` through `Z`, and `a` through `z`); otherwise, zero is returned.

See Also

`isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, ANSI C 4.3.1.1, POSIX.1 8.1

isalpha

Tests whether an argument is a letter.

Syntax

```
#include <ctype.h>
int isalpha (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an unsigned char or the macro EOF.

Return Values

≠0	The argument passed in *c* is a letter.
0	The argument passed in *c* is not a letter.

Description

The isalpha macro returns a nonzero value if the argument passed in *c* is a letter (ASCII characters A through Z, and a through z); otherwise, the returned value is zero.

See Also

isalnum(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), ANSI C 4.3.1.2, POSIX.1 8.1

isatty

Checks whether a file descriptor is associated with a display device, such as a terminal.

Syntax

`int isatty (int` *`fildes`*`)`

Parameters

`fildes`	An open file descriptor.

Return Values

1	The file descriptor is a terminal device.
0	The file descriptor is not a terminal device.

Description

The `isatty` function returns true or false depending on whether or not *`fildes`* is associated with a terminal.

NOTE If linking with the POSIX/iX library, refer to the description of `isatty()` located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

dup(), open()

iscntrl

Tests whether an argument is a control character.

Syntax

```
#include <ctype.h>
int iscntrl (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is an ASCII control character.
0	The argument passed in *c* is not an ASCII control character.

Description

The `iscntrl` function returns a nonzero value if the argument passed in *c* is an ASCII control character (octal values 00 through 037 and 0177); otherwise, the returned value is zero.

See Also

`isalnum()`, `isalpha()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, ANSI C 4.3.1.3, POSIX.1 8.1

isdigit

Tests whether an argument is a decimal digit.

Syntax

```
#include <ctype.h>
int isdigit (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is a decimal digit.
0	The argument passed in *c* is not a decimal digit.

Description

The `isdigit` function returns a nonzero value if the argument passed in *c* is a decimal digit (ASCII characters `0` through `9`); otherwise, the returned value is zero.

See Also

`isalnum()`, `isalpha()`, `iscntrl()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, ANSI C 4.3.1.4, POSIX.1 8.1

isgraph

Tests whether an argument is a printable nonspace character.

Syntax

```
#include <ctype.h>
int isgraph (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is printable.
0	The argument passed in *c* is not printable.

Description

The `isgraph` function returns a nonzero value if the argument passed in *c* is a printable non-space character; otherwise, the returned value is zero.

See Also

`isalnum`, `isalpha`, `iscntrl`, `isdigit`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`, ANSI C 4.3.1.5, POSIX.1 8.1

islower

Tests whether an argument is a lowercase letter.

Syntax

```
#include <ctype.h>
int islower (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is a lowercase letter.
0	The argument passed in *c* is not a lowercase letter.

Description

The `islower` function returns a nonzero value if the argument passed in *c* is a lowercase letter (ASCII characters `a` through `z`); otherwise, the returned value is zero.

See Also

`isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`, ANSI C 4.3.1.6, POSIX.1 8.1

isprint

Tests whether an argument is any printable character including the space character (octal values 040 through 0176).

Syntax

```
#include <ctype.h>
int isprint (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is any printable character including the space character (octal values 040 through 0176).
0	The argument passed in *c* is not any printable character.

Description

The `isprint` function returns a nonzero value if the argument passed in *c* is any printable character including the space character (octal values 040 through 0176); otherwise, the returned value is zero.

See Also

`isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `ispunct`, `isspace`, `isupper`, `isxdigit`, ANSI C 4.3.1.7, POSIX.1 8.1

ispunct

Tests whether an argument is any printable character that is not a space, a digit, or a letter.

Syntax

```
#include <ctype.h>
int ispunct (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is any printable character that is not a space, a digit, or a letter.
0	The argument passed in *c* is not any printable character that is not a space, a digit, or a letter.

Description

The `ispunct` function returns a nonzero value if the argument passed in *c* is any printable character that is not a space, a digit, or a letter; otherwise, the returned value is zero.

See Also

`isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `isspace`, `isupper`, `isxdigit`, ANSI C 4.3.1.8, POSIX.1 8.1

isspace

Tests whether an argument is a white-space character.

Syntax

```
#include <ctype.h>
int isspace (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is a white-space character.
0	The argument passed in *c* is not a white-space character.

Description

The `isspace` function returns a nonzero value if the argument passed in *c* is a white-space character (white-space characters are space, form feed, newline, carriage return, horizontal tab, and vertical tab); otherwise, the returned value is zero.

See Also

`isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isupper`, `isxdigit`. ANSI C 4.3.1.9, POSIX.1 8.1

isupper

Tests whether an argument is an uppercase letter.

Syntax

```
#include <ctype.h>
int isupper (int c);
```

Parameters

c	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *c* is an uppercase letter.
0	The argument passed in *c* is not an uppercase letter.

Description

The `isupper` function returns a nonzero value if the argument passed in *c* is an uppercase letter (ASCII characters `A` through `Z`); otherwise, the returned value is zero.

See Also

`isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isxdigit`. ANSI C 4.3.1.10, POSIX.1 8.1

isxdigit

Tests whether an argument is a hexadecimal digit.

Syntax

```
#include <ctype.h>
int isxdigit (int c);
```

Parameters

`c`	An argument to be evaluated. The argument must be representable as an `unsigned char` or the macro `EOF`.

Return Values

≠0	The argument passed in *`c`* is a hexadecimal digit.
0	The argument passed in *`c`* is not a hexadecimal digit.

Description

The `isxdigit` function returns a nonzero value if the argument passed in *`c`* is a hexadecimal digit (ASCII characters `0` through `9`, `A` through `F`, and `a` through `f`); otherwise, the returned value is zero.

See Also

`isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, ANSI C 4.3.1.11, POSIX.1 8.1

l3tol

The `l3tol` function converts 3-byte integers to long integers.

Syntax

```
void l3tol (long *lp, char *cp, int *n);
```

Parameters

lp	A pointer to an array of *n* converted long integers.
cp	A pointer to a character string containing the *n* three-byte integers to be converted.
n	The number of three-byte integers packed in *cp*.

Return Values

None.

Description

The `l3tol` function converts a list of *n* 3-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

This function supports file systems where block numbers are 3 bytes long.

See Also

`ltol3()`

l64a

Converts a long integer to a base-64 ASCII string.

Syntax

```
char *l64a (l)
   long  l;
```

Parameters

l	A long integer.

Return Values

x	A pointer to the base-64 ASCII string.
`NULL`	The argument is 0.

Description

This function maintains numbers stored in base-64 ASCII characters. Long integers can be represented by up to six characters. Each character represents a *digit* in a radix-64 notation.

The characters used to represent digits are:

Characters	Digits
`.`	0
`/`	1
`0` through `9`	2 through 11
`A` through `Z`	12 through 37
`a` through `z`	38 through 63

The leftmost character is the least significant digit. For example:

$$a0 = (38 \times 64^{0}) + (2 \times 64^{1}) = 166$$

This function takes a long argument and returns a pointer to the corresponding base-64 representation. If the argument is zero, `l64a` returns a pointer to a null string.

The value returned by `l64a` is a pointer into a static buffer, the contents of which are overwritten by each call.

See Also

`a64l()`

labs

Computes the absolute value of a long integer argument.

Syntax

```
#include <stdlib.h>
long int labs (long int j);
```

Parameters

j	A long integer value whose absolute value is to be computed.

Return Values

x	The absolute value of the long integer specified in *j*.

Description

The `labs` function returns the absolute value of the long integer value specified in *j*.

See Also

`abs()`, `fabs()`, ANSI C 4.10.6.3

ldexp

Accepts a `double` *`value`* and an integer exponent *`exp`*, and returns a `double` quantity equal to *`value`* `*` 2^{exp}.

Syntax

```
#include <math.h>
double ldexp (double value, int exp);
```

Parameters

`value`	A real number that is to be multiplied by 2^{exp}.
`exp`	The integer exponent value to which 2 is raised.

Return Values

n	The result of *`value`* `*` 2^{exp}.
0	An underflow condition has occurred; `errno` is set to indicate the error.
`HUGE_VAL`	An overflow condition has occurred.

Description

The `ldexp` function multiplies the floating-point argument *`value`* by an integral power of 2^{exp}.

Example

The following program accepts two number arguments, *`value`* and *`exp`*, and outputs the result:

```
main(argc, argv)
int argc;
char *argv[ ];
{
   double value, result, ldexp();
   int exp;



   result = ldexp(value, exp);
   printf("%g * 2^%d = %g\n", value, exp, result);
}
```

See Also

ANSI C 4.5.4.3, POSIX.1 8.1

ldiv

Computes the quotient and remainder of two long integers.

Syntax

```
#include <stdlib.h>
ldiv_t ldiv (long int numer, long int denom);
```

Parameters

numer	The numerator.
denom	The denominator.

Return Values

Returns a structure of type `ldiv_t` comprising the quotient and the remainder. The structure contains the following:

```
long int quot;    /* quotient */
long int rem;     /* remainder */
```

Description

The `ldiv` function computes and returns the quotient and the remainder of the division of *numer* by *denom*.

If the division is inexact, the sign of the resulting quotient and the algebraic quotient are the same, and the magnitude of the resulting quotient is the largest long int less than the magnitude of the algebraic quotient.

If the result cannot be represented, the behavior is undefined; otherwise, *quot* * *denom* + *rem* equals *numer*.

See Also

`div()`, `abs()`, `labs()`, ANSI C 4.10.6.4

lfind

Performs a linear search.

Syntax

```
#include <stdio.h>
char *lfind ((char *)key, (char *)base, nelp, sizeof(*key),
compar)
unsigned *nelp
int (*compar) ( );
```

Parameters

key	A pointer to the value to be found in the table.
base	A pointer to the first element in the table.
nelp	A pointer to an integer containing the current number of elements in the table.
width	The size of each datum in the table; it is the width of each table row.
compar	A pointer to a comparison function that you must supply, such as `strcmp`. It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal, and non-zero if they are not equal.

Return Values

x	A character pointer to the table entry being sought.
`NULL`	The searched item is not found.

Description

This function is a linear search function generalized from Knuth Algorithm S (6.1). [1] It returns a pointer into a table indicating where an item may be found.

This function is the same as `lsearch` except that if the item is not found, it is not added to the table. Instead, a null pointer is returned.

The pointers to the key and the element at the base of the table should be of type pointer-to-element and should be cast to type pointer-to-character.

The comparison function does not need to compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The value returned is declared as type pointer-to-character, but should be cast as type pointer-to-element.

1. *The Art of Computer Programming, Vol.3 (Sorting and Searching)* by Donald Ervin Knuth (Reading, Mass:Addison-Wesley, 1973).

See Also

`lsearch()`

localeconv

Returns information about the editing symbols of a numeric quantity specific to a locale.

Syntax

```
#include <locale.h>
struct lconv *localeconv (void);
```

Parameters

None.

Return Values

x	A pointer to an object of type `struct lconv`.

Description

The `localeconv` function returns a pointer to an object of type `struct lconv` that contains information about the editing symbols of a numeric quantity specific to a locale. The structure returned by `localeconv()` must not be altered and may be overwritten by subsequent calls to `localeconv()`.

The struct type `lconv`, defined in the header `<locale.h>`, contains the following members:

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

See Also

`setlocale()`, ANSI C 4.4.2.1

localtime

Converts time to the local time zone.

Syntax

```
#include <time.h>
struct tm *localtime (const time_t *timer);
```

Parameters

timer	A pointer to a variable of type `time_t`.

Return Values

x	A pointer to a structured time variable of type `tm`.

Description

The `localtime` function is passed a pointer to a `time_t` variable whose value is typically set by the `time` function. The `localtime` function converts this value into the structured `tm` format expressed in local time, corrected for daylight saving time if applicable, and returns a pointer to the structure.

By default, `localtime()` adjusts the return value to Eastern Standard Time (EST). You may control this by using the MPE/iX command `SETVAR TZ` *name*. Time zone names, and the format of `TZTAB.LIB.SYS` file containing offsets from UTC (Coordinated Universal Time) are listed in appendix A, "Time Zones."

Example

The following code fragment assigns values to the `tm` structure members for the local time zone:

```
#include <time.h>
    …
struct tm *ptr, *localtime();
int time(), nseconds;
    …
nseconds = time(NULL);
```

Once this code is executed, you can use `ptr` to access the different components of the local time. For example, `ptr -> tm_mon` references the month of the year, and `ptr -> tm_wday` references the day of the week.

See Also

`time()`, `ctime()`, ANSI C 4.12.3.4, POSIX.1 8.1

log, log10

Syntax

```
#include <math.h>
double log (double x);
double log10 (double y);
```

Parameters

x	A real number whose natural logarithm is to be returned.
y	A real number whose logarithm in base 10 is to be returned.

Return Values

n	The logarithm of the input value *x*.
`-HUGE`	The input value *x* is ≤0, and `errno` is set to `EDOM`. A `DOMAIN` error or `SING` error if *x*=0 is printed on the standard error output device.

Description

The `log` function returns the natural logarithm of *x*. The value of *x* must be positive.

The `log10` function returns the logarithm base ten of *x*. The value of *x* must be positive.

Error handling can be changed by a user-written `matherr` function.

See Also

`matherr()`, ANSI C 4.5.4.4, POSIX.1 8.1

longjmp

Restores an environment previously saved by `setjmp()`.

Syntax

```
#include <setjmp.h>
void longjmp (jmp_buf env, int val);
```

Parameters

env	Passes information needed to restore a previous environment. This variable was used in a previous call to `setjmp()` to save the environment. The type `jmp_buf` (defined in `<setjmp.h>`) defines an array of unsigned integers. For this reason, the *env* argument does not require an & operator.
val	Passes a value to be returned by `setjmp()`. If a zero is passed in this argument, it is changed to a value of 1 to ensure that `longjmp()` never causes `setjmp()` to return a zero value.

Return Values

None. Control is returned to the program at the statement following the call to `setjmp`.

Description

The `longjmp` function restores an environment saved in the *env* argument by a previous call to `setjmp()`. If the *env* argument is not the result of a successful call to `setjmp()`, the operation of `longjmp()` is undefined and usually results in the program aborting.

After the call to `longjmp()` completes, the program executes as if the call to `setjmp()` (which stored information into the *env* argument) has returned a second time. The result of the second return from `setjmp()` is the return of the value of the nonzero *val* argument supplied to `longjmp()`.

The calling environment defined in *env* is restored by `longjmp()`. This includes trimming the stack so that all stack frames beyond the frame marked by *env* are removed.

The `longjmp` function cannot add stack frames. This means that if a sequence of functions is:

```
A == calls ==> B == calls ==> C
```

and `setjmp()` is used in function C to save an environment in a global *env*, functions B or A may not contain any `longjmp()` calls that reference the *env* values. Only subordinate functions may issue calls to `longjmp()`. As a special case, a function may issue a `longjmp()` call that references a `setjmp()` within itself, although this is not usually done.

The `longjmp` function works correctly in the context of signals and interrupts and any of their associated functions. However, the effects of invoking `longjmp()` from a nested signal handler (that is, a function invoked as a result of a signal raised while handling another signal) are undefined.

Control does not return directly from a call to `longjmp()`, so there are no return values. Instead, control is returned to `setjmp()`, and the value stored in *`val`* is used as the return value of `setjmp()`.

NOTE This function is also implemented as the macro `_longjmp`.

See Also

`setjmp()`, ANSI C 4.6.2.1, POSIX.1 8.1

lsearch

Performs a linear search and update.

Syntax

```
#include <stdio.h>
#include <search.h>
char *lsearch ((char *)key, (char *)base, nelp,
sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

Parameters

key	A pointer to the value to be found in the table.
base	A pointer to the first element in the table.
nelp	A pointer to an integer containing the current number of elements in the table. This integer is incremented if the item is added to the table.
width	The width of each table row.
compar	A pointer to a comparison function that you must supply, such as `strcmp`. It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal, and nonzero if they are not equal.

Return Values

x	A character pointer to the element being sought, whether newly added or pre-existing in the table.

Description

The `lsearch` function is a linear search function generalized from Knuth Algoritm S (6.1). 1

It returns a pointer into a table indicating where an item may be found. If the item does not occur, it is added at the end of the table.

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function does not need to compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The value returned is declared as type pointer-to-character, but should be cast as type

1. *The Art of Computer Programming, Vol.3 (Sorting and Searching)* by Donald Ervin Knuth (Reading, Mass:Addison-Wesley, 1973).

pointer-to-element.

Undefined results can occur if there is not enough room in the table to add a new item.

Example

This fragment reads in TABSIZE strings of length ELSIZE and stores them in a table, eliminating duplicates.

```
#include stdio.h>
#include search.h>
#define TABSIZE 50
#define ELSIZE 120

   char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
   unsigned nel = 0;
   int strcmp( );
      …
   while (fgets(line, ELSIZE, stdin) != NULL
      nel TABSIZE)
      (void) lsearch(line, (char *)tab,nel,
         ELSIZE, strcmp);
             …
```

See Also

lfind()

lseek

Moves the file position indicator.

Syntax

```
#include <unistd.h>
long lseek (int fildes, long offset, int whence);
```

Parameters

filedes	An open file descriptor.
offset	The number of bytes to move the current file position indicator, according to the method defined by *whence*.
whence	The starting point for the seek operation. The possible values are:

SEEK_SET	Seek relative to the beginning of file.
SEEK_CUR	Seek relative to the current location in the file.
SEEK_END	Seek relative the end of file.

Return Values

≥0	The resulting file position indicator location, as measured in bytes from the beginning of the file.
–1	An error occurred and `errno` is set to one of the following values:

EBADF	The *fildes* parameter is not an open file descriptor.
EINVAL	The *whence* parameter is not 0, 1 or 2.
EINVAL	The resulting file position indicator would be negative.
ESEEK	The *fildes* parameter does not refer to a file that supports seeking.
ESYSERR	A call to a system intrinsic failed.

Description

The `lseek` function repositions the file position indicator associated with *fildes*.

Some devices are incapable of seeking. The value of the file position indicator associated with such a device is undefined.

NOTE If linking with the POSIX/iX library, refer to the description of `lseek()` located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

open( ), dup( )

ltol3

Converts long integers to 3-byte integers.

Syntax

```
void ltol3 (char *cp, long *lp, int n)
   char *cp;
   long *lp;
   int n;
```

Parameters

cp	A pointer to a character string to which *n* 3-byte integers are returned.
lp	A pointer to an array of *n* long integers.
n	The number of long integers to be converted.

Return Values

None.

Description

The `ltol3` function converts long integers (*lp*) to 3-byte integers (*cp*).

This function supports file systems where block numbers are 3-byte integers.

See Also

`l3tol()`

mallinfo

Returns information describing space usage.

Syntax

```
#include <malloc.h>
struct mallinfo mallinfo (void);
```

Parameters

None.

Return Values

x	A pointer to an object of type `struct mallinfo`.

Description

The `mallinfo` function provides instrumentation describing space usage, but may not be called until the first small block is allocated.

It returns the following structure:

```
struct mallinfo {
   int arena;    /* total space in arena */
   int ordblks;  /* number of ordinary blocks */
   int smblks;   /* number of small blocks */
   int hblkhd;   /* space in holding block headers */
   int hblks;    /* number of holding blocks */
   int usmblks;  /* space in small blocks in use */
   int fsmblks;  /* space in free small blocks */
   int uordblks; /* space in ordinary blocks in use */
   int fordblks; /* space in free ordinary blocks */
   int keepcost; /* space penalty if keep option */
                 /* is used */
}
```

This structure is defined in the `<malloc.h>` header file.

NOTE The header `<malloc.h>` and the `mallopt()` and `mallinfo()` functions are not ANSI C and should be avoided if portability is a consideration.

See Also

`malloc()`, `calloc()`, `mallopt()`, `realloc()`

malloc

Allocates a block of memory.

Syntax

```
#include <stdlib.h>
void *malloc (size_t size);
```

Parameters

size	The number of bytes in the block to be allocated.

Return Values

x	A pointer to an allocated block of memory.
`NULL`	There is not enough memory available, or *size* is 0.

Description

The `malloc` function returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The `malloc` and `free` functions provide a simple generalized memory allocation package.

Undefined results occur if the space assigned by `malloc()` is overrun.

See Also

`free()`, `realloc()`, `calloc()`, ANSI C 4.10.3.3, POSIX.1 8.1

mallopt

Provides control over the memory allocation algorithm.

Syntax

```
#include <malloc.h>
int mallopt (int cmd, int value);
```

Parameters

cmd The available values for *cmd* are:

M_MXFAST	Set `maxfast` to *value*. The algorithm allocates all blocks below the size of `maxfast` in large groups and then passes them out very quickly. The default value for `maxfast` is 24.
M_NLBLKS	Set `numlblks` to *value*. The above mentioned large groups each contain `numlblks` blocks. `Numlblks` must be greater than 1. The default value for `numlblks` is 100.
M_GRAIN	Set `grain` to *value*. The sizes of all blocks smaller than `maxfast` are considered to be rounded up to the nearest multiple of `grain`. `grain` must be greater than zero. The default value of `grain` is the smallest number of bytes that allows alignment of any data type. The *value* parameter is rounded up to a multiple of the default when `grain` is set.
M_KEEP	Preserve data in a freed block until the next `malloc`, `realloc`, or `calloc`. This option is provided only for compatibility with other systems and is not recommended.

value An integer value used by *cmd*.

Return Values

0	Success.
1	Indicates `malloc()` has been previously called or that arguments have illegal values.

Description

The `mallopt` function returns a pointer to space suitably aligned, after possible pointer coercion, for storage of any type of object. It also provides control over the main memory allocation algorithm. The `mallopt` function may be called repeatedly, but may not be called after the first small block is allocated.

The contents of a block are not preserved when it is freed, unless the M_KEEP option of `mallopt()` is specified in *cmd*.

NOTE The header `<malloc.h>` and the `mallopt()` and `mallinfo()` functions are not ANSI C and should be avoided if portability is a consideration.

See Also

`free()`, `realloc()`, `calloc()`, `mallinfo()`, `mallopt()`, `malloc()`

matherr

The `matherr` function is a user-written call-back routine invoked by many functions in the math library when errors are detected.

Syntax

```
#include <math.h>
int matherr (x)
   struct exception *x;
```

Parameters

x	A pointer to the `exception` structure defined in the `<math.h>` header file.

Return Values

x	A user-defined integer value.

Description

Users override the default math library error handler by defining a function named `matherr` in their programs. This user-written `matherr` function must follow the syntax described above.

When an error occurs, a pointer to the `exception` structure *x* is passed to your `matherr` function.

The structure is defined as follows:

```
struct exception {
     int type;
     char *name;
     double arg1, arg2, retval;
};
```

The element `type` is an integer describing the type of error that occurred, from the following list of constants defined in the header file:

`DOMAIN`	argument domain error
`SING`	argument singularity
`OVERFLOW`	overflow range error
`UNDERFLOW`	underflow range error
`TLOSS`	total loss of significance
`PLOSS`	partial loss of significance

The element `name` points to a string containing the name of the function that caused the error. The variables `arg1` and `arg2` are the arguments you use to invoke the function. `retval` is set to the default value that is returned by the function unless your `matherr` sets it to a different value.

Consult the function descriptions in this chapter to determine if a specific function calls `matherr`.

If your `matherr` function returns nonzero, no error message is printed, and `errno` is not set.

If `matherr` is not supplied, the default error-handling procedures, described with the math functions involved, are invoked upon error. In every case, `errno` is set to EDOM or ERANGE, and the program continues.

Example

```
#include <math.h>

int matherr(struct exception *x)
{
  switch (x->type) {
  case DOMAIN:
       /* change sqrt to return sqrt(-arg1), not 0 */
       if (!strcmp(x->name, "sqrt")) {
            x->retval = sqrt(-x->arg1);
            return (0); /* print message and set errno */
       }

  case SING:
       /* all other domain or sing errors, */
       /* print message and abort */
       (void) fprintf(stderr, "domain error in %s\n", x->name);
       abort( );
  case PLOSS:
       /* print detailed error message */
       (void) fprintf(stderr, "loss of significance in %s(%g)=%g\n",
            x->name, x->arg1, x->retval);
       return (1); /* take no other action */
  }
  return (0); /* all other errors, execute default procedure */
}
```

mblen

Determines the number of bytes in a multibyte character.

Syntax

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Parameters

s	A pointer to a single multibyte character.
n	A variable of type `size_t` that controls the number of characters that `mblen` searches when scanning for a multibyte character. This argument is typically set to `MB_CUR_MAX`.

Return Values

>0	The length of the multibyte character to which *s* points.
–1	The *s* parameter does not point to a valid multibyte character.
=0	The *s* parameter is a null pointer and multibyte character encodings are not state-dependent, or *s* points to a null character.

Description

The `mblen` function examines the multibyte character pointed to by *s*. If a valid multibyte character is recognized within *n* bytes from the location pointed to by *s*, the length of the multibyte character is returned.

This function retains state information. Multibyte encodings can be state-dependent, employing "shift characters" to alter the meaning of subsequent characters. The shift state is persistent between calls to the routines for processing extended character sets unless the `LC_CTYPE` category of the locale is changed.

Calling this function with the *s* argument set to `NULL` resets the function to its initial state. When using a `NULL` pointer to clear the shift state, zero is returned if the multibyte shift state was previously clear. A nonzero value is returned if the locale-specific shift state was previously set.

Locale-specific character sets that are too large to be represented within one byte are handled in ANSI C by using extended character sets. Extended character sets have two representations, the internal representation, and the external representation. The external representation is a multibyte character. The multibyte character is a sequence of normal characters used to represent the locale-specific extended character. The internal representation of this multibyte character is a wide character of type `wchar_t`. The maximum number of bytes in a multibyte character in the current locale (see also `LC_CTYPE`) is given by the macro `MB_CUR_MAX`.

See Also

`wchar_t`, `LC_CTYPE`, `MB_CUR_MAX`, `mbtowc()`, `wctomb()`, `mbstowcs()`, `wcstombs()`, ANSI C 4.10.7.1

mbstowcs

Converts a sequence of multibyte characters in a null-terminated string to a sequence of wide character codes.

Syntax

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Parameters

pwcs	A pointer to an array of wide characters where the converted wide character codes are stored.
s	A pointer to a sequence of multibyte characters to be converted.
n	An expression indicating the maximum number of codes to be stored into the array pointed to by *pwcs*.

Return Values

x	The number of array elements modified, not including a terminating zero. If x = *n*, the array is not zero-terminated.
`size_t(-1)`	Invalid multibyte character found.

Description

The multibyte characters from the array pointed to by *s* are converted to wide character codes and stored into the array pointed to by *pwcs*. No more than *n* elements will be modified in the array pointed to by *pwcs*.

See Also

`wchar_t`, `MB_CUR_MAX`, `mbtowc()`, `wctomb()`, `mbstowcs()`, `wcstombs()`, ANSI C 4.10.8.1

mbtowc

Converts a single multibyte character to its wide character representation.

Syntax

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s,size_t n);
```

Parameters

pwc	A pointer to an object of type `wchar_t` to which the function returns the converted value.
s	A pointer to a multibyte character to be converted.
n	An expression of type `size_t` indicating the number of characters in *s* to be examined. This should be no greater than the value of `MB_CUR_MAX`.

Return Values

>0	The number of bytes that are in the converted multibyte character.
–1	The *s* parameter does not point to a valid multibyte character.
0	The *s* parameter is a null pointer and multibyte character encodings are not state-dependent, or *s* points to a null character.

Description

If *s* is not a null pointer, `mbtowc` determines the number of bytes in the multibyte character pointed to by *s*. It then determines the code for the value of type `wchar_t` that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.)

If the multibyte character is valid and *pwc* is not a null pointer, `mbtowc` stores the code in the object pointed to by *pwc*. A maximum of *n* characters are examined, starting at the character pointed to by *s*.

This function retains state information. Multibyte encodings can be state-dependent, employing "shift characters" to alter the meaning of subsequent characters. The shift state is persistent between calls to the routines for processing extended character sets unless the `LC_CTYPE` category of the locale is changed.

Calling this function with the *s* argument set to `NULL` resets the function to its initial state. When using a `NULL` pointer to clear the shift state, zero is returned if the multibyte shift state was previously clear. A nonzero value is returned if the locale-specific shift state was previously set.

See Also

`wchar_t`, `MB_CUR_MAX`, `wctomb()`, `mbstowcs()`, `wcstombs()`, ANSI C 4.10.7.2

memccpy

Copies characters from one memory location to another until a specified character is found or until the specified count is reached.

Syntax

```
#include <memory.h>
char *memccpy (char *s1, char *s2, int c, int n)
```

Parameters

s1	A pointer to the target string.
s2	A pointer to the source string.
c	The character used to signal the end of the source string.
n	The number of bytes to be copied.

Return Values

x	A character pointer to the first character in *s1* after *c*.
`NULL`	The *c* parameter was not found in *s2*.

Description

The `memccpy` function copies characters from memory area *s2* into *s1*, stopping after the first occurrence of *c* has been copied or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* characters of *s2*. This function operates as efficiently as possible on memory areas (arrays of characters bound by a count that are not terminated by a null character). There is no check for the overflow of the destination memory area. Character movement is performed differently in different implementations. Therefore, avoid overlapping moves.

See Also

`memcpy()`, `memcmp()`, `memchr()`

memchr

Searches memory for a specified character.

Syntax

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Parameters

s	A pointer to the object to search.
c	The character value to find in the object.
n	The maximum number of characters to examine.

Return Values

x	A pointer to the first occurrence of the character. If the character *c* is not found, a null pointer is returned.

Description

The `memchr` function returns a pointer to the first occurrence of character *c* in the object pointed to by *s*. Only the first *n* characters of the *s* array are examined. This function does not terminate when a null character is encountered. Each character is treated as an unsigned character.

See Also

`memcpy()`, `memcmp()`, `memmove()`, `memset()`, ANSI C 4.11.5.1

memcmp

Compares the first *n* characters of two objects.

Syntax

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Parameters

s1	A pointer to the first object.
s2	A pointer to the second object.
n	The number of characters to compare.

Return Values

<0	*s1* is less than *s2*.
0	*s1* is equal to *s2*.
>0	*s1* is greater than *s2*.

Description

The `memcmp` function compares the first *n* characters of the object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*. The result is returned as an integer. Null characters in the objects do not cause this comparison function to stop.

The contents of "holes" used as padding for alignment with structure objects are indeterminate. Strings shorter than their allocated space and unions can also cause comparison problems.

See Also

`memccpy()`, `memchr()`, `memmove()`, `memset()`, `strcmp()`, `strcoll()`, `strxfrm()`, ANSI C 4.11.4.1

memcpy

Copies a specified number of characters from one object to another.

Syntax

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Parameters

s1	A pointer to the target object.
s2	A pointer to the source object.
n	The number of characters to copy.

Return Values

x	The value of *s1*.

Description

The `memcpy` function copies *n* characters from the object pointed to by *s2* to the object pointed to by *s1*. Unlike the `strcpy` function, the `memcpy` function does not stop when a null character is encountered. Use `memmove()` rather than `memcpy()` if the source and destination objects might overlap in memory.

See Also

`memmove()`, `strcpy()`, `memchr()`, `memset()`, ANSI C 4.11.2.1

memmove

Copies a specified number of characters from one object to another.

Syntax

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Parameters

s1	A pointer to the target object.
s2	A pointer to the source object.
n	The number of characters to copy.

Return Values

x	The value of *s1*.

Description

This function copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. The `memmove` function is similar to `memcpy` but allows the source and destination objects to overlap.

See Also

`memcpy()`, `strcpy()`, `memcpy()`, `memchr()`, `memcmp()`, `memset()`, `strncpy()`, ANSI C 4.11.2.2

memset

Initializes an object with a supplied character value.

Syntax

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Parameters

s	A pointer to an object.
c	The value to be duplicated throughout the object that *s* points to.
n	The number of characters in object *s* to be filled with the value *c*.

Return Values

x	The value of *s*.

Description

The `memset` function stores *n* copies of the character *c* into the object pointed to by *s*. The value of *c* is converted to an `unsigned char` before it is stored.

See Also

`memchr()`, `memcmp()`, `memcpy()`, `memmove()`, ANSI C 4.11.6.1

mktemp

Creates a unique file name.

Syntax

```
char *mktemp (char *template)
```

Parameters

template	A character pointer to a string containing a template file name having six trailing Xs.

Return Values

x	A pointer to *template*, or to a null string if it runs out of letters.

Description

The `mktemp` function replaces the contents of the string pointed to by *template* with a unique file name, and returns the address of *template*.

This function replaces the Xs in *template* with a letter and a number. The letter is chosen so that the resulting name does not duplicate the name of an existing file. If there are fewer than 6 Xs in *template*, the letter is dropped first, and then high-order digits of the process ID are dropped.

The `mktemp` function returns the unique file name in *template*. Therefore, you must refresh the template for every unique file you want to open. If `mktemp` runs out of letters, it returns a pointer to the empty string `""`.

`mktemp` does not check to see if the file name part of *template* exceeds the maximum length of a file name.

See Also

`getpid()`, `open()`, `tmpfile()`, `tmpnam()`

mktime

Converts a calendar time value of type tm to a time value in time_t.

Syntax

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Parameters

timeptr	A pointer to a structure of type tm, as defined in <time.h>.

Return Values

x	The value pointed to by *timeptr*, as a type time_t.

Description

The mktime function converts the broken-down time in the structure pointed to by *timeptr* into a calendar time value. The file pointed to by *timeptr* is expressed in the local time. The return value has the same encoding as the values returned by the time function.

The original values of the tm_wday and tm_yday components of the structure (shown below) are ignored.

A positive or zero value for tm_isdst causes the mktime function to presume initially that Daylight Saving Time, respectively, is or is not in effect. A negative value for tm_isdst causes the mktime function to attempt to determine whether Daylight Saving Time is in effect for the specified time. The original values of the tm components are not restricted to the ranges indicated below.

On successful completion, the values of the tm_wday and tm_yday components of the structure are set appropriately. The values of the other components are set to represent the specified calendar time, but with their values forced into valid ranges.

The final value of tm_mday is not set unless tm_mon and tm_year are determined.

The tm data structure is declared in <time.h>. The declaration is shown below:

```
struct tm {
   int tm_sec;    /* seconds after the minute (0 through 59 */
   int tm_min;    /* minutes after the hour (0 through 59) */
   int tm_hour;   /* hours since midnight (0 through 23) */
   int tm_mday;   /* day of the month (1 through 31) */
   int tm_mon;    /* month of the year (0 through 11) */
   int tm_year;   /* years since 1900 */
   int tm_wday;   /* days since Sunday (0 through 6) */
   int tm_yday;   /* day of the year (0 through 365) */
   int tm_isdst;  /* daylight savings time flag  (1 = dst */
};
```

By default, mktime adjusts the returned value to the Eastern Standard Time (EST) zone. You may override this default behavior by using the MPE/iX command SETVAR TZ *name*. Time zone names, and the format of TZTAB.LIB.SYS file containing time zone offsets from GMT are listed in appendix A, "Time Zones."

Example

What day of the week is July 4, 2001?

```
#include <stdio.h>
$include <time.h>
static const char *const wday[] = {
         "Sunday", "Monday", "Tuesday",
         "Wednesday", "Thursday", "Friday",
         "Saturday", "-unknown"
};
struct tm time_str;

time_str.tm_year        = 2001 - 1900;
time_str.tm_mon         = 7 - 1;
time_str.tm_mday        = 4;
time_str.tm_hour        = 0;
time_str.tm_min         = 0;
time_str.tm_sec         = 1;
time_str.tm_isdst       = -1;
if (mktime(time_str) == -1)
        time_str.tm_wday = 7;
printf("%s\n", wday[time_str.tm_wday]);
```

See Also

`clock()`, `difftime()`, `time()`, ANSI C 4.12.2.3, POSIX.1 8.1

modf

Accepts a double value and splits the value into its integer and fractional parts.

Syntax

```
#include <math.h>
double modf (double value, double *iptr);
```

Parameters

value	A real number input to the function.
iptr	A pointer to a real number output from the function containing the integer part of *value*.

Return Values

n	The signed fractional part of *value*.

Description

The modf function splits *value* into two parts, a fraction and an integer, such that *fraction* + *integer* = *value*.

iptr points to a double variable where the integer part of *value* is to be stored. The fractional part of *value* is the return value of the function.

Example

The following program shows how to use the modf function:

```
main(argc, argv)
int argc;
char *argv[ ];
{
   double value, iptr, frac, modf();


   printf("Integer part: %g; Fractional part: %g\n", iptr, frac);
}
```

The program accepts one argument and prints the integer and fractional parts of that value. Note that the address of *iptr* is passed to modf(), because modf() expects the address of a double variable where the integer part can be stored.

See Also

ANSI C 4.5.4.6, POSIX.1 8.1

_mpe_fileno

Maps a file descriptor to an MPE file number.

Syntax

```
int _mpe_fileno(int fildes)
```

Parameters

fildes	A file descriptor.

Return Values

x	The MPE file number of the *fildes*.

Description

The `_mpe_fileno` function returns the MPE file number associated with *fildes*. This file number may be passed to MPE file system intrinsics to access files.

Caution should be used when accessing the same file using both MPE file system intrinsics and C library routines as the calling program is responsible for any coordination required between the two function libraries.

NOTE The `_mpe_fileno` function is not supported in the POSIX/iX library. For equivalent functionality, use the `_MPE_FILENO` macro and include the header file `<fcntl.h>`.

See Also

open( ), dup( )

offsetof

Finds the offset of a member in a structure.

Syntax

```
#include <stddef.h>
offsetof (type, member);
```

Parameters

type	The name of a structured data type.
member	The name of an element within the data structure *type*.

Return Values

x	The byte offset of *member* within the structure *type* returned as an unsigned integer of type `size_t`.

Description

The `offsetof` macro calculates the offset in bytes of *member* from the beginning of the structure (*type*). The value returned is a variable of type `size_t`, which is defined in `<stddef.h>`.

See Also

ANSI C 4.1.5

open

Opens a file for reading or writing.

Syntax

```
#include <fcntl.h>
int open (char *path, int oflag [,int mode [,char mpe_opts]] );
```

Parameters

path	A pointer to a path name naming a file.
oflag	An integer containing open mode bit flags.
mode	An unused integer parameter provided for compatibility with other systems.
mpe_opts	A pointer to a string containing file attributes and options.

Return Values

x	Upon successful completion, the integer file descriptor is returned.
–1	Unsuccessful completion. In addition, errno is set to the indicated value if one of the following conditions is true:

ENOENT	The *fname* is null, or the named file does not exist and you did not use *oflag* to request that it be created.
EACCES	The *oflag* permission is denied for the named file.
EMFILE	The maximum number of file descriptors allowed are currently open.
EINVAL	The *oflag* specifies incompatible read/write access flags.
ESYSERR	A call to a system intrinsic failed.

Description

The open function opens the file descriptor described in *fname*. It uses the value of *oflag* to determine how to open the file.

Opening a file in read mode fails if the file does not exist or cannot be read.

The *oflag* parameter values are constructed by OR-ing flags from the list below. Notice that exactly one of the first three flags below must be used.

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_APPEND	If set, the file pointer is set to the end of the file prior to each write.

`O_CREAT`	If the file exists, this flag has no effect. If the file does not exist, the file is created.
`O_TRUNC`	If the file exists, its length is truncated to zero and the mode and owner are unchanged. The file pointer used to mark the current position within the file is set to the beginning of the file.
`O_MPEOPTS`	If this flag is specified, the argument *`mpe_opts`* specifies additional `open` options that provide greater control in the MPE file environment.

The *`mode`* parameter is ignored. It is provided for compatibility with other systems.

The *`mpe_opts`* argument points to a string of characters described below. Spaces can be used in the *`mpe_opts`* string to improve readability. Notice that the case of the options is important. An uppercase `B` is different from a lowercase `b`.

`b`	If the `b` option is specified, the file is created as a binary file if this call to `fopen` creates the file. The default is to create an ASCII file.
`Bl`*`n`*	The `Bl` option specifies the blocking factor to use if this call to `fopen()` creates the file. The option character is followed by an integer that indicates the blocking factor. If the `Bl` option is not specified, then the default is one record per block.
`Bs`	If the `Bs` option is specified, the file is opened or created as a byte stream file. This is the only required option for opening byte stream files. The maximum file size for a byte stream file is two gigabytes. If specified, the `Rn` option is ignored. The `S`*`n`* option can be used to reset the file size. This option is mutually exclusive with the `V` option. If the `Bs` or `V` options are not specified, the file is created with an MPE fixed-length record format.
`Bu`*`n`*	The `Bu` option specifies the number of buffers to be allocated to this file. If the `Bu` option is not specified, the default is 2.
`C`	If the `C` option is specified, then the file accepts carriage control information. The default is to not have carriage control.
`Df`*`n`*	The `Df` option specifies the final disposition of the file after the file is closed. The affect of each value of *`n`* is defined as follows:

0	Don't change the disposition.
1	Save the file as a permanent file.
2	Save the file as a temporary file.
3	Don't rewind on close.
4	Purge the file on close.

If the `Df` option is not specified and the file is a new file, then the default is to save the file as a permanent file. If the file is old, the default is not to change the disposition.

`Ds`*`n`*	The `Ds` option specifies the disk space disposition of the file after the file is closed for fixed, undefined, and variable format files. The affect of each

value of *n* is defined as follows:

0	Don't return any disk space allocated beyond the end-of-file indicator.
1	Return to the system any disk space allocated beyond the end-of-file indicator. The EOF becomes the file limit. No records may be added to the file beyond this new limit.
2	Return to the system any disk space allocated beyond the end-of-file indicator, but do not set the file limit to EOF, and allow records to be added to the file up to the file limit.

If the `Ds` option is not specified, the default is not to return any disk space allocated beyond the end-of-file indicator.

`E`*n*	The `E` option specifies the maximum number of extents that can be allocated to the file. The maximum value is 32. The default value, if the `E` option is not specified, is 8 extents.
`F`*n*	The `F` option indicates the value used as the file code if this call to `fopen()` creates the file. If the `F` option is not specified, the file code is zero.
`L`	If specified, the `L` option indicates that dynamic locking should be allowed on this file.
`M`*n*	The `M` option controls multiaccess. The option character is followed by an integer that indicates the level of multiaccess for this open request. The levels are specified in the *MPE/iX Intrinsics Reference Manual* under the `FOPEN` intrinsic description.
`Q`	If the `Q` option is specified, file equations are disallowed. The default is to allow file equations.
`R`*n*	The `R` option specifies the size of the record if the file is created by this open request. If the `V` option is also used, this option specifies the maximum size of the variable-sized records. The option letter is followed by a decimal number that is equal to the number of bytes in the record size. Notice that the number must be positive. A byte count is always specified. If the `R` option is not provided, then the default record size is 256 bytes.
`S`*n*	The `S` option specifies the maximum size of the file. The value of *n* is the maximum size of the file in records for text and binary streams, and in bytes for byte streams. Notice that if the `S` parameter is not specified, the default is 4095.
`Te`	If the `Te` option is specified, the file is saved in the temporary file domain. If the `Te` option is not specified and the file is a new file, the default is to save the file as a permanent file. If the file is old, the default is to not change the disposition.
`Tm`	If the `Tm` option is specified, disk read functions trim editor line numbers, if they exist, and trailing blanks from each record of an ASCII fixed record length file before returning file data to the reader. This option is used on files opened with read only access. Random access to file data using `fseek()` and `lseek()` is not permitted. The default is to not trim editor line numbers and blanks.

U*n*	If the U option is specified, the file is created with *n* user-label records. If this option is not specified, the default is no user-label records.
V	If the V option is specified, the file is created with an MPE variable-length record format. If the V or Bs options are not specified, then the file is created with an MPE fixed-length record format. This option is mutually exclusive with the Bs option.
X*n*	The X option controls exclusive access ability for the file. The option character is followed by an integer that indicates the level of exclusivity for this open request. The levels are specified in the *MPE/iX Intrinsics Reference Manual* under the FOPEN intrinsic.

NOTE If linking with the POSIX/iX library, refer to the description of open() located in the *MPE/iX Developer's Kit Reference Manual*.

Examples

The following creates or opens a fixed record binary file f1 for writing with 256 byte records, a file size of 10000 records, and a file code of 1030:

```
#include <fcntl.h>
int fd;
fd = open("f1",O_WRONLY | O_CREAT | O_MPEOPTS, 0664, "b R256
s10000 F1030");
```

To open an existing file f1 for reading:

```
#include <fcntl.h>
int fd;
fd = open("f1",O_RDONLY);
```

See Also

fopen()

perror

Prints an error message corresponding to `errno`.

Syntax

```
#include <stdio.h>
void perror (const char *s);
```

Parameters

s	A pointer to an optional string to be printed with the error message. If a null pointer is passed, the parameter is ignored.

Return Values

None.

Description

The `perror` function prints an error message corresponding to the value of `errno`. First, if the argument *s* is not a null pointer or a pointer to a null character, the string *s* is printed, followed by a colon and a blank, then the message and a newline character are printed.

See Also

`errno`, `strerror()`, ANSI C 4.9.10.4, POSIX.1 8.1

pow

Returns the value of *x* raised to the power *y*.

Syntax

```
#include <math.h>
double pow (double x, double y);
```

Parameters

x	A real number.
y	A real number.

Return Values

n — The value of x^y.

0 — Indicates any of the following:

- The *x* parameter is zero and *y* is non-positive. The `errno" variable is set to `EDOM`. A `DOMAIN` error message is also printed on the standard error output.
- The *x* parameter is negative and *y* is not an integer. The `errno` variable is set to `EDOM`. A `DOMAIN` error message is also printed on the standard error output.
- An underflow condition has occurred, and `errno` is set to `ERANGE`.

±`HUGE_VAL` — An overflow condition has occurred, and `errno` is set to `ERANGE`.

Description

The `pow` function returns x^y. If *x* is zero, *y* must be positive. If *x* is negative, *y* must be an integer. Error handling can be changed by a user-written `matherr` function.

See Also

`matherr()`, ANSI C 4.5.5.1, POSIX.1 8.1

printf

Writes data in formatted form to the standard output stream `stdout`.

Syntax

```
#include <stdio.h>
int printf (const char *format [,item [,item]...] );
```

Parameters

format	A pointer to a character string defining the format (or the character string itself enclosed in double quotes).
item,...	Each *item* is a variable or expression specifying the data to print.

Return Values

≥0	If successful, the number of characters written.
<0	An error occurred.

Description

The `printf` function enables you to output data in formatted form. In the `printf` function, *format* is a pointer to a character string (or the character string itself enclosed in double quotes) that specifies the format and content of the data to be printed. Each *item* is a variable or expression specifying the data to print.

The `printf()` format is similar to the `scanf` function. It is made up of conversion specifications and literal characters. Literal characters are all characters that are not part of a conversion specification. Literal characters are printed on `stdout` exactly as they appear in the format.

Conversion Specifications

The following list shows the different components of a conversion specification in their correct sequence:

1. A percent sign (`%`), which signals the beginning of a conversion specification; to output a literal percent sign, you must type two percent signs (`%%`).
2. Zero or more flags, which affect the way a value is printed (see below).
3. An optional decimal digit string which specifies a minimum *field width*.
4. An optional *precision* consisting of a dot (`.`) followed by a decimal digit string.
5. An optional `l`, `h`, or `L` indicating that the argument is of an alternate type. When used in conjunction with an integer conversion character, an `l` or `h` indicates a long or short integer argument, respectively. When used in conjunction with a floating-point conversion character, an `L` indicates a long double argument.

6. A conversion character, which indicates the type of data to be converted and printed.

A one-to-one correlation must exist between each specification encountered and each item in the item list.

The available *`flags`* are:

`-`	Causes the data to be left-justified within its output field. Normally, the data is right-justified.
`+`	Causes all signed data to begin with a sign (`+` or `-`). Normally, only negative values have signs.
`blank`	Causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the `blank` and `+` flags both appear, the `blank` flag is ignored.
`#`	Causes the data to be printed in an *alternate form*. Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.
`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. Space padding is not performed. If the `0` and `-` flag s both appear, the `0` flag is ignored. The `0` flag is also ignored for `d`, `i`, `o`, `u`, `x`, and `X` conversions if a precision is specified.

A *`field width`*, if specified, determines the *minimum* number of spaces allocated to the output field for the particular piece of data being printed. If the data happens to be smaller than the field width, the data is blank-padded on the left (or on the right, if the `-` flag is specified) to fill the field. If the data is larger than the *`field width`*, the *`field width`* is simply expanded to accommodate the data. An insufficient *`field width`* never causes data to be truncated. If *`field width`* is not specified, the resulting field is made just large enough to hold the data.

The *`precision`* is a value which means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

NOTE A *`field width`* or *`precision`* may be replaced by an asterisk (`*`). If so, the next item in the item list is fetched, and its value is used as the *`field width`* or *`precision`*. The *`item`* fetched must be an integer.

Conversion Characters

Conversion characters specify the type of data to expect in the item list and cause the data to be formatted and printed appropriately. The integer conversion characters are:

`d`, `i`	An integer *`item`* is converted to signed decimal. The *`precision`*, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the *`precision`*, the value is expanded with

leading zeros. The default *precision* is 1. A null string results if a zero value is printed with a zero *precision*. The # flag has no effect.

`u`	An integer *item* is converted to unsigned decimal. The effects of the *precision* and the # flag are the same as for `d`.
`o`	An integer *item* is converted to unsigned octal. The # flag, if specified, causes the *precision* to be expanded, and the octal value is printed with a leading zero (a C convention). The *precision* behaves the same as in `d` above, except that printing a zero value with a zero *precision* results in only the leading zero being printed, if the # flag is specified.
`x`	An integer *item* is converted to hexadecimal. The letters `abcdef` are used in printing hexadecimal values. The # flag, if specified, causes the *precision* to be expanded, and the hexadecimal value is printed with a leading `"0x"` (a C convention). The *precision* behaves as in `d` above, except that printing a zero value with a zero *precision* results in only the leading `"0x"` being printed, if the # flag is specified.
`X`	Same as `x` above, except that the letters `ABCDEF` are used to print the hexadecimal value, and the # flag causes the value to be printed with a leading `"0X"`.

The character conversion characters are as follows:

`c`	The character specified by the `char` *item* is printed. The *precision* is meaningless, and the # flag has no effect.
`s`	The string pointed to by the character pointer *item* is printed. If a *precision* is specified, characters from the string are printed until the number of characters indicated by the *precision* is reached, or until a null character is encountered, whichever comes first. If the *precision* is omitted, all characters up to the first null character are printed. The # flag has no effect.

The floating-point conversion characters are:

`f` The `float` or `double` *item* is converted to decimal notation in style `f`; that is, in the form

```
[-]ddd.ddd
```

where the number of digits after the decimal point is equal to the *precision*. If *precision* is not specified, six digits are printed after the decimal point. If the *precision* is explicitly zero, the decimal point is eliminated entirely. If the # flag is specified, a decimal point always appears, even if no digits follow the decimal point.

`e` The `float` or `double` *item* is converted to scientific notation in style `e`; that is, in the form

```
[-]d.ddde±ddd
```

where there is always one digit before the decimal point. The number of digits after the decimal point is equal to the *precision*. If *precision* is not given, six digits are printed after the decimal point. If the *precision* is explicitly zero, the decimal point is eliminated entirely. The exponent

	always contains exactly three digits. If the # flag is specified, the result always contains a decimal point, even if no digits follow the decimal point.
`E`	Same as `e` above, except that `E` is used to introduce the exponent instead of `e` (style `E`).
`g`	The `float` or `double` *`item`* is converted to either style `f` or style `e`, depending on the size of the exponent. If the exponent resulting from the conversion is less than -4 or greater than the *`precision`*, style `e` is used. Otherwise, style `f` is used. The *`precision`* specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the # flag is specified, the result always has a decimal point, even if no digits follow the decimal point, and trailing zeros are *not* removed.
`G`	Same as the `g` conversion above, except that style `E` is used instead of style `e.`
`p`	The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters.
`n`	The argument is a pointer to an integer into which is written the number of characters written to the output stream so far by this call to `fprintf()`. No argument is converted.
`%`	A `%` is written. No argument is converted. The complete conversion specification is &%&%.

The *`item`*s in the item list may be variable names or expressions. Note that, with the exception of the `s` conversion, pointers are *not* required in the item list. If the `s` conversion is used, a pointer to a character string must be specified.

Examples

Some examples of `printf()` conversion specifications and a brief description are shown below:

`%d`	Output a signed decimal integer. The field width is just large enough to hold the value.
`%-*d`	Output a signed decimal integer. The left-justify flag `(-)` and the blank flag are specified. The asterisk causes a *`field width`* value to be extracted from the item list. Thus, the item specifying the desired field width must occur before the item containing the value to be converted by the `d` conversion character.
`%+7.2f`	Output a floating-point value. The + flag causes the value to have an initial sign (+ or -). The value is right-justified in a 7-column field, and has exactly two digits after the decimal point. This conversion specification is ideal for a debit/credit column on a finance worksheet. (If the + sign is not necessary, use the blank flag instead.)

The following program reads a number from `stdin` and prints the number, followed by its square and its cube:

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

```
   main()
   {
      double x;

      printf("Enter your number: ");

      printf("Your number is %g\n", x);
      printf("Its square is %g\nIts cube is %g\n", x*x, x*x*x); }
```

The `g` conversion character is used so that the decision about using an exponent is automatic. Note that the item list contains expressions to calculate `x` squared and `x` cubed. Also note that the address of the variable is required in order to read a value for it with `scanf()`, but `printf()` requires the variable name itself.

The following program accepts a decimal integer and prints the integer itself, its square, and its cube in decimal, octal, and hexadecimal. The program also prints the headings "Decimal," "Octal," and "Hexadecimal" and prints the data in tabular form.

```
   #include <stdio.h>
   main()
   {
      long n, n2, n3;

   /* get value */

      printf("Enter your number: ");


   /* print headings */

       printf("\n\n              Decimal     Octal     Hexadecimal\n");

    /* do the computation */

      n2 = n * n;
      n3 = n * n * n;
      printf("n itself:    %7ld   %9lo     %6lx\n", n, n, n);
      printf("n squared:   %7ld   %9lo     %6lx\n", n2, n2, n2);
      printf("n cubed:     %7ld   %9lo     %6lx\n", n3, n3, n3);
   }
```

Strings are easy to manipulate using the `printf()` function. The following program shows how strings can be inserted in text.

```
   #include <stdio.h>
   main()
   {
      char first[15], last[25];

      printf("Enter your first and last names: ");
      scanf("%s%s", first, last);
      printf("\nWell, hello %s, it's good to meet you!\n", first);
      printf("%s, huh?  Are you any relation to that famous\n", last);
      printf("computer programmer, Mortimer Zigfelder %s?\n", last);
      printf("No, sorry, that was my mistake.  I was thinking of\n");
      printf("O'%s, not %s.\n", last, last);
    }
```

See Also

fprintf(), vprintf(), sprintf(), putc(), setlocale(), scanf(), ANSI C 4.9.6.3, POSIX.1 8.1

printmsg

Prints formatted output with numbered arguments to `stdout`.

Syntax

```
#include <stdio.h>
int printmsg (format [, arg] ...)
   char *format;
```

Parameters

format	A pointer to the string containing the formatting information. It contains optional placeholders and formatting specifications where *arg1* through *argn* are to be substituted.
arg1 ... argn	A character, character pointer or integer value giving the parameter to be converted, formatted, and merged with *format* prior to output.

Return Values

x	The number of characters transmitted.
`EOF`	An error occurred.

Description

The `printmsg` function places output on the standard output stream `stdout` after performing parameter substitution.

The `printmsg` function is derived from `printf()`. In `printmsg()`, the conversion character `%` is replaced by the sequence `%n$`. `n` is a decimal digit in the range 1-9, and indicates that this conversion should be applied to the `n`th argument, rather than to the next unused one. All other aspects of formatting are unchanged. All conversion specifications must contain the `%n$` sequence, and you should make sure the numbering is correct. All parameters must be used exactly once.

See `printf()` for more details on formatting and conversion specifications.

Example

The following creates a date and time printing function:

```
printmsg(format, weekday, month, day, hour, min);
```

The format is a pointer to the following string:

```
"%1$s, %2$s %3$d, %4$d:%5$.2d\n"
```

The resulting output is:

```
Sunday, July 3, 10:02
```

See Also

`printf()`, `fprintmsg()`, `sprintmsg()`

putc

Writes a character to an open stream.

Syntax

```
#include <stdio.h>
int putc (int c, FILE *stream);
```

Parameters

c	A character to be written to an open stream.
stream	A pointer to an open stream.

Return Values

x	The character written.
EOF	An error occurred, and errno is set to indicate the error condition.

Description

The putc function writes a single character to the specified stream. This function is equivalent to fputc() except that it is implemented as a macro. Because putc() can evaluate the stream more than once, the arguments should never be an expression with side effects.

Example

Refer to the example located in the getc function description.

See Also

fputc(), getc(), putchar(), puts(), fwrite(), ANSI C 4.9.7.8, POSIX.1 8.1

putchar

Writes a character to the standard output stream `stdout`.

Syntax

```
#include <stdio.h>
int putchar (int c);
```

Parameters

c	A character to be written to `stdout`.

Return Values

x	The character written to `stdout`.
`EOF`	An error occurred; `errno` is set to indicate the error condition.

Description

The `putchar` function writes a single character *c* to the standard output stream `stdout`. The `putchar(`*c*`)` function is equivalent to `putc(`*c*`,` *stdout_ptr*`)`.

Examples

Refer to the examples located in the `getchar` function description.

See Also

`fputc()`, `putc()`, `puts()`, `fwrite()`, `getchar()`, ANSI C 4.9.7.9, POSIX.1 8.1

puts

Writes a string to the standard output stream `stdout`.

Syntax

```
#include <stdio.h>
int puts (const char *s);
```

Parameters

s	A pointer to a character array containing the string to be written to `stdout`. The character array must be terminated with a null character.

Return Values

≥0	Success.
`EOF`	An error occurred.

Description

The `puts` function writes the string from a character array pointed to by *s* to the standard output stream `stdout.` The string is terminated by a null character in the array, which `puts()` replaces with a new line in the output.

Examples

The following example uses `gets()` and `puts()`:

```
#include <stdio.h>
main()
{
   char line[80], *gets();

   while((gets(line)) != NULL)
      puts(line);
}
```

To terminate this program, generate an end of file on `stdin.` Using string comparison and string length functions, you can write a termination condition for this program.

See Also

`fputc()`, `fwrite()`, `gets()`, `putc()`, `putchar()`, ANSI C 4.9.7.10, POSIX.1 8.1

putw

Writes a word to an open stream.

Syntax

```
#include <stdio.h>
int putw (int w, FILE *stream);
```

Parameters

w	A word to be written to an open stream.
stream	A pointer to an open stream.

Return Values

0	Indicates success.
Non-zero	An error occurred.

Description

The `putw` function writes the word (`int` in C) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. The `putw` function neither assumes nor causes special alignment in the file.

See Also

`putc()`, `putchar()`, `fputc()`

qsort

Sorts an array of objects.

Syntax

```
#include <stdlib.h>
void qsort (void *base, size_t nmemb, size_t size,
            int (*compar) (const void *,  const void *));
```

Parameters

base	A pointer to an array to be sorted.
nmemb	The number of elements in the array.
size	The size, in bytes, of each element of the array.
compar	A pointer to a user-written comparison function.

Return Values

None.

Description

The qsort function sorts an array of objects. The *size* parameter specifies the size of each object.

The contents of the array are sorted in ascending order as determined by the user-written comparison function *compar*, which is called with two arguments that point to the objects being compared. The function must return an integer less than, equal to, or greater than zero as a consequence of whether its first argument is to be considered less than, equal to, or greater than the second.

The order of two members that compare as equal in the sorted array is unspecified.

See Also

ANSI C 4.10.5.2, POSIX.1 8.1

raise

Causes a signal to be raised.

Syntax

```
#include <signal.h>
int raise (int sig);
```

Parameters

sig	A signal number specifying the signal to be raised.

Return Values

0	The signal was successfully raised.
≠0	The signal was not raised.

Description

The `raise` function causes the signal specified in *sig* to be raised to the calling process.

The name and meaning of each signal is given below:

Name	**Description**
SIGABRT	Abnormal termination, (for example, by the `abort` function).
SIGFPE	An erroneous arithmetic operation, (for example, divide by 0).
SIGILL	An illegal instruction was executed (possibly after a jump).
SIGINT	An interactive interrupt signal was received.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request was sent to the program.

NOTE Signals are provided for conformance with ANSI C. However, the only way to generate a signal on MPE/iX is by an explicit call to the `raise` function. For information on a more comprehensive facility for handling exceptions, see the *Trap Handling Programmer's Guide*.

Examples

Refer to the example located in the `signal` function description.

See Also

`signal()`, ANSI C 4.7.2.1

rand

Returns a random number.

Syntax

```
#include <stdlib.h>
int rand (void);
```

Parameters

None.

Return Values

x	A pseudo-random integer in the range 0 to RAND_MAX. The macro RAND_MAX expands to the value 32767.

Description

If the srand function is not used to initialize the random number generator to a particular starting point, rand() returns the same sequence of numbers every time the program is executed.

See Also

rand(), srand(), ANSI C 4.10.2.1, POSIX.1 8.1

rand48

The `drand48`, `erand48`, `lrand48`, `nrand48`, `mrand48`, `jrand48`, `srand48`, `seed48`, and `lcong48` functions generate uniformly distributed pseudo-random numbers.

Syntax

```
double drand48 ( )

double erand48 (xsubi)
   unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
   unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
   unsigned short xsubi[3];

void srand48 (seedval)
   long seedval;

unsigned short *seed48 (seed16v)
   unsigned short seed16v[3];

void lcong48 (param)
   unsigned short param[7];
```

Parameters

xsubi	A pointer to a 3-word (48-bit) `unsigned short int` array used by the random number generator to store successive values of X.
seedval	A 32-bit seed value used to initialize the high-order bits of seed value to the random number generator
seed16v	A pointer to a 3-word (48-bit) `unsigned short int` array used internally by the random number generator to hold the previous value of the seed.
param	A pointer to a 7-word `unsigned short int` array arranged as follows:

param[0-2]	The 48-bit seed value.
param[3-5]	The multiplier A used to expand the random number from the 0 to 1 range to the desire range.
param[6]	The addend C used to shift the random number from the 0 to 1 range to the desired range.

Return Values

x	Random numbers appropriate to the type and function called (except *seed*, which returns a pointer to the internal buffer where X is stored).

Description

This family of functions generates uniform pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The `drand48` and `erand48` functions return non-negative double-precision floating-point values uniformly distributed over the interval of 0.0 (inclusive) to 1.0 (non-inclusive) or, in mathematical nomenclature, (0.0,1.0).

The `lrand48` and `nrand48` functions return non-negative long integers uniformly distributed over the interval of 0 (inclusive) to 2^{31} (non-inclusive), or $(0, 2^{31})$.

The `mrand48` and `jrand48` functions return signed long integers uniformly distributed over the interval of -2^{31} (inclusive) to 2^{31} (non-inclusive), or $(-2^{31}, 2^{31})$.

The `srand48`, `seed48` and `lcong48` functions are initialization entry points, one of which should be invoked before either `drand48`, `lrand48` or `mrand48` is called. Although it is not recommended practice, constant default initializer values are supplied automatically if `drand48`, `lrand48` or `mrand48` is called without a prior call to an initialization entry point. The `erand48`, `nrand48`, and `jrand48` functions do not require an initialization entry point to be called first.

All the functions work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula:

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \quad n \geq 0.$$

The parameter $m = 2^{48}$; therefore 48-bit integer arithmetic is performed. Unless `lcong48` has been invoked, the multiplier value a and the addend value c are given by:

$$a = 5DEECE66D_{16} = 273673163155_8$$

$$c = B_{16} = 13_8$$

The value returned by `drand48`, `erand48`, `lrand48`, `nrand48`, `mrand48` or `jrand48` is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The `drand48`, `lrand48` and `mrand48` functions store the last 48-bit X_i generated in an internal buffer, which is why they must be initialized prior to being invoked. The `erand48`, `nrand48` and `jrand48` functions require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these functions do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, the `erand48`, `nrand48` and `jrand48` functions allow separate modules of a large program to generate several independent streams of pseudo-random numbers. For example, the sequence of numbers in each stream does not depend upon how many times the functions have been called to generate numbers for the other streams.

The initializer function `srand48` sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function `seed48` sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by `seed48`, and a pointer to this buffer is the value returned by `seed48`. This returned pointer, which can be ignored if not needed, is useful if a program is to be restarted from a given point at some future time. Use the pointer to get at and store the last X_i value, and then use this value to reinitialize using `seed48` when the program is restarted.

The initialization function `lcong48` allows the user to specify the initial X_i, the multiplier value *a*, and the addend value *c*. Argument array elements *`param[0-2]`* specify X_i, *`param[3-5]`* specify the multiplier *a*, and *`param[6]`* specifies the 16-bit addend `c`. After `lcong48` is called, a subsequent call to either `srand48` or `seed48` restores the standard multiplier and addend values *a* and *c*, as specified previously.

See Also

`rand()`

read

Reads input from a file.

Syntax

```
int read (int fildes, char *buffer, unsigned nbyte);
```

Parameters

`fildes`	An open file descriptor.
`buffer`	A pointer to a buffer where the function returns data.
`nbyte`	The number of bytes to read and place in *`buffer`*.

Return Values

>0 Indicates success and the number of bytes read. This number may be less than *`nbyte`* if:

- The file is associated with a communication line.
- The number of bytes left in the file is less than *`nbyte`* bytes.

`EOF` Returned when an end-of-file is reached.

–1 Indicates unsuccessful completion. The `errno` variable is set if one of the following conditions is true:

`EBADF`	The *`fildes`* parameter is not a valid file descriptor open for reading.
`ESYSERR`	A call to a system intrinsic failed.

Description

The `read` function reads *`nbyte`* bytes from the file associated with *`fildes`* and places the data read into the buffer pointed to by *`buffer`*.

On devices capable of seeking, `read()` starts at a position in the file given by the file offset associated with *`fildes`*. Upon return from `read()`, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file offset associated with such a device is undefined.

Unless an error occurs, a process blocks until a `read()` request is completed.

NOTE If linking with the POSIX/iX library, refer to the description of `read()` located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

`open()`, `write()`

realloc

Changes the size of a block of allocated memory.

Syntax

```
#include <stdlib.h>
void *realloc (void *ptr, size_t size);
```

Parameters

ptr	A pointer to a block of memory previously allocated.
size	The new size, in bytes.

Return Values

x	A successful call to `realloc()` returns a pointer to the possibly moved block of allocated memory.
`NULL`	There is not enough available memory. The block pointed to by *ptr* is left intact, or, *size* is 0.

Description

The `realloc` function changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the block. The location of the block may be changed by this function. The contents are unchanged up to the lesser of the new and old sizes.

See Also

`malloc()`, `free()`, `calloc()` ANSI C 4.10.3.4, POSIX.1 8.1

remove

Purges an existing file.

Syntax

```
#include <stdio.h>
int remove (const char *filename);
```

Parameters

filename	A pointer to a character array containing the name of a file to purge. The string must be terminated by a null character.

Return Values

0	The file is successfully purged.
–1	An error occurred and `errno` is set to one of the following values:

`ENOENT`	The file does not exist.
`ESYSERR`	A call to a system intrinsic failed.

Description

The `remove` function purges the specified file from the file system. The call fails if *filename* references an open file.

See Also

ANSI C 4.9.4.1, POSIX.1 8.1

rename

Renames an existing file.

Syntax

```
#include <stdio.h>
int rename (const char *oldname, const char *newname);
```

Parameters

oldname	A pointer to a string containing the name of the existing file whose name is to be changed. The string must be terminated by a null character.
newname	A pointer to a string containing the new name of the file. The string must be terminated by a null character.

Return Values

0	The file is successfully renamed.
–1	An error occurred. The file is not renamed.

Description

The `rename` function changes the file named by *oldname* so that it has the name *newname*.

NOTE The `rename` function is not supported in the POSIX/iX library. If called, `rename()` returns a -1 and sets `errno` to ENOSYS to indicate that `rename()` is not supported.

See Also

`remove()`, ANSI C 4.9.4.2

rewind

Sets the file position indicator for a stream to the beginning of the file.

Syntax

```
#include <stdio.h>
void rewind (FILE *stream);
```

Parameters

stream	A pointer to an open stream.

Return Values

None.

Description

The `rewind` function sets the file position indicator for the specified stream to the beginning of the file.

NOTE If you have a stream open for both reading and writing, a read operation cannot be followed by a write operation without one of the following occurring first: a `rewind()`, an `fseek()`, or a read operation that encounters end-of-file. Similarly, a write operation cannot be followed by a read operation unless a `rewind()` or `fseek()` is performed.

Example

Suppose you sometimes wish to use a password on a data file accessed by an application program. This password is to be optionally stored in encrypted form on the first line of the file. The line is recognized as a password line if the first two characters are `"*P"`. If the file has no password line, access to the file is unrestricted. If a password line is found, the program prompts for the password before permitting access. The following code looks for a password line:

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[ ];
{
   FILE *pswd;
   char line[256];

   if(argc != 2) {
      fprintf(stderr, "Usage: getpswd file\n");
      exit(1);
      }
      pswd = fopen(argv[1], "r");
      if(pswd == NULL) {
         fprintf(stderr, "Can't open %s.\n", argv[1]);
         exit(1);
      }
      fgets(line, 256, pswd);
      if(line[0] == '*'  line[1] == 'P') {
      /* ask for and check password */
      } else
         rewind(pswd);
      /* application program goes here */
      exit(0);
   }
```

If the first two characters of the first line are `"*P"`, the code that asks for and checks a password is executed. However, if the first line is not a password line, the file is assumed to be unprotected. Thus, the file must be rewound so the data contained in the first line is available to the application program.

See Also

`ftell()`, `fseek()`, ANSI C 4.9.9.5, POSIX.1 8.1

scanf

Reads externally formatted data from the standard input stream `stdin`.

Syntax

```
#include <stdio.h>
int scanf (const char *format [,item [,item]...] );
```

Parameters

format	A pointer to a character string defining the format of the data to be read (or the character string itself enclosed in double quotes).
item	Each *item* is the address of a variable into which the data will be placed.

Return Values

≥0	The number of successfully matched and assigned input items.
`EOF`	An error occurred on input (no input characters, or a matching error occurred before any conversion).

Description

The `scanf` function reads externally formatted data from the standard input stream `stdin`, converts the data to internal format, and stores the results in a string of arguments.

In the `scanf` function, *format* is a character pointer to a character string (or the character string itself enclosed in double quotes), and *item* is the address of a variable. The `scanf` function returns the number of successfully matched and assigned input items or returns `EOF` if there are no input characters available or if a matching error occurred before any conversion was made.

The purpose of the format is to specify how the data to be read is presented on `stdin` and what types of data are found there. The format consists of white-space characters, conversion specifications, and literal characters.

White-Space Characters

White-space characters (blanks, tabs, newlines, or form feeds) cause input to be read up to the next non-white-space character.

Conversion Specifications

A conversion specification is a character sequence that tells `scanf()` how to interpret the data received at that point in the input.

In the format, a conversion specification is introduced by a percent sign `(%)`, optionally followed by an asterisk `(*)` (called the assignment suppression character), optionally followed by an integer value (called the *field width*). The conversion specification is terminated by a character specifying the type of data to expect; the terminating characters are called conversion characters. The integer and floating-point conversion characters may be optionally preceded by a character indicating the size of the receiving variable.

When a conversion specification is encountered in a format, it is matched up with the corresponding item in the item list. The data formatted by that specification is then stored in the location pointed to by that item. For example, if there are four conversion specifications in a format, the first specification is matched up with the first item, the second specification with the second item, and so on.

The number of conversion specifications in the format is directly related to the number of items specified in the item list. With one exception, there must be at least as many items as there are conversion specifications in the format. If there are too few items in the item list, an error occurs; if there are too many items, the excess items are ignored. The one exception occurs when the assignment suppression character (`*`) is used. If an asterisk occurs immediately after the percent sign (before the *field width*, if any), the data formatted by that conversion specification is discarded. No corresponding item is expected in the item list; this is useful for skipping over unwanted data in the input.

Conversion Characters

There are 14 conversion characters: five format integer data, three format character data, three format floating-point data, and three special characters.

The integer conversion characters are:

`d`	A decimal integer is expected.
`i`	A signed integer is expected.
`o`	An octal integer is expected.
`u`	An unsigned decimal integer is expected.
`x`	A hexadecimal integer is expected.

The character conversion characters are:

`c`	A single character is expected, normal skip over leading white space is suppressed.
`s`	A character string is expected.
`[`	A character string is expected, normal skip over leading white space is suppressed.

The floating-point conversion characters are:

`e, f, g`	A floating-point number is expected (the capitalized forms of these characters are also accepted).

The special characters are:

`p`	Matches an implementation-defined set of sequences.
`n`	No input is consumed. The corresponding argument is a pointer to an integer into which is written the number of characters read from the input

stream so far by this call to `fscanf()`.

`%`	Matches a single `%`. No conversion or assignment occurs. The complete conversion specification is &%&%

Integer Conversion Characters

The `d`, `o`, and `x` conversion characters read characters from `stdin` until an inappropriate character is encountered, or until the number of characters specified by the *`field width`*, if given, is exhausted (whichever comes first).

For `d`, an inappropriate character is any character except +, -, and 0 through 9. For `o`, an inappropriate character is any character except +, -, and 0 through 7. For `x`, an inappropriate character is any character except +, -, 0 through 9, and the characters `a` through `f` and `A` through `F`. Note that negative octal and hexadecimal values are stored in their twos complement form with sign extension. Thus, they might look unfamiliar if you print them out later using `printf()`.

These integer conversion characters can be preceded by a `l` to indicate that a `long int` should be expected rather than an `int.` They can also be preceded by `h` to indicate a `short int.` The corresponding items in the item list for these conversion characters must be pointers to integer variables of the appropriate length.

Character Conversion Characters

The `c` conversion character reads the next character from `stdin`, no matter what that character is. The corresponding item in the item list must be a pointer to a character variable. If a *`field width`* is specified, the number of characters indicated by the *`field width`* are read. In this case, the corresponding item must refer to a character array large enough to hold the characters read.

Note that strings read using the `c` conversion character are not automatically terminated with a null character in the array. Because all C library functions that use strings assume the existence of a null terminator, be sure to add the `'\0'` character yourself. If you do not, library functions are not able to tell where the string ends and you will get unexpected results.

The `s` conversion character reads a character string from `stdin` which is delimited by one or more space characters (blanks, tabs, or newlines). If no *`field width`* is given, the input string consists of all characters from the first nonspace character up to (but not including) the first space character. Any initial space characters are skipped over. If a *`field width`* is given, characters are read, beginning with the first nonspace character, up to the first space character, or until the number of characters specified by the *`field width`* is reached (whichever comes first). The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating null character which is added automatically.

The `s` conversion character cannot be made to read a space character as part of a string. Space characters are always skipped over at the beginning of a string, and they terminate reading whenever they occur in the string. For example, suppose you want to read the first character from the following input line:

```
     "          Hello, there!"
```

(Ten spaces followed by "Hello, there!"; the double quotes are added for clarity). If you use `%c`, you get a space character. However, if you use `%1s`, you get "H" (the first nonspace character in the input).

The `[` conversion character also reads a character string from `stdin`. However, you should use this character when a string is not to be delimited by space characters. The left bracket is followed by a list of characters, and is terminated by a right bracket. If the first character after the left bracket is a circumflex (`^`), characters are read from `stdin` until a character is read which matches one of the characters between the brackets. If the first character is not a circumflex, characters are read from `stdin` until a character not occurring between the brackets is found. The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating null character which is added automatically. In some implementations, a minus sign (-) may specify a range of characters.

The three string conversion characters provide you with a complete set of string-reading capabilities. The `c` conversion character can be used to read any single character, or to read a character string when the exact number of characters in the string is known beforehand. The `s` conversion character enables you to read any character string which is delimited by space characters, and is of unknown length. Finally, the `[` conversion character enables you to read character strings that are delimited by characters other than space characters, and which are of unknown length.

Floating-Point Conversion Characters

The `e, f,` and `g` (or `E, F, and G`, respectively) conversion characters read characters from `stdin` until an inappropriate character is encountered, or until the number of characters specified by the *`field width`*, if given, is exhausted (whichever comes first).

The `e, f,` and `g` characters expect data in the following form: an optionally signed string of digits (possibly containing a decimal point), followed by an optional exponent field consisting of an `E` or `e` followed by an optionally signed integer. Thus, an inappropriate character is any character except `+, -, .,` `0` through `9, E,` or `e`.

These floating-point conversion characters may be preceded by a lowercase L (`l`), to indicate that a `double` value is expected rather than a `float`, or by an uppercase `L` (in ANSI C) to indicate that a long double value is expected rather than a `float`. The corresponding items in the item list for these conversion characters must be pointers to floating-point variables of the appropriate length.

Literal Characters

Any characters included in the format which are not part of a conversion specification are literal characters. A literal character is expected to occur in the input at exactly that point. Note that since the percent sign is used to introduce a conversion specification, you must type two percent signs (%%) to get a literal percent sign.

Suppose that you want to read the following line of data:

```
NAME: Joe Kool; AGE: 27; PROF: Elec Engr; SAL: 39550
```

To get the vital data, you must read two strings (containing spaces) and two integers. You also have data that should be ignored, such as the semicolons and the identifying strings

("NAME:"). To read the strings, first note that the identifying strings are always delimited by space characters. This suggests use of the `s` conversion character to read them. Second, you can never know the exact sizes of the NAME and PROF fields, but note that they are both terminated by a semicolon. Thus, you can use `[` to read them. Finally, the `d` conversion character can be used to read both integers. (Note: On 16-bit processors, you probably need to use a `long int` to read the salaries. Thus, `ld` should be used instead of `d`.)

The following code fragment successfully reads this data:

```
     char name[40], prof[40];
     int age;
     long int salary;
         .
         .
  scanf("%*s%*[ ]%[^;]%*c%*s%d%*c%*s%*[ ]%[^;]%*c%*s%ld",name,&age,
  prof,&salary);
```

For easier understanding, break the format into pieces:

`%*s`	This reads the string "NAME:". Since an asterisk is given the string is simply read and discarded.
`%*[ ]`	This removes all blanks occurring between "NAME:" and the employee's name. Note that this removes one or more blanks, giving the format some flexibility.
`%[^;]`	This reads all characters from the current character up to a semicolon, and assigns the characters to the array `name`.
`%*c`	This removes the semicolon left over after reading the name.
`%*s`	This reads the next identifying string, "AGE:", and discards it.
`%d`	This reads the integer age given, and assigns it to `age`. The semicolon after the age terminates `%d`, because that character is not appropriate for an integer value. Note that the address of `age` is given in the item list `()` instead of the variable name itself. If this is not done, a memory fault occurs at runtime due to the attempt of `scanf()` to use the parameter as a pointer.
`%*c`	This removes the semicolon following the age.
`%*s`	This reads the next identifying string, "PROF:", and discards it.
`%*[ ]`	This removes all blanks between "PROF:" and the next string.
`%[^;]`	This reads all characters up to the next semicolon, and assigns them to the character array `prof`.
`%*c`	This removes the semicolon following the profession string.
`%*s`	This reads the final identifying string, "SAL:", and discards it.
`%ld`	This reads the final integer and assigns it to the long integer variable `salary`. Again, note that the address of `salary` is given, not the variable name itself.

Although somewhat confusing to read, this format is quite flexible, because it allows for multiple spaces between items and varying identifying strings (that is, "PROFESSION:" could be specified instead of "PROF:"). The following `scanf()` call reads the same data, but is much less flexible:

```
  scanf("NAME: %[^;]; AGE:%d; PROF: %[^;]; SAL: %d",name,&age,prof,&salary);
```

In this example, literal characters are used to exactly match the characters in the input line. This only works if you can be sure that the data always appears in this form. However, if a typing variation is made, such as typing "SALARY:" instead of "SAL:", the `scanf()` fails.

`Scanf()` waits for more data as long as there are unsatisfied conversion specifications in the format. Thus, the `scanf()` call

```
  scanf("%f%f%f", &float1, &float2, &float3);
```

where `float1`, `float2`, and `float3` are all variables of type `float`, allows you to enter data in several ways. For example,

```
  14.77 29.8 13.0
```

is read correctly by `scanf()`, as is

```
 14.77
 29.8
 13.0
```

Using decimal points in floating-point data is recommended whenever floating-point variables are being read. However, `scanf()` converts integer data to floating-point if the conversion specification so demands. Thus, "13.0" in the previous example could have been entered as "13" with no side effects.

As a final example, consider the input string:

```
abcdef137 d14.77ghijklmnop
```

Suppose the following code fragment is used to read this string:

```
char arr1[10], arr2[10], arr3[10], arr4[10];
float float1;
scanf("%4c%[^3]%6c%f%[ghijkl]",arr1,arr2,arr3,&float1,arr4);
```

`%4c`	Reads four characters and assigns them to `arr1`. Thus, the string `abcd` is assigned to `arr1`. Note that a null character is not appended to the end of the string.
`%[ ^ 3]`	Reads all characters from the current character up to the character `3`. This assigns `ef1`, along with an added null character, to the array `arr2`.
`%6c`	Reads the next six characters and stores them in the array *`arr3`*. Thus, `37 d14` is assigned to `arr3`. A null character is not appended to the end of the string.
`%f`	Reads a floating-point value which, due to the lack of a field width, is terminated by the first `inappropriate` character. Thus, the value `.77` is assigned to `float1`.
`%[ghijkl]`	Reads all characters up to the first character not occurring between the brackets. This stores the string `ghijkl`, along with an appended null

character, in the array `arr4.`

Note that there are some characters left in `stdin` that were not read. Any characters left unread in the input remain there, which might cause unexpected errors.

Suppose that, later in the above program fragment, you want to read a string from `stdin` using `%s.` No matter what string you type in as input, it will never be read, because the `%s` conversion specification is satisfied by reading `"mnop"` (the characters left over from the previous read operation). To solve this, be sure you have read the entire current line of input before attempting to read the next. To fix this in the previous `scanf()` example, add a `%*s%*c` conversion specification at the end of the format (`%*s` reads characters up to the next newline character, and `%*c` reads the newline). This reads and discards the excess characters.

You can use a minus character (-) between characters in the match list inside a `[` conversion specifier to indicate a range of characters. For example, the conversion specifier `[A-Z]` matches all the characters from A through Z

See Also

`fscanf()`, `sscanf()`, `getc()`, `setlocale()`, `printf()`, `strtod()`, `strtol()`, ANSI C 4.9.6.4, POSIX.1 8.1

setbuf

Assigns a buffer to an open stream.

Syntax

```
#include <stdio.h>
void setbuf (FILE *stream, char *buffer);
```

Parameters

stream	A pointer to an open stream.
buffer	Either a pointer to a character array for buffered I/O, or null for unbuffered I/O.

Return Values

None.

Description

Normally, a standard I/O buffer is obtained through a call to `malloc()` on the first call to `getc()` or `putc()` (which all I/O functions eventually call). The standard I/O system normally buffers I/O in a buffer which is `BUFSIZ` bytes long. Exceptions are `stdout`, which, when directed to a terminal, is line buffered, and `stderr`, which is normally unbuffered.

NOTE Using an automatic array as a standard I/O buffer can be dangerous. Automatic variables are only defined in the code block in which they are declared. Thus, buffering which relies on an automatic array is only in effect during the current code block (main program or function). If you pass a file pointer to another function, and the stream pointed to by that file pointer is buffered using an automatic array, memory faults or other errors can occur. Therefore, if you use an automatic array for stream buffering, the stream should be used and closed only in the code block containing the array declaration. To avoid this restriction, use global or static arrays for buffering:

```
char buffer[BUFSIZ];
   …
main()
{
   …
  setbuf (fp, buffer);
}
```

The `setbuf` function enables you to change the buffer used for all standard I/O functions. The following example of a code fragment causes the array *buffer* to be used for buffering:

```
      …
```

```
FILE *fp;
char buffer[BUFSIZ];

fp = fopen(argv[1], "r");
   …
setbuf(fp, buffer);
   …
```

This fragment shows the correct order of events. First, the file is opened, and the buffering is assigned using `setbuf()`. From that point on any input taken from `fp` is buffered through the array *`buffer`*. Buffering can be eliminated altogether by specifying the null pointer in place of the buffer name, as in

```
setbuf(fp, NULL);
```

This causes input or output using `fp` to be completely unbuffered.

The `setbuf` function is limited to buffer sizes of either BUFSIZ bytes or zero. `setbuf()` assumes that the character array pointed to by *`buffer`* is BUFSIZ bytes. Passing `setbuf()` a (non-null) pointer to a smaller array can cause severe problems during operation because the standard I/O functions might overwrite memory following the end of the buffer.

See Also

`setvbuf()`, `fopen()`, `getc()`, `malloc()`, `putc()`, ANSI C 4.9.5.5, POSIX.1 8.1

setjmp

Saves the current environment.

Syntax

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

Parameters

env	An array of unsigned integers as defined by the type `jmp_buf`.

Return Values

0	Successful completion of `setjmp()`.
≠0	Returned as a result of a call to `longjmp()`. The value returned is the value passed in the *val* parameter of `longjmp()`.

Description

The `setjmp` macro creates an entry point in your program that can be reached with `longjmp()`.

The `setjmp` macro saves the current environment of the calling process in the *env* parameter. The parameter *env* is of type `jmp_buf`, defined in `<setjmp.h>`. It is an array of unsigned integers and therefore the *env* argument does not require an & operator.

A subsequent call to `longjmp()` requires that the *env* variable initialized by `setjmp()` be passed as a parameter. This allows `longjmp()` to restore the program environment saved by `setjmp()` and to continue program execution just after the `setjmp()` statement.

Upon successful completion, the `setjmp()` macro returns a zero value. A zero indicates that the return is from `setjmp()` itself and not a return as a result of a call to `longjmp()`.

If a nonzero value is returned, this indicates that the return is a result of a call to `longjmp()`. After the call to `longjmp()` is completed, the program executes as if the call to `setjmp()` (which stored information into the *env* argument) had returned a second time. The result of the second return from `setjmp()` is the return of the value of the nonzero *val* argument supplied to `longjmp()`.

See Also

`longjmp()`, ANSI C 4.6.1.1, POSIX.1 8.1

setkey

Defines the key used for encrypting blocks of text.

Syntax

```
void setkey (key)
   char *key;
```

Parameters

key	A pointer to a character string that contains the encryption key.

Return Values

None.

Description

The `setkey` function provides primitive access to the hashing algorithm used by `crypt()`. It is used in conjunction with `encrypt` to first prime the machine and then encrypt a block of text.

The argument of `setkey` is an 8-byte character array treated as a 64-bit binary number. The string is divided into groups of 8 bits, and the low-order bit in each group is ignored. This gives a 56-bit key that is used to prime the NBS Data Encryption Standard encryption algorithm. This is the key that is used with the hashing algorithm to encrypt the string *`block`* with the `encrypt` function.

See Also

`crypt()`, `encrypt()`

setlocale

Controls locale-specific features of the library.

Syntax

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

Parameters

category	Specifies that only a certain aspect is to be set to that locale and the others are unchanged. This parameter can be set to any one of the following macros (defined in `<locale.h>`): • `LC_COLLATE` • `LC_CTYPE` • `LC_MONETARY` • `LC_NUMERIC` • `LC_TIME` • `LC_ALL`
locale	Typically the name of a supported language. German, for example, is a supported language.

Return Values

x	A pointer to a string that defines the locale.
`NULL`	The program's locale has not been changed. The request has failed.

Description

The `setlocale` function controls locale-specific features of the library.

The string returned by `setlocale` must not be altered and may be overwritten by subsequent calls to `setlocale()`.

Following is a description of the behavior of `setlocale()` under four different conditions:

- **When in Program Startup**

  — When a program is started, the locale of the program is the default, `C-locale`.

- **When Locale is Specified**

  — If `locale` is specified, the named category is set to that locale.

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

- **When Locale is Not Specified**

— If `locale` is not specified (an empty string is used), as in the following example,

```
setlocale (category,"")
```

the locale is set according to the following scheme:

1. If an equivalent `'LC_'` environment variable is set, the language is set to the language specified by the variable.
2. If the environment variable `LANG` is set, the language is set as specified by `LANG`.
3. If the JCW `NLUSERLANG` or `NLDATALANG` is set:

   — For `LC_COLLATE` and `LC_TYPE`, the language is set as specified by `NLDATALANG`.

   — For `LC_TIME`, `LC_MONETARY` and `LC_NUMERIC`, the language is set as specified by `NLUSERLANG`.

   — Otherwise, `locale` is set to be the `C-locale`.

- **When Locale is Null**

  — If the `locale` is a null pointer, the `setlocale` function returns the current locale of the named category to the program. For example,

  ```
  setlocale (LC_MONETARY,NULL);
  ```

  returns to the program the locale that is set for monetary processing. This is a query operation that does not change the locale environment of the program.

NOTE The default locale is always the `C-locale`.

See Also

`localeconv()`, ANSI C 4.4.1.1, POSIX.1 8.1.2

setvbuf

Assigns a buffer and buffering method to an open stream.

Syntax

```
#include <stdio.h>
int setvbuf (FILE *stream, char *buffer,
             int type, size_t size);
```

Parameters

stream	A pointer to an open stream.
buffer	Either a pointer to a character array for buffered I/O, or null.
type	The method of buffering.
size	The size of the buffer.

Return Values

0	Success.
≠0	An error occurred.

Description

The `setvbuf` function enables you to assign a character array for buffering, and also provides the means to specify the size of the buffer to be used (*size*) and the type of buffering to be done (*type*). Acceptable values for *type* (defined in `<stdio.h>`) include:

_IOFBF	Input/output is fully buffered.
_IOLBF	Output is line buffered. The buffer is flushed each time a new line is written, the buffer is full, or input is requested.
_IONBF	Input/output is completely unbuffered.

If *type* _IONBF is specified, *stream* is totally unbuffered. Because no buffer is needed, values for *buffer* and *size* are ignored.

If *buffer* is the null pointer and type is specified as _IOFBF or _IOLBF, `setvbuf()` automatically allocates a buffer of *size* bytes through a call to `malloc()`.

If *size* is zero, a buffer of size BUFSIZ is used. This behavior can be used to change the buffer size for a stream even if you still want the standard I/O system to automatically allocate the buffer. This is particularly useful when a buffer larger than the specified BUFSIZ is needed.

Examples

In the following examples, the following two calls, though different, are functionally identical:

```
   setvbuf(fp, NULL, _IONBF, 0)
   setbuf(fp, NULL)
```

When *type* is _IOFBF or _IOLBF, buffering for *stream* is determined by *buffer* and *size.* If *buffer* is not the null pointer, it must point to a character array of *size* bytes. All buffering of *stream* is then handled through this array.

```
  …
FILE *fp;
char buffer [256]
char *filename;
int retcode;
fp=fopen(filename, "w");
retcode=setvbuf(fp, buffer, _IOFBF, 256);
if (retcode !=0) error ();
```

This fragment causes stream fp to be buffered through the 256-byte array *buffer.* Serious run-time errors can occur if the buffer array is not the size specified in the call to setvbuf() (here 256 bytes). As with setbuf, it is dangerous to use an automatic array for the buffer. Note that the return value of setvbuf() can be used to verify that the request was completed successfully.

```
  …
FILE * fp;
char * filename;
int retcode;
  …
fp = fopen(filename, "rt")
retcode=setvbuf(fp, NULL, _IOFBF, 2048);
if(retcode !=0) error( );
```

This fragment buffers stream fp through a 2048-byte buffer that is allocated by the system.

See Also

setbuf(), ANSI C 4.9.5.6

signal

Specifies how a signal is to be handled.

Syntax

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);
```

Parameters

sig	A signal number.
func	A pointer to the function that performs the exception handling.

Return Values

x	If successful, the most recent value of *func*.
SIG_ERR	An error occurred; `errno` is set to a positive value.

Description

The `signal` function defines the actions to take when the specified signal is raised. The action can be one of the following:

- Take the default action of terminating the program with some message.
- Ignore the signal.
- Invoke the user-defined signal handling function.

The `signal` function accepts two arguments: a signal number ("sig"), and a second argument (of type pointer to function accepting an int) that defines an action to take when a signal is raised.

You can define your own signal numbers in addition to using any of the predefined signal names listed below:

SIGABRT	Abnormal termination, (for example, by the `abort` function).
SIGFPE	An erroneous arithmetic operation, (for example, divide by 0).
SIGILL	An illegal instruction was executed (possibly after a jump).
SIGINT	An interactive interrupt signal was received.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request was sent to the program.

The second parameter, *func*, is a pointer to a function accepting an `int`. The *func* parameter defines the action to be taken upon receipt of the signal specified in *sig*.

In addition to passing the name of a user signal handler, you can use the following predefined macro values as the *func* parameter. The macros expand to constant expressions that have a type compatible with *func*, and whose values compare unequal with any declarable function. These macros cause `signal()` to behave as follows:

`SIG_DFL`	The default handling for that signal occurs. Usually this default action is to terminate the program with some message.
`SIG_IGN`	The signal is ignored when it is raised.

If the value is anything else, it is taken to be the address of a function that is called when the corresponding signal is raised. If *func* points to a function when a signal is raised, the following actions occur:

1. The equivalent of `signal (sig, SIG_IGN);` is performed, to prevent an infinite loop of signal handler calls if the same signal is raised again while it is being handled.
2. The function is called as follows: `(*`*func*`)(sig);`. The user function may terminate using `return;`, or by calling `abort()`, `exit()`, or `longjmp()`.

If a computational exception such as `SIGFPE` or `SIGSEGV` is raised, it is inadvisable to return normally (because the same instruction will very likely be executed again). One alternative is to use `setjmp()` at an early stage of the program, when it is in a known state, and jump to that place using `longjmp()` when a signal occurs. Another alternative is to exit the program by calling `exit()` or `abort()`.

If you choose to continue with program execution by returning normally or executing `longjmp()`, remember to first re-arm the signal handler (by calling `signal()`), so that subsequent occurrences of the signal may be caught.

NOTE Signals are provided for conformance with ANSI C. However, the only way to generate a signal on MPE/iX is by an explicit call to the `raise` function. For information on a more comprehensive facility for handling exceptions, see the *Trap Handling Programmer's Guide*.

Example

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

jmp_buf state0;   /* to hold a known state */

main()
{
   void goodbye (int);             /* trap handlers */
   void segv_handler (int);

   signal (SIGINT, SIG_IGN);     /* ignore interrupts */
   signal (SIGTERM, goodbye);
   signal (SIGSEGV, segv_handler);

   if (setjmp (state0) == 0) {
 /* body */
 printf ("about to raise SIGSEGV\n");
```

```
   raise(SIGSEGV);
      printf ("did not raise SIGSEGV\n");
    } else {
 printf ("longjmp'ed back\n");
    }

    printf ("about to raise SIGTERM\n");
    if (raise (SIGTERM))
 printf ("could not raise SIGTERM\n");
    /* else it should not get here.. */
}

void segv_handler (int sig)
{
    printf ("Raised SIGSEGV: In handler\n");
    longjmp (state0, 100); /* leap back */
}

void goodbye (int sig)
{
    printf ("Raised SIGTERM: In handler\n");
    exit (0);
}
```

See Also

`raise()`, `exit()`, `abort()`, ANSI C 4.7.1.1

sin

Computes a sine value.

Syntax

```
#include <math.h>
double sin (double x);
```

Parameters

x	A real number measured in radians.

Return Value

n	The sine of *x* measured in radians.
0	Indicates a complete loss of significance for large values of *x*. A `TLOSS` error message is printed on the standard error output; `errno` is set to `ERANGE`.

Description

The `sin` function loses accuracy when its argument is far from zero. For arguments causing partial loss of significance, a `PLOSS` error is generated but no message is printed and `errno` is set to `ERANGE`.

Error-handling can be changed by a user-written `matherr` function.

See Also

`cos()`, `tan()`, ANSI C 4.5.2.6, POSIX.1 8.1

sinh

Calculates the hyperbolic sine of an angle.

Syntax

```
#include <math.h>
double sinh (double x);
```

Parameters

x	A real number.

Return Values

n	The hyperbolic sine of the given angle.
±HUGE_VAL	An overflow condition has occurred for large absolute values of *x*; `errno` is set to `ERANGE`.

Description

Error-handling can be changed by a user-written `matherr` function.

See Also

`cosh()`, `tanh()`, `sin()`, `matherr()`, ANSI C 4.5.3.2, POSIX.1 8.1

sleep

Suspends program execution for an interval.

Syntax

```
unsigned long sleep (unsigned long seconds);
```

Parameters

seconds	The number of seconds to suspend program execution.

Return Values

x	The difference between the requested sleep time and the actual sleep time.

Description

The current process is suspended from execution for the number of seconds specified by the argument.

The suspension time can be longer than requested by an arbitrary amount due to the scheduling of other activity in the system.

The *seconds* parameter must be less than 2,147,485.

The `sleep` function returns the difference of the requested sleep time and the actual sleep time if the actual sleep time is less than the requested sleep time.

NOTE If linking with the POSIX/iX library, refer to the description of `sleep()` located in the *MPE/iX Developer's Kit Reference Manual*.

sprintf

Writes formatted data to a character string in memory.

Syntax

```
#include <stdio.h>
int sprintf (char *string, const char *format
             [,item [,item]...]);
```

Parameters

string	A pointer to a buffer in memory where the data is to be written.
format	Pointer to a character string defining the format (or the character string itself enclosed in double quotes).
item,...	Each *item* is a variable or expression specifying the data to write. Refer below to descriptions of conversion specifications and characters.

Return Values

≥0	If successful, the number of characters written, not counting the terminating null character.
<0	An error occurred.

Description

The `sprintf` function enables you to write data to a buffer in formatted form. The *string* parameter is a buffer in memory where the data is written. The *format* parameter is a pointer to a character string (or the character string itself enclosed in double quotes) which specifies the format and content of the data to be written. Each *item* is a variable or expression specifying the data to write.

The only difference between `sprintf()` and `printf()` is that `sprintf()` writes data into a character array, while `printf()` writes data to `stdout`, the standard output device.

The `sprintf()` format is made up of conversion specifications and literal characters. Literal characters are all characters that are not part of a conversion specification. Literal characters are written to *string* exactly as they appear in the format.

The `sprintf` function returns the number of characters written or a negative value (if an error is returned).

Conversion Specifications

The following list shows the different components of a conversion specification in their correct sequence:

1. A percent sign (`%`), which signals the beginning of a conversion specification; to output a literal percent sign, you must type two percent signs (`%%`).

2. Zero or more flags, which affect the way a value is written (see below).
3. An optional decimal digit string which specifies a minimum `field width`.
4. An optional *`precision`* consisting of a dot (`.`) followed by a decimal digit string.
5. An optional `l`, `h`, or `L` indicating that the argument is of an alternate type. When used in conjunction with an integer conversion character, an `l` or `h` indicates a long or short integer argument, respectively. When used in conjunction with a floating-point conversion character, an `L` indicates a long double argument.
6. A conversion character, which indicates the type of data to be converted and written.

A one-to-one correlation must exist between each specification encountered and each item in the item list.

The available *`flags`* are:

`-`	Causes the data to be left-justified within its output field. Normally, the data is right-justified.
`+`	Causes all signed data to begin with a sign (`+` or `-`). Normally, only negative values have signs.
`blank`	Causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the `blank` and `+` flags both appear, the `blank` flag is ignored.
`#`	Causes the data to be written in an alternate form. Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.
`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 is ignored. The `0` flag is also ignored for `d`, `i`, `o`, `u`, `x`, and `X` conversions if a precision is specified.

A *`field width`*, if specified, determines the minimum number of spaces allocated to the output field for the particular piece of data being written. If the data happens to be smaller than the field width, the data is blank- padded on the left (or on the right, if the `-` flag is specified) to fill the field. If the data is larger than the *`field width`*, the *`field width`* is simply expanded to accommodate the data. An insufficient *`field width`* never causes data to be truncated. If no *`field width`* is specified, the resulting field is made just large enough to hold the data.

The *`precision`* is a value which means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

NOTE A *`field width`* or *`precision`* may be replaced by an asterisk (`*`). If so, the next item in the item list is fetched, and its value is used as the *`field width`* or *`precision`*. The *`item`* fetched must be an integer.

Conversion Characters

Conversion characters specify the type of data to expect in the item list and cause the data to be formatted and written appropriately. The integer conversion characters are:

`d`, `i`	An integer *`item`* is converted to signed decimal. The *`precision`*, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the *`precision`*, the value is expanded with leading zeros. The default *`precision`* is 1. A null string results if a zero value is written with a zero *`precision`*. The # flag has no effect.
`u`	An integer *`item`* is converted to unsigned decimal. The effects of the *`precision`* and the # flag are the same as for `d`.
`o`	An integer *`item`* is converted to unsigned octal. The # flag, if specified, causes the *`precision`* to be expanded, and the octal value is written with a leading zero (a C convention). The *`precision`* behaves the same as in `d` above, except that writing a zero value with a zero *`precision`* results in only the leading zero being written, if the # flag is specified.
`x`	An integer *`item`* is converted to hexadecimal. The letters `abcdef` are used in writing hexadecimal values. The # flag, if specified, causes the *`precision`* to be expanded, and the hexadecimal value is written with a leading `"0x"` (a C convention). The *`precision`* behaves as in `d` above, except that writing a zero value with a zero *`precision`* results in only the leading `"0x"` being written, if the # flag is specified.
`X`	Same as `x` above, except that the letters `ABCDEF` are used to write the hexadecimal value, and the # flag causes the value to be written with a leading `"0X"`.

The character conversion characters are as follows:

`c`	The character specified by the `char` *`item`* is written. The *`precision`* is meaningless, and the # flag has no effect.
`s`	The string pointed to by the character pointer *`item`* is written. If a *`precision`* is specified, characters from the string are written until the number of characters indicated by the *`precision`* is reached, or until a null character is encountered, whichever comes first. If the *`precision`* is omitted, all characters up to the first null character are written. The # flag has no effect.

The floating-point conversion characters are:

`f`	The `float` or `double` *`item`* is converted to decimal notation in style `f`; that is, in the form `[-]ddd.ddd` where the number of digits after the decimal point is equal to the *`precision`*. If no *`precision`* is specified, six digits are written after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. If the # flag is specified, a decimal point always appears, even if no digits follow the decimal point.

`e`	The `float` or `double` *`item`* is converted to scientific notation in style `e`; that is, in the form `[-]d.ddde±ddd` where there is always one digit before the decimal point. The number of digits after the decimal point is equal to the *`precision`*. If no *`precision`* is given, six digits are written after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. The exponent always contains exactly three digits. If the # flag is specified, the result always contains a decimal point, even if no digits follow the decimal point.
`E`	Same as `e` above, except that `E` is used to introduce the exponent instead of `e` (style `E`).
`g`	The `float` or `double` *`item`* is converted to either style `f` or style `e`, depending on the size of the exponent. If the exponent resulting from the conversion is less than -4 or greater than the *`precision`*, style `e` is used. Otherwise, style `f` is used. The *`precision`* specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the # flag is specified, the result always has a decimal point, even if no digits follow the decimal point, and trailing zeros are *not* removed.
`G`	Same as the `g` conversion above, except that style `E` is used instead of style `e.`

Other conversion characters are:

`p`	The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters.
`n`	The argument is a pointer to an integer into which is written the number of characters written to the output stream so far by this call to `fprintf()`. No argument is converted.
`%`	A `%` is written. No argument is converted. The complete conversion specification is &%&%.

The *`item`*s in the item list may be variable names or expressions. Note that, with the exception of the `s` conversion, pointers are not required in the item list. If the `s` conversion is used, a pointer to a character string must be specified.

Example

The following program formats data. A user enters data and that data is reformatted into a string, which is passed along to another program, such as a database maintainer. The string contains the data that the user entered, but in a form using strict field widths for the various pieces of data.

The database program might require these field widths to be processed correctly, and you need not require the user to enter the data with the field widths. Users can enter data in a convenient form without the fixed field restrictions imposed by the database.

```
#include <stdio.h>
main()
```

```
{
   char name[31], prof[31], hdate[7], curve[3], string[81];
   char *format = "%30s%2d%30s%6ld%6s%2d%2s";
   int age, rank;
   long salary;
/* start asking questions */
   printf("\nName (30 chars max): ");
   gets(name);
   while(name[0] != ']') {
      printf("Age: ");

      printf("Job title (30 chars max): ");
      gets(prof);
      printf("Salary (6 digits max, no comma): ");

      printf("Hire date (numerical MMDDYY): ");
      gets(hdate);


      printf("Pay curve: ");
      gets(curve);
/* format string */
      sprintf(string,format,name,age,prof,salary,hdate,rank,curve);
      printf("\n%s\n", string);
/* start next round */
      printf("\nName (30 chars max): ");
      gets(name);
   }
}
```

The program above asks questions about name, age, job title, salary, hire date, ranking, and pay curve. This data is then packed into a 78-character string using the `sprintf()` function. This program writes the string on your screen, but in an actual working environment, the string would probably be passed directly to the database program. Note that `sprintf()` function format is specified as an explicit character pointer. When lengthy, unchanging formats are used, this is more convenient than typing the entire format string, especially if the item list is long.

See Also

`setlocale()`, `putc()`, `scanf()`, ANSI C 4.9.6.5, POSIX.1 8.1

sprintmsg

Prints formatted output with numbered arguments to a character array.

Syntax

```
int sprintmsg (char *s, char *format [, arg] ...)
```

Parameters

s	A pointer to a character array where the output is directed
format	A pointer to the string containing the formatting information. It contains optional placeholders and formatting specifications where *arg1* thru *argn* are to be substituted.
arg1 ... argn	A character, character pointer, or integer value giving the parameter to be converted, formatted, and merged with *format* prior to output.

Return Values

x	The number of characters transmitted.
`EOF`	Indicates failure.

Description

This function is derived from `printf()`. In `sprintmsg()`, the conversion character % is replaced by the sequence `%n$`. The `n` character is a decimal digit in the range 1-9, and indicates that this conversion should be applied to the `n`th argument, rather than to the next unused one. All other aspects of formatting are unchanged. All conversion specifications must contain the `%n$` sequence, and you should make sure the numbering is correct. All parameters must be used exactly once. See `printf()` for more details on formatting and conversion specifications.

See Also

`fprintmsg()`, `printf()`, `printmsg()`

sqrt

Computes the square root of a number.

Syntax

```
#include <math.h>
double sqrt (double x);
```

Parameters

x	A real number.

Return Values

n	The square root of the real number.
0	The *x* parameter is negative; `errno` is set to `EDOM`.

Description

Error handling can be changed by a user-written `matherr` function.

See Also

`pow()`, `matherr()`, ANSI C 4.5.5.2, POSIX.1 8.1

srand

Sets a starting point for subsequent calls to `rand()`.

Syntax

```
#include <stdlib.h>
void srand (unsigned int seed);
```

Parameters

seed	A value that sets the starting point for subsequent calls to the `rand` function.

Return Values

None.

Description

The `srand` function causes subsequent calls to `rand()` to return a new random sequence based on the value passed in *seed*. The sequence of pseudo-random numbers that `rand()` produces is always the same for any given *seed*. Thus, given *seed n*, the random sequence is always the same.

If `srand()` is not used to initialize the random number generator to a particular starting point, `rand()` returns the same sequence of numbers as when `srand()` is first called with a seed value of one.

See Also

`rand()`, ANSI C 4.10.2.2, POSIX.1 8.1

sscanf

Reads formatted data from a character string in memory.

Syntax

```
#include <stdio.h>
int sscanf (const char *string, const char *format
            [,item [,item]...]);
```

Parameters

string	A pointer to a buffer in memory containing the formatted data to be read.
format	A pointer to a character string defining the format of the data to be read (or the character string itself enclosed in double quotes).
item	The address of a variable into which the data will be placed. Refer below for descriptions of conversion specifications.

Return Values

≥0	The number of successfully matched and assigned input items.
`EOF`	An error occurred on input (no input characters, or a matching error occurred before any conversion).

Description

The `sscanf` function reads externally formatted data from a buffer in memory, converts the data to internal format, and stores the results in a group of arguments. The format consists of white-space characters, conversion specifications, and literal characters.

The `sscanf` function returns the number of successfully matched and assigned input items or returns `EOF` if there are no input characters available or if a matching error occurred before any conversion was made.

This function behaves identically to the `scanf()` function except that `sscanf()` reads data from a character string instead of from `stdin`.

White-Space Characters

White-space characters (blanks, tabs, newlines, or form feeds) cause input to be read up to the next non-white-space character.

Conversion Specifications

A conversion specification is a character sequence that tells `sscanf()` how to interpret the data received at that point in the input.

In the format, a *conversion specification* is introduced by a percent sign (`%`), optionally followed by an asterisk (`*`) (called the *assignment suppression* character), optionally followed by an integer value (called the *field width*). The conversion specification is terminated by a character specifying the type of data to expect; the terminating characters are called *conversion characters*. The integer and floating-point conversion characters may be optionally preceded by a character indicating the size of the receiving variable.

When a conversion specification is encountered in a format, it is matched up with the corresponding item in the item list. The data formatted by that specification is then stored in the location pointed to by that item. For example, if there are four conversion specifications in a format, the first specification is matched up with the first item, the second specification with the second item, and so on.

The number of conversion specifications in the format is directly related to the number of items specified in the item list. With one exception, there must be at least as many items as there are conversion specifications in the format. If there are too few items in the item list, an error occurs; if there are too many items, the excess items are ignored. The one exception occurs when the assignment suppression character (`*`) is used. If an asterisk occurs immediately after the percent sign (before the *field width*, if any), the data formatted by that conversion specification is discarded. No corresponding item is expected in the item list; this is useful for skipping over unwanted data in the input.

Conversion Characters

There are 14 conversion characters: five format integer data, three format character data, three format floating-point data, and three special characters.

The integer conversion characters are:

`d`	A decimal integer is expected.
`i`	A signed integer is expected.
`o`	An octal integer is expected.
`u`	An unsigned decimal integer is expected.
`x`	A hexadecimal integer is expected.

The character conversion characters are:

`c`	A single character is expected, normal skip over leading white space is suppressed.
`s`	A character string is expected.
`[`	A character string is expected, normal skip over leading white space is suppressed.

The floating-point conversion characters are:

`e, f, g`	A floating-point number is expected. (The capitalized forms of these characters are also accepted.)

The special characters are:

`p`	Matches an implementation-defined set of sequences.

`n`	No input is consumed. The corresponding argument is a pointer to an integer into which is written the number of characters read from the input stream so far by this call to `fscanf()`.
`%`	Matches a single %. No conversion or assignment occurs. The complete conversion specification is &%&%

Integer Conversion Characters

The `d`, `o`, and `x` conversion characters read characters from *`string`* until an inappropriate character is encountered, or until the number of characters specified by the *`field width`*, if given, is exhausted (whichever comes first).

For `d`, an inappropriate character is any character except +, -, and 0 through 9. For `o`, an inappropriate character is any character except +, -, and 0 through 7. For `x`, an inappropriate character is any character except +, -, 0 through 9, and the characters `a` through `f` and `A` through `F`. Note that negative octal and hexadecimal values are stored in their twos complement form with sign extension. Thus, they might look unfamiliar if you print them out later using `printf()`.

These integer conversion characters can be preceded by a `l` to indicate that a `long int` should be expected rather than an `int`. They can also be preceded by `h` to indicate a `short int`. The corresponding items in the item list for these conversion characters must be pointers to integer variables of the appropriate length.

Character Conversion Characters

The `c` conversion character reads the next character from *`string`* no matter what that character is. The corresponding item in the item list must be a pointer to a character variable. If a *`field width`* is specified, the number of characters indicated by the *`field width`* are read. In this case, the corresponding item must refer to a character array large enough to hold the characters read.

Note that strings read using the `c` conversion character are not automatically terminated with a null character in the array. Because all C library functions that use strings assume the existence of a null terminator, be sure to add the `'\0'` character yourself. If you do not, library functions are not able to tell where the string ends, and you will get unexpected results.

The `s` conversion character reads a character string from *`string`*, which is delimited by one or more space characters (blanks, tabs, or newlines). If *`field width`* is not given, the input string consists of all characters from the first nonspace character up to (but not including) the first space character. Any initial space characters are skipped over. If a *`field width`* is given, characters are read, beginning with the first nonspace character, up to the first space character, or until the number of characters specified by the *`field width`* is reached (whichever comes first). The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating null character, which is added automatically.

The `s` conversion character cannot be made to read a space character as part of a string. Space characters are always skipped over at the beginning of a string, and they terminate reading whenever they occur in the string. For example, suppose you want to read the first character from the following input line:

```
"          Hello, there!"
```

(Ten spaces followed by "Hello, there!"; the double quotes are added for clarity). If you use `%c`, you get a space character. However, if you use `%1s`, you get "H" (the first nonspace character in the input).

The `[` conversion character also reads a character string from *`string`*. However, you should use this character when a string is not to be delimited by space characters. The left bracket is followed by a list of characters, and is terminated by a right bracket. If the first character after the left bracket is a circumflex (`^`), characters are read from *`string`* until a character is read that matches one of the characters between the brackets. If the first character is not a circumflex, characters are read from *`string`* until a character not occurring between the brackets is found. The corresponding item in the item list must refer to a character array large enough to hold the characters read, plus a terminating null character, which is added automatically. In some implementations, a minus sign (-) may specify a range of characters.

The three string conversion characters provide you with a complete set of string-reading capabilities. The `c` conversion character can be used to read any single character, or to read a character string when the exact number of characters in the string is known beforehand. The `s` conversion character enables you to read any character string that is delimited by space characters and is of unknown length. Finally, the `[` conversion character enables you to read character strings that are delimited by characters other than space characters, and which are of unknown length.

Floating-Point Conversion Characters

The `e, f,` and `g` (or `E, F, and G`, respectively) conversion characters read characters from *`string`* until an inappropriate character is encountered, or until the number of characters specified by the *`field width`*, if given, is exhausted (whichever comes first).

The `e, f,` and `g` characters expect data in the following form: an optionally signed string of digits (possibly containing a decimal point), followed by an optional exponent field consisting of an `E` or `e` followed by an optionally signed integer. Thus, an inappropriate character is any character except `+, -, .,` `0` through `9, E,` or `e.`

These floating-point conversion characters may be preceded by a lowercase `L` (`l`), to indicate that a `double` value is expected rather than a `float`, or by an uppercase `L` (in ANSI C) to indicate that a long double value is expected rather than a `float`. The corresponding items in the item list for these conversion characters must be pointers to floating-point variables of the appropriate length.

Literal Characters

Any characters included in the format that are not part of a conversion specification are literal characters. A literal character is expected to occur in the input at exactly that point. Note that since the percent sign is used to introduce a conversion specification, you must type two percent signs (%%) to get a literal percent sign.

Examples

The following program reads a string from `stdin`, stores the string in the character array `string`, and prints the first word of the string.

```
#include <stdio.h>
main()
{
   char string[80], word[25], *gets();

/* get the string */

   printf("Enter your string: ");
   gets(string);

/* get the first word */

   sscanf(string, "%s", word);
   printf("The first word is %s.\n", word);
}
```

The `sscanf()` function is often used to convert ASCII characters into other forms, such as integer or floating-point values. For example, the following program uses `sscanf()` to implement a five-function calculator:

```
#include <stdio.h>
main()
{
   char line[80], *gets(), op[4];
   long n1, n2;
   double arg1, arg2;

/* print prompt (>) and get input */

   printf("\n> ");
   gets(line);

/* begin loop */

      while(line[0] != 'q') {
         sscanf(line, "%*s%s", op);
         if(op[0] == '+') {

            printf("Answer:  %g\n\n", arg1+arg2);
      } else if(op[0] == '-') {

         printf("Answer:  %g\n\n", arg1-arg2);
      } else if(op[0] == '*') {

         printf("Answer:  %g\n\n", arg1*arg2);
      } else if(op[0] == '/') {

         printf("Answer:  %g\n\n", arg1/arg2);
      } else if(op[0] == '%') {
            while(n1 >= n2)
               n1 -= n2;
            printf("Answer:  %ld\n\n", n1);
         } else
            printf("Can't recognize operator: %s\n\n", op);
         printf("> ");
         gets(line);
      }
   }
```

The calculator program above accepts input lines having the form

`value operator value`

where *`value`* is any number, and *`operator`* is the symbol `+`, `-`, `*`, `/`, or `%`, representing addition, subtraction, multiplication, division, or remainder, respectively. All functions, except for the remainder function, are performed in floating-point values; values for these functions can be entered with or without a decimal point. Values for the remainder function must not have a decimal point. There must be at least one space between each value and the operator.

Note that in this program, the entire input line is read using `gets()`. Then, the different parts of the input line are read from `line` using `sscanf()`. The input line is stored as an ASCII string in `line`, but portions are converted to floating-point or integer values, depending on the operator.

Examples of valid entries are

```
15.778 * 3.89
27 % 8
17 + 39.72
```

The program terminates when it reads a line beginning with the letter "q", such as "quit".

There are two differences between reading data from `stdin` and reading data from a string. First, reading data from `stdin` causes that data to no longer remain in `stdin`. This is not true for a string. Because the data is stored in a string, the data remains in memory, even if that data has been read several times. Second, because the data read from `stdin` disappears as you read it, the next read operation from `stdin` always begins when the previous read operation is terminated. This is not true when you read from a string using `sscanf()`.

Each successive read operation starts at the beginning of the string. Thus, if you want to read five words from a string stored in a character array, you must read the words in a single `sscanf()` call. If you try to read one word in five separate `sscanf()` calls, each call starts reading at the beginning of the string and so you read the same word five times.

See Also

`getc()`, `setlocale()`, `printf()`, `strtod()`, `strtol()`, ANSI C 4.9.6.6, POSIX.1 8.1

strcat

Appends one string to another.

Syntax

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Parameters

s1 and *s2*	Character pointers to null-terminated character strings.

Return Values

x	The value of *s1*.

Description

The `strcat` function appends the entire string pointed to by *s2* including the terminating null character onto the end of string *s1*. The first character of *s2* overwrites the terminating null character of *s1*.

Example

Refer to the example in the `strcpy()` description.

See Also

`strcpy()`, `strncat()`, `strncpy()`, ANSI C 4.11.3.2, POSIX.1 8.1

strchr

Locates the first occurrence of a specified character within a string.

Syntax

```
#include <string.h.>
char *strchr(const char *s, int c);
```

Parameters

s	A pointer to a null-terminated character string.
c	The value to find in the target string.

Return Values

x	A character pointer to the first occurrence of *c* in string *s*.
`NULL`	The character is not found.

Description

The `strchr` function searches the null-terminated string *s* for the first occurrence of *c* converted to type `char`.) The terminating null character is part of the string.

Example

The following example replaces all occurrences of @ in the array `string` with #:

```
char *ptr, *strchr(), string[100];
     …
while((ptr = strchr(string, '@') != NULL)
   *ptr = '#';
```

See Also

`strrchr()`, `strpbrk()`, ANSI C 4.11.5.2, POSIX.1 8.1

strcmp

Compares two strings and returns an integer indicating the result of the comparison.

Syntax

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Parameters

s1, *s2*	Pointers to character strings.

Return Values

< 0	The *s1* parameter is less than *s2*
= 0	The *s1* and *s2* parameters are equal.
> 0	The *s1* parameter is greater than *s2*.

Description

In the `strcmp` function, *s1* and *s2* are character pointers to the null-terminated character strings to be compared. This function compares the entire strings, stopping as soon as the result is determined or when a null character is encountered.

See Also

`strncmp()`, `strcoll()`, `memcmp()`, ANSI C 4.11.4.2, POSIX.1 8.1

strcoll

Compares two strings, interpreting them as appropriate for the current locale. This function is generally used in conjunction with Native Language Support (NLS).

Syntax

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Parameters

s1	A pointer to the first string.
s2	A pointer to the second string.

Return Values

> 0	The *s1* parameter is greater than *s2*.
= 0	The *s1* parameter is equal to *s2*.
< 0	The *s1* parameter is less than *s2*.

Description

The `strcoll` function returns an integer greater than, equal to, or less than zero, indicating that the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*. Both strings are interpreted as appropriate to the `LC_COLLATE` category of the current locale.

See Also

`strcmp()`, `strncmp()`, `memcmp()`, ANSI C 4.11.4.3

strcpy

Copies the contents of *s2* into *s1*.

Syntax

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

Parameters

s1	A pointer to the target string.
s2	A pointer to the source string.

Return Values

x	The value of *s1*.

Description

In the `strcpy` function, *s2* is a character pointer to the string to be copied, and *s1* is a character pointer to the beginning of the string into which the contents of string *s2* are copied. The `strcpy` function copies the entire string, up to and including the first null encountered. Use `strlen()` and `memmove()` rather than `strcpy()` if the string at *s2* overlaps *s1*; otherwise, the behavior is undefined.

Examples

The following program uses the `strcpy` function and the `strcat` function to build a character string representing the lowercase alphabet, one character at a time:

```
#include <stdio.h>
main()
{
   int b = 'b', z = 'z', i;
   char alpha[30], chr[4];

   chr[1] = NULL;
   strcpy(alpha, "a");
   printf("%s\n", alpha);

   for(i = b; i <= z; i) {
        chr[0] = i;
        strcat(alpha, chr);
        printf("%s\n", alpha);
   }
}
```

The array `chr` is always going to be a two-character array consisting of the next character in the alphabet followed by a null character. Thus, the second element of `chr` is set to a null character early in the program. The first `chr` element is then successively set to the next lowercase character in the `for` loop, and the resulting two-character string is concatenated onto the end of the alphabet assembled so far in `alpha`. Note the use of `strcpy()` to initialize `alpha`. Remember that C transforms one or more characters enclosed in double quotation marks into a character pointer to those characters followed by a null character. Thus, the `strcpy()` statement above copies the character 'a' followed by a null character into `alpha`.

There are some things to be aware of when using `strcpy()`, `strncpy()`, `strcat()`, and `strncat()`. These functions all modify string *`s1`* in some way, but none of them check for overflow in that string. Therefore, be sure there is enough room in *`s1`* to hold the added or copied characters plus a terminating null character. Also, be sure you use a character *`array`* for *`s1`* (not just a character pointer), especially when using `strcat` or `strncat`. This is because an explicitly declared array can have sufficient memory allocated to it to contain all of its elements, but a character pointer simply points to a single location in memory. Concatenating a string to the end of a string contained in an array is guaranteed to work if the array is large enough. However, concatenating a string to a string of characters referenced by a simple character pointer is dangerous, because the concatenated characters could overwrite data in memory. For example,

```
char array[100], *ptr = "abcdef";
     …
strcat(array, ptr);
```

works fine, because you are guaranteed that 100 storage elements have been set aside for the array. However,

```
char *ptr1 = "abcdef", *ptr2 = "ghijkl";
     …
strcat(ptr1, ptr2);
```

will not work. Although C makes sure there is enough room for the initializing strings ("abcdef" and "ghijkl" in this example), there are no guarantees that there is enough room to add characters to the end of one of these strings. Therefore, the last fragment could easily overwrite valid data occurring after the string pointed to by `ptr1`.

Because string *`s2`* is not modified, you can use arrays or character pointers for this item with no ill effects.

See Also

`memmove()`, `strlen()`, `strcat()`, `strncat()`, `strncpy()`, ANSI C 4.11.2.3, POSIX.1 8.1

strcspn

Returns the length of the first substring in *s1* composed entirely of non-members of the character set *s2*.

Syntax

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Parameters

s1	A pointer to a character string to search.
s2	A pointer to a character string defining the character set.

Return Values

x	The length of the initial segment in *s1* formed by characters not in *s2*.

Description

The `strcspn` function sequentially processes each character in the array referenced by *s1*. For each character in the array, it scans *s2* looking for a match. If a match is *not* found, a counter is incremented and the function continues. If a match is found, the scanning stops.

Example

Given the following two strings:

```
'A tattletale never wins.'
```

for string *s1*, and

```
' -Aatle'
```

for *s2*. Executing

```
strcspn(s1, s2);
```

returns 0, because there is no initial segment of *s1* that contains characters not found in *s2*.

See Also

`strspn()`, ANSI C 4.11.5.3, POSIX.1 8.1

strerror

Maps an error number to a message string.

Syntax

```
#include <string.h>
char *strerror(int errnum);
```

Parameters

errnum	An error number.

Return Values

x	A pointer to the error message.
NULL	A null pointer is returned if there is no matching error message.

Description

The `strerror` function returns a pointer to an error message that corresponds with the specified error number. Do not modify the value of the error message referenced by the returned pointer.

See Also

ANSI C 4.11.6.2

strftime

Creates a formatted time string.

Syntax

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
                const char *format,
                const struct tm *timeptr);
```

Parameters

s	A pointer to a character array to which the function returns a formatted character string.
maxsize	The size of the character array.
format	A pointer to an array containing conversion specifications and ordinary multibyte characters to be inserted the string.
timeptr	A pointer to a `tm` variable in broken-down time format.

Return Values

x	A value of type `size_t` indicating the number of characters placed into the array pointed to by *s*.
0	The resulting formatted character string is greater than *maxsize*, and *s* is indeterminate.

Description

This function places characters into the array pointed to by *s*, which is controlled by the string pointed to by *format*. This string consists of zero or more conversion specifications and ordinary multibyte characters. A conversion specification consists of a % followed by a character that determines the conversion specification's behavior.

All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying occurs between objects that overlap, the behavior is undefined. No more than *maxsize* characters are placed into the array.

Each conversion specification is replaced by appropriate characters, as described below. These characters are determined by the program's locale as determined by `lc_time` and by the values contained in the structure pointed to by *timeptr*.

- %a is replaced by the locale's abbreviated weekday name.
- %A is replaced by the locale's full weekday name.
- %b is replaced by the locale's abbreviated month name.
- %B is replaced by the locale's full month name.

- `%c` is replaced by the locale's appropriate date and time representation.
- `%d` is replaced by the day of the month as a decimal number (01-31).
- `%e` is replaced by the day of the month as a decimal number (1-31 in a two-digit right-justified field with leading space> fill).
- `%H` is replaced by the hour (24-hour clock) as a decimal number (00-23).
- `%I` is replaced by the hour (12-hour clock) as a decimal number (01-12).
- `%j` is replaced by the day of the year as a decimal number (001-366).
- `%m` is replaced by the month as a decimal number (01-12).
- `%M` is replaced by the minute as a decimal number (00-59).
- `%p` is replaced by the locale's equivalent of either AM or PM.
- `%S` is replaced by the second as a decimal number (00-61).
- `%U` is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
- `%w` is replaced by the weekday as a decimal number [0 (Sunday)-6].
- `%W` is replaced by the week number of the year (Monday as the first day of week 1) as decimal number (00-53).
- `%x` is replaced by the locale's appropriate date representation.
- `%X` is replaced by the locale's appropriate time representation.
- `%y` is replaced by the year without century as a decimal number (00-99).
- `%Y` is replaced by the year with century as a decimal number.
- `%Z` is replaced by the time zone name, or by no character if no time zone is determinable.
- `%` is replaced by `%`.

The behavior is undefined for any conversion specification not described above.

If the total number of resulting characters (including the terminating null character) is not more than *`maxsize`*, `strftime` returns the number of characters placed into the array pointed to by *`s`* (not including the terminating null character). Otherwise, zero is returned and the array contents are indeterminate.

See Also

ANSI C 4.12.3.5, POSIX.1 8.1

strlen

Computes the length of the string pointed to by *s*.

Syntax

```
#include <string.h>
size_t strlen(const char *s);
```

Parameters

s	A pointer to the character string.

Return Values

x	The length of the string specified as an unsigned integer.

Description

In the `strlen` function, *s* is a character pointer to the null-terminated string to be scanned. `strlen()` counts up to, but not including, the terminating null character. The length of a string containing only a null character is zero.

NOTE Since the value returned by `strlen()` is an unsigned number, care must be taken when performing subtraction operations on the value returned by the function. The following code fragment is incorrect:

```
    …
for (i=0;i<=strlen(string)-1;i++
    …
```

The string length of a null string minus one (`strlen(string)-1`) is a very large positive number, not minus one (-1) as was intended. Thus, the code above does not bypass the `for` loop, but erroneously executes the loop a large number of times.

Example

```
len = strlen(string);
```

The integer `len` contains the total number of non-null characters in the string pointed to by `string`. Thus,

`string[len]`

is the terminating null in `string`.

See Also

ANSI C 4.11.6.3, POSIX.1 8.1

strncat

Appends a copy of string 2 to string 1.

Syntax

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

Parameters

s1	A pointer to a destination.
s2	A pointer to a null-terminated source character string.
n	The maximum number of characters to concatenate from *s2* to *s1*, unless `strncat()` first encounters a null terminator in *s2*.

Return Values

x	The address of *s1*.

Description

In the `strncat` function, *s1* and *s2* are character pointers to null-terminated character strings.

The `strncat` function is similar to `strcat()`, except that at most *n* characters are appended to *s1* (or up to a null character, whichever comes first). Note that string *s2* need not be null-terminated when using `strncat()` if *n* is less than or equal to the length of *s2.* This function returns a character pointer to the null-terminated result. The first character of *s2* overwrites the null terminator of *s1*. A terminating null is appended to the result.

This function does not check if there is room in *s1* for the additional characters of *s2.* Thus, to be safe, *s1* should *always* be a declared array having plenty of space for the additional characters of *s2,* plus a terminating null character.

See Also

`strcat()`, `strcpy()`, `strncpy()`, ANSI C 4.11.3.2, POSIX.1 8.1

strncmp

Compares two strings up to a maximum of *n* characters and returns an integer result of the comparison.

Syntax

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Parameters

s1, *s2*	Pointers to character strings.
n	Specifies the maximum number of characters to compare in both *s1* and *s2*.

Return Values

<0	*s1* is less than *s2*.
=0	*s1* and *s2* are equal.
>0	*s1* is greater than *s2*.

Description

In the `strncmp` function, *s1* and *s2* are character pointers to the null-terminated character strings to be compared. `strncmp()` compares at most *n* characters of both strings. Neither string need be null-terminated if *n* is less than or equal to the length of the shorter string.

Example

The following program fragment uses `strncmp()` to analyze the contents of a file coded with macros. In this example, a period (`.`) at the beginning of a line indicates a macro. The program reads each line of the file and keeps a count of the number of times selected macros are used, and prints a summary of its findings at the end.

```
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[ ];
{
   char *fgets(), line[100];
   FILE *fp;
   int nsh, npp, ntp, nrs, nre, npd, nip, nmisc, nlines;

   nsh = npp = ntp = nrs = nre = npd = nip = nmisc = nlines = 0;

   if(argc != 2) {
        fprintf(stderr, "Usage: count file\n");
```

```
               exit(2);
          }

          fp = fopen(argv[1], "r");
          if(fp == NULL) {
               fprintf(stderr, "Can't open %s.\n", argv[1]);
               exit(1);
          }

          while(fgets(line, 100, fp) != NULL) {
               if(strncmp(line, ".SH", 3) == 0)

               else if(strncmp(line, ".PP", 3) == 0)

               else if(strncmp(line, ".TP", 3) == 0)

               else if(strncmp(line, ".RS", 3) == 0)

               else if(strncmp(line, ".RE", 3) == 0)

               else if(strncmp(line, ".PD", 3) == 0)

               else if(strncmp(line, ".IP", 3) == 0)

               else if(line[0] == '.')


          }


          printf("No. of .SH's: %d\n", nsh);
          printf("No. of .PP's: %d\n", npp);
          printf("No. of .TP's: %d\n", ntp);
          printf("No. of .RS's: %d\n", nrs);
          printf("No. of .RE's: %d\n", nre);
          printf("No. of .PD's: %d\n", npd);
          printf("No. of .IP's: %d\n", nip);
          printf("No. of misc. macros: %d\n", nmisc);

          fclose(fp);
          exit(0);
     }
```

In the program above, `strncmp()` compares the first three characters of each line read. If the first three characters match a particular macro, the appropriate counter is incremented. If the line begins with "`.`", but is not one of the macros being searched for, the "miscellaneous" counter is incremented. The total number of lines in the file is also given.

See Also

`strcmp()`, `strcoll()`, `memcmp()`, ANSI C 4.11.4.4, POSIX.1 8.1

strncpy

Copies all or part of *s2* into *s1*.

Syntax

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Parameters

s1	A pointer to a destination.
s2	A pointer to a null-terminated source character string.
n	The maximum number of characters to copy from *s2* to *s1*.

Return Values

x	The address of *s1*.

Description

In the `strncpy` function, *s2* is a character pointer to the string to be copied, and *s1* is a character pointer to the beginning of the string into which the contents of string *s2* are copied. The `strncpy` function copies up to *n* characters, or up to and including the first encountered null character, whichever occurs first. If `strncpy()` encounters a null before copying *n* characters, it pads the string *s1* with nulls up to *n* characters. String *s2* does not have to be null-terminated when using `strncpy()` if *n* is less than or equal to the length of *s2*. If there is no null character in the first *n* characters of *s2*, the result is not null-terminated. If the strings overlap, behavior is undefined.

See Also

`memcpy()`, `memmove()`, `strcat()`, `strcpy()`, `strlen()`, `strncat()`, ANSI C 4.11.2.4, POSIX.1 8.1

strpbrk

Returns a pointer to the location in *s1* of the first occurrence of any member of the character set *s2*.

Syntax

```
#include <string.h.>
char *strpbrk(const char *s1, const char *s2);
```

Parameters

s1	A pointer to a null-terminated character string to be searched.
s2	A pointer to a null-terminated character string containing a character set.

Return Values

x	A character pointer to the first occurrence of a character from *s2* in string *s1*.
`NULL`	A character from the character set *s2* is not found.

Description

The `strpbrk` function scans null-terminated string *s1*, stopping when it finds a character from the supplied character set *s2*.

Example

This example uses `strpbrk()` to parse embedded numerical data from user-supplied input data. For simplicity, assume that the following conventions are used:

- Positive numbers do not begin with +;
- Fractional numbers always begin with zero, as in `0.25`;
- The first occurrence of a digit in the string signals the beginning of the number to be read.

The following code fragment does the job:

```
char line[100], *chrs = "-0123456789", *ptr;
float value;
    &vellip;
ptr = strpbrk(line, chrs);

    &vellip;
```

The character pointer `chrs` is initialized to point to a string of characters that might introduce the embedded number. The `strpbrk` function then finds the first occurrence of one of these characters in line and returns a pointer to that location in `ptr.` Finally, `ptr` is passed to `sscanf(),` which interprets `ptr` as if it were a pointer to the beginning of a string from which input is to be taken. The number is read correctly because `ptr` points to the beginning of a number, and because the `f` conversion terminates at the first inappropriate character.

See Also

`strchr()`, `strrchr()`, ANSI C 4.11.5.4, POSIX.1 8.1

strrchr

Locates the last occurrence of a supplied character within a string.

Syntax

```
#include <string.h.>
char *strrchr(const char *s, int c);
```

Parameters

s	A pointer to a null-terminated character string.
c	The value to find in the target string.

Return Values

x	A character pointer to the last occurrence of *c* in string *s*. This function returns a null pointer if the character is not found.

Description

The `strrchr` function searches the null-terminated string *s* for the last occurrence of *c* (converted to type `char`). The terminating null character is part of the string.

Example

```
   while((ptr = strrchr(string, '@')) != NULL)
      *ptr = '#';
```

Replace all @'s with #'s, starting from the end of the array, working backward toward the beginning.

See Also

`strchr()`, `strbrk()`, ANSI C 4.11.5.5, POSIX.1 8.1

strspn

Returns the length of the first substring in *s1* composed entirely of members of the character set *s2*.

Syntax

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Parameters

s1	A pointer to a null-terminated character string to be searched.
s2	A pointer to a null-terminated character string containing a character set.

Return Values

x	The length of the initial segment in *s1* formed by characters found in the *s2* character set.

Description

The `strspn` function sequentially processes each character in the array referenced by *s1*. For each character in the array, it scans *s2* looking for a match. If a match is found, a counter is incremented and the process continues. If a match is not found, this function ends, returning the value of the counter.

Example

Given the following two strings:

```
'A tattletale never wins.'
```

for string *s1,* and

```
' -Aatle'
```

for *s2.* Executing

```
strspn(s1, s2);
```

with the strings shown returns a value of 13, because the first 13 characters in *s1* all occur in *s2* "A tattletale ".

See Also

`strcspn()`, ANSI C 4.11.5.6, POSIX.1 8.1

strstr

Locates the first occurrence in one string of the sequence of characters specified by another string.

Syntax

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Parameters

s1	A pointer to the character string to search.
s2	A pointer to the character string with the search value.

Return Values

x	A pointer to the string found in *s1*. If *s2* is null, the value of *s1* is returned.
`NULL`	The *s2* was not found in *s1*.

Description

The `strstr` function scans *s1* searching for the first occurrence of the sequence of characters in *s2* (excluding the null character). If *s2* is found in *s1*, a pointer to the start of that string in *s1* is returned.

See Also

`strchr()`, `strrchr()`, ANSI C 4.11.5.7, POSIX.1 8.1

strtod

Converts a string to a double-precision, floating-point number.

Syntax

```
#include <stdlib.h>
double strtod (const char *str, char **ptr);
```

Parameters

str	A pointer to a character string to be converted.
ptr	If *ptr* is not NULL, a pointer to the character terminating the scan is stored in the object pointed to by *ptr*.

Return Values

x — A double-precision floating-point number resulting from the successful conversion of the string.

0 — Indicates failure unless the value pointed to by *str* is zero.

- If **ptr* is set to *str*, no number can be formed.
- If **ptr* is greater than *str*, the value pointed to by **str* is zero.
- If errno is set to ERANGE, the correct value of the conversion would cause an underflow.

±HUGE_VAL — The conversion would cause an overflow; errno is set to ERANGE.

Description

The strtod() function returns as a double-precision, floating-point number the value represented by the character string pointed to by *str.* The string is scanned up to the first unrecognizable character.

The string must contain a decimal constant or a floating-point constant that may optionally be preceded by white space. The complete string may contain the following in the order listed:

1. Optional white-space characters (as defined by isspace()).
2. Optional sign.
3. Required string of digits, optionally containing a decimal point.
4. Optional e or E.
5. Optional sign or space.
6. Integer.

See Also

`strtol()`, `strtoul()`, `atof()`, ANSI C 4.10.1.4

strtok

Divides string `s1` into zero or more tokens. The token separators consist of any characters contained in string `s2`.

Syntax

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Parameters

s1	A pointer to a string with zero or more tokens.
s2	A pointer to a character string with token delimiters.

Return Values

x	A pointer to the first character of a token.
`NULL`	No token found.

Description

A *token* is a string of characters delimited by one or more token delimiters. In the `strtok` function, `s1` is a character pointer to the string that is to be broken up into tokens, and `s2` is a character pointer to a string consisting of characters to be treated as token separators.

The `strtok` function returns the next token from *s1* each time it is called. The first time `strtok` is called, both *s1* and *s2* must be specified. On subsequent calls, *s1* is not specified (a null pointer is specified in its place). The `strtok` function remembers the string from call to call. String *s2* must be specified for each call, but need not contain the same characters (token separators).

The `strtok` function returns a pointer to the beginning of the token, and writes a null character into *s1* immediately following the end of the returned token, overwriting the token delimiter. This function returns a null pointer when no tokens remain.

Example

This example assumes that you are reading lines from a file containing several fields delimited by pound signs `(#)`. The following code could be used to read the fields of each line:

```
int count = 0;
char *delims = "#", *token, *arg1, *strtok(), line[256];
arg1 = line;
    …
while((token = strtok(arg1, delims)) != NULL) {
   count
   printf("field %d: %s\n", count, token);
```

```
      arg1 = NULL;
   }
```

This code sees to it that `strtok()`'s first argument is null after the first call. Also, note that `delims` did not change from call to call, but it could have. This greatly increases the power of `strtok`, because it enables you to change the token delimiters between calls.

See Also

ANSI C 4.11.5.8, POSIX.1 8.1

strtol

Converts a string to a long integer value.

Syntax

```
#include <stdlib.h>
long strtol (const char *str, char **ptr, int base);
```

Parameters

str	A pointer to a character string to be converted. If *base* is set to zero, leading characters in *str* define the conversion. After an optional leading sign, a leading zero indicates octal conversion, and a leading "0x" or "0X" indicates hexadecimal conversion. Otherwise, decimal conversion is used.
ptr	If *ptr* is not NULL, a pointer to the character terminating the scan is stored in the object pointed to by *ptr*.
base	If *base* is between 2 and 36, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16. If *base* is set to zero, the string itself determines the *base*.

Return Values

x	If successful, a long integer value.
0	Indicates failure unless the value pointed to by *str* is zero. • If **ptr* is set to *str*, no number can be formed. • If **ptr* is greater than *str*, the value pointed to by **str* is zero.
LONG_MAX	The conversion would cause an overflow; errno is set to ERANGE.
LONG_MIN	The conversion would cause an underflow; errno is set to ERANGE.

Description

The strtol function returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with *base*. Leading white-space characters (as defined by the isspace function) are ignored.

See Also

strtod(), strtoul(), atof(), ANSI C 4.10.1.5

strtoul

Converts a string to an `unsigned long int` representation.

Syntax

```
#include <stdlib.h>
unsigned long int strtoul (const char *str,
                           char **ptr,  int base);
```

Parameters

str	A pointer to a character string to be converted. If *base* is set to zero, leading characters in *str* define the conversion. After an optional leading sign, a leading zero indicates octal conversion, and a leading "0x" or "0X" indicates hexadecimal conversion. Otherwise, decimal conversion is used.
ptr	If *ptr* is not NULL, a pointer to the character terminating the scan is stored in the object pointed to by *ptr*.
base	If *base* is between 2 and 36, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16. If *base* is set to zero, the string itself determines the *base*.

Return Values

≠0	A string value converted to a long integer.
0	When **ptr* is set to *str* and a value of zero is returned, it indicates that no number can be formed.
ULONG_MAX	The conversion would cause an overflow; `errno` is set to ERANGE.

Description

The `strtoul` function returns as an unsigned long int the value represented by the character string pointed to by *str*.

See Also

`strtod()`, `strtol()`, `atof()`, ANSI C 4.10.1.6

strxfrm

Transforms a string in a manner appropriate for the current locale.

Syntax

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2,
      size_t n);
```

Parameters

s1	A string pointer to the destination string.
s2	A string pointer to a null-terminated source string.
n	The maximum number of characters to transform, including the terminating null character.

Return Values

x	The length of the transformed string, not including the terminating null character. If the value returned is greater than or equal to *n*, the contents of the array referenced by *s1* are undefined.

Description

The transformation is such that the `strcmp` function can be used to compare two transformed strings, giving the same result as `strcoll` applied to the original strings.

No more than *n* characters are placed into the resulting array pointed to by *s1* (including the terminating null character). If copying takes place between overlapping objects, the behavior is undefined.

See Also

`memcmp()`, `strcmp()`, `strcoll()`, ANSI C 4.11.4.5

swab

Swaps bytes in an array.

Syntax

```
void swab (char *from, char *to, int nbytes);
```

Parameters

from	A pointer to the source array.
to	A pointer to the target array.
nbytes	The number of bytes to copy.

Return Values

None.

Description

This function copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between byte-swapped and non-byte-swapped machines. The *nbytes* parameter should be even and non-negative. If *nbytes* is odd and positive, `swab()` uses *nbytes-1* instead. If *nbytes* is negative, `swab()` does nothing.

system

Executes an MPE/iX command.

Syntax

```
#include <stdlib.h>
int system (const char *string);
```

Parameters

string	A pointer to a string containing an MPE/iX command.

Return Values

0	Success.
<0	An error occurred. The value returned is the negated value of the error code returned by the HPCICOMMAND intrinsic.
>0	A warning occurred.

Description

The system function executes an MPE/iX command pointed to by *string*. The command can include UDCs and command files. The command is executed as if *string* has been entered at a terminal. The current process waits until the command completes.

This function is implemented by calling the MPE/iX system intrinsic HPCICOMMAND. Several commands are not executed when using this function. Refer to the description of HPCICOMMAND in the *MPE/iX Intrinsics Reference Manual* for a description of these commands.

All error and warning messages resulting from the execution of the command are printed to $STDLIST.

See Also

ANSI C 4.10.4.5

tan

Computes a tangent value.

Syntax

```
#include <math.h>
double tan (double x)
```

Parameters

x	A real number giving the angle measured in radians.

Return Values

n	The tangent of the angle.
0	Indicates a complete loss of accuracy for large values of *x*. A `TLOSS` error message is printed on the standard error output. `errno` is set to `ERANGE`.

Description

The `tan` function returns the tangent of its argument *x*, measured in radians. This function loses accuracy when its argument is sufficiently large. For less extreme arguments causing partial loss of significance, a `PLOSS` error is generated but no message is printed and `errno` is set to `ERANGE`.

Error handling can be changed by a user-written `matherr` function.

See Also

`sin()`, `cos()`, ANSI C 4.5.2.7, POSIX.1 8.1

tanh

Computes the hyperbolic tangent value for a given angle.

Syntax

```
#include <math.h>
double tanh (double x);
```

Parameters

x	A real number giving the angle measured in radians.

Return Values

n	The hyperbolic tangent of the angle.

Description

This function returns the hyperbolic tangent of its argument (*x*) in radians.

See Also

`sin()`, `cos()`, ANSI C 4.5.3.3, POSIX.1 8.1

tdelete

Deletes a specified node from a binary search tree.

Syntax

```
#include <search.h>
void *tdelete (void *key, void **rootp, int (*compar)());
```

Parameters

key	A pointer to an item to be searched for and deleted.
rootp	A pointer to the variable at the root of the tree.
compar	A pointer to a comparison function supplied by the user.

Return Values

x	A pointer to the parent node of the deleted entry.
`NULL`	Entry not found or *rootp* is `NULL` on entry.

Description

The `tdelete` function searches a binary search tree for the specified entry and deletes it if found. The tree is composed only of pointers. The values reference by these pointers are stored separately from the tree by the calling program.

The `tdelete`, `tfind`, `tsearch`, and `twalk` functions manage binary search trees generalized from Knuth Algorithms T and D (6.2.2) described in *The Art of Computer Programming, Vol3 (Sorting and Searching)* by Donald Ervin Knuth (Reading, Mass.:Addison-Wesley, 1973).

If there is an item in the tree equal to **key* (the value pointed to by *key*), the pointer to the item is removed from the tree and the descendants of that node are resorted. If no matching item is found in the tree, a null pointer is returned.

If the deleted node is the root of the tree, the variable pointed to by *rootp* is changed. A null value for the variable pointed to by *rootp* indicates an empty tree.

The pointers to the key and the root of the tree should be of type pointer-to- element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

All comparisons are done with the function *compar*, which must be supplied by the programmer. This function is called with two arguments, the pointers to the elements being compared. The comparison function does not need to compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

The comparison function should return an integer either less than, equal to, or greater than zero, according to whether the first argument is to be considered less than, equal to, or greater than the second argument.

If the calling function alters the pointer to the root, results are unpredictable.

NOTE The `tdelete` function and the header file `<search.h>` are not part of ANSI C. Using them may make your program less portable.

Examples

Refer to the example located in the `tsearch()` function description.

See Also

`tfind()`, `tsearch()`, `twalk()`

tfind

Searches for a specified entry in a binary search tree.

Syntax

```
#include <search.h>
void *tfind (void *key, void **rootp, int (*compar)());
```

Parameters

key	A pointer to an item to be searched for. If there is an item in the tree equal to **key* (the value pointed to by *key*), a pointer to the item found is returned. Otherwise, a null pointer is returned. Only pointers are copied, so the calling function must store the data.
rootp	A pointer to a variable that points to the root of the tree. A null value for the variable pointed to by *rootp* indicates an empty tree.
compar	All comparisons are done with the function *compar*, which must be supplied by the programmer. This function is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than zero, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function does not need to compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

Return Values

If successful, `tfind()` returns a pointer to the value pointed to by *key*. Otherwise, a null pointer is returned either if the entry was not found or if *rootp* is NULL on entry.

Description

The `tfind` function searches a binary search tree for the specified entry. The `tfind()`, `tsearch()`, `tdelete()`, and `twalk()` functions manage binary search trees generalized from Knuth Algorithms T and D (6.2.2) described in *The Art of Computer Programming, Vol3 (Sorting and Searching)* by Donald Ervin Knuth (Reading, Mass.:Addison-Wesley, 1973).

All comparisons are done with the function *compar*, which must be supplied by the programmer.

The pointers to the key and the root of the tree should be of type pointer-to-element, and should be cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

If the calling function alters the pointer to the root, results are unpredictable.

NOTE The `tfind` function and the header file `<search.h>` are not part of ANSI C. Using them may make your program less portable.

Examples

Refer to the example located in the `tsearch` function description.

See Also

`tsearch()`, `tdelete()`, `twalk()`

time

Returns the current calendar time.

Syntax

```
#include <time.h>
time_t time (time_t *timer);
```

Parameters

timer	If not NULL, a pointer to where the returned time is stored.

Return Values

x	Specifies the time elapsed seconds since the Epoch (00:00:00 Coordinated Universal Time, January 1, 1970).
0	An error occurred and errno is set to EFAULT.

Description

The time function returns the number of elapsed seconds since the Epoch, 00:00:00 Coordinated Universal Time (Greenwich Mean Time), January 1, 1970. If *timer* is not NULL, the return value is also assigned to the object that *timer* points to.

NOTE If linking with the POSIX/iX library, refer to the description of time() located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

clock(), difftime(), mktime(), ANSI C 4.12.2.4

tmpfile

Creates a temporary file.

Syntax

```
#include <stdio.h>
FILE *tmpfile (void);
```

Parameters

None.

Return Values

x	A pointer to a stream associated with the temporary file.
`NULL`	The file cannot be opened.

Description

The `tmpfile` function creates a temporary file using a name generated by `tmpnam()` and returns a pointer to the resulting stream. The file is automatically deleted when the process using it terminates.

If linking with the HP C/iX libraries, the file is opened for update as a binary stream using `wb+` mode.

If linking with the POSIX/iX library, the file is opened for update as a byte stream using the `ws+` mode.

See Also

`tmpnam()`, ANSI C 4.9.4.3, POSIX.1 8.1

tmpnam

Creates a name for a temporary file.

Syntax

```
#include <stdio.h>
char *tmpnam (char *s);
```

Parameters

s	Either `NULL` or a pointer to an array of at least `L_tmpnam` bytes, where `L_tmpnam` is a constant defined in `stdio.h>`.

Return Values

x	If *s* is `NULL`, a pointer to a static buffer which contains a file name. If *s* is not `NULL`, the value of the argument *s*.

Description

The `tmpnam` function generates a file name that can safely be used as a temporary file. This function generates a different file name each time it is called.

If *s* is null, `tmpnam()` leaves its result in an internal static area and returns a pointer to that area. The next call to `tmpnam()` destroys the contents of the area. If *s* is not null, it is assumed to be the address of an array of at least `L_tmpnam` bytes, where `L_tmpnam` is a constant defined in `<stdio.h>`; `tmpnam` places its result in that array and returns *s*.

A file created using `tmpnam()` and `fopen()` is temporary only in the sense that it is intended for temporary use. It is your responsibility to remove the file when it is no longer needed.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This is extremely unlikely if the other process is using this function or `mktemp()` because these functions choose file names in a way that minimizes duplication.

See Also

`tmpfile()`, ANSI C 4.9.4.4, POSIX.1 8.1

toascii

Converts an integer to 7-bit ASCII.

Syntax

```
#include <ctype.h>
int toascii (int c);
```

Parameters

c	The integer to convert to ASCII.

Return Values

x	Is returned with all bits turned off that are not part of the standard 7-bit ASCII character.

Description

The `toascii` function returns its argument with all bits turned off that are not part of a standard 7-bit ASCII character. It is intended for compatibility with other systems.

tolower, _tolower

Converts an uppercase letter to lowercase.

Syntax

```
#include <ctype.h>
int tolower (int c);
```

Parameters

c	An argument to be converted to lowercase.

Return Values

x	The lowercase letter that corresponds with *c*. If *c* is not an uppercase letter and the function is called, *c* is returned unchanged. If *c* is not an uppercase letter and the macro is called, the results are undefined.

Description

This conversion routine that downshifts ASCII characters is implemented both as a function and as a macro. The `tolower` function and `_tolower` macro have a domain the range of `getc()` (the integers from -1 through 255). If the argument passed in *c* represents an uppercase letter, `tolower()` returns the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

The `_tolower` macro accomplishes the same thing as the function, but has a restricted domain and is faster. The `_tolower` macro requires a lowercase letter as its argument. Arguments outside the domain cause undefined results.

NOTE The `tolower` function and macro do not work with foreign character sets.

Example

The following code fragment might appear in the scanner of a case-insensitive compiler, where all source input is mapped to lowercase before any processing is performed:

```
unsigned char *ps;
while (*ps != '\0'){
    *ps = tolower(*ps);
    &+&+ps;
}
```

See Also

`toupper()`, ANSI C 4.3.2.1, POSIX.1 8.1

toupper, _toupper

Converts a lowercase letter to uppercase.

Syntax

```
#include <ctype.h>
int toupper (int c);
```

Parameters

c	An argument to be converted to uppercase.

Return Values

x	The uppercase letter that corresponds with *c*. If *c* is not a lowercase letter and the function is called, *c* is returned unchanged. If *c* is not a lowercase letter and the macro is called, the results are undefined.

Description

This conversion routine that upshifts ASCII characters is implemented both as a function and as a macro. The `toupper` function and `_toupper` macro have a domain the range of `getc()` (the integers from -1 through 255). If the argument passed in *c* represents an lowercase letter, `toupper()` returns the corresponding uppercase letter. All other arguments in the domain are returned unchanged.

The `_toupper` macro accomplishes the same thing as the function, but has a restricted domain and is faster. The `_toupper` macro requires a lowercase letter as its argument. Arguments outside the domain cause undefined results.

NOTE The `toupper` function and macro do not work with foreign character sets.

Examples

The following code fragment might appear in the scanner of a case-insensitive compiler, where all source input is mapped to uppercase before any processing is performed.

```
unsigned char *ps;
while (*ps != '\0'){
    *ps = toupper(*ps);
    &+&+ps;
}
```

See Also

`tolower()`, ANSI C 4.3.2.2, POSIX.1 8.1

tsearch

Builds and provides access to a binary search tree.

Syntax

```
#include <search.h>
void *tsearch (void *key, void **rootp,
               int (*compar)());
```

Parameters

key	A pointer to an item to be accessed or stored.
rootp	A pointer to a variable that points to the root of the tree.
compar	A pointer to a comparison function supplied by the programmer.

Return Values

x	A pointer to the value pointed to by *key*.
NULL	There is not enough space available to create a new node or *rootp* is NULL.

Description

The tsearch function builds and accesses a binary search tree. The tsearch, tfind, tdelete, and twalk functions manage binary search trees generalized from Knuth Algorithms T and D (6.2.2) described in *The Art of Computer Programming, Vol3 (Sorting and Searching)* by Donald Ervin Knuth (Reading, Mass.:Addison-Wesley, 1973).

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

If the calling function alters the pointer to the root, results are unpredictable.

If there is an item in the tree equal to **key* (the value pointed to by *key*), a pointer to the item found is returned. Otherwise, **key* is inserted, and a pointer to it is returned. Only pointers are copied, so the calling function must store the data.

A null value for the variable pointed to by *rootp* indicates an empty tree; in this case, the variable is set to point to the item that is at the root of the new tree.

All comparisons are done with the function *compar*, which must be supplied by you. The function is called with two arguments, the pointers to the elements being compared.

It returns an integer less than, equal to, or greater than zero, according to whether the first argument is to be considered less than, equal to, or greater than the second argument.

The comparison function does not need to compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

NOTE The `tsearch()` function and the header file `<search.h>` are not part of ANSI C. Using them makes your program less portable.

Examples

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node { /* pointers to these are stored in the tree */
   char *string;
   int length;
};

char string_space[10000]; /* space to store strings */
struct node nodes[500]; /* nodes to store */
struct node *root = NULL; /* this points to the root */

main( )
{
   char *strptr = string_space;
   struct node *nodeptr = nodes;
   void print_node( ), twalk( );
   int i = 0, node_compare( );

   while (gets(strptr) != NULL  i++  500) {

      /* set node */
      nodeptr->string = strptr;
      nodeptr->length = strlen(strptr);

      /* put node into the tree */
      (void) tsearch((char *)nodeptr, root,
         node_compare);

      /* adjust pointers, so we don't overwrite tree */
      strptr += nodeptr->length + 1;
      nodeptr++;
   }
   twalk(root, print_node);
}

/* This function compares two nodes, based on an
   alphabetical ordering of the string field.  */
int
node_compare(node1, node2)
struct node *node1, *node2;
{
   return strcmp(node1->string, node2->string);
}

/* This function prints out a node, the first time twalk encounters it.  */
void
print_node(node, order, level)
struct node **node;
VISIT order;
int level;
{
   if (order == preorder||order == leaf) {
      (void)printf("string = %20s, length = %d\n",
         (*node)->string, (*node)->length);
   }
}
```

See Also

`tfind()`, `tdelete()`, `twalk()`

twalk

Traverses a binary search tree and returns the value at the specified node.

Syntax

```
#include <search.h>
void *twalk (void *root, void *(action)( ));
```

Parameters

root	A pointer to the starting node for the tree traversal.
action	The name of a user-supplied function to be invoked at each node.

Return Values

None.

Description

The `twalk` function performs a depth-first, left-to-right traversal of a binary search tree. The `tdelete`, `tfind`, `tsearch`, and `twalk` functions manage binary search trees generalized from Knuth Algorithms T and D (6.2.2). [1]

Any node in the tree can be used as the root for a walk below that node.

The pointer to the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to- element.

The *root* argument to `twalk()` is one level of indirection less than the *rootp* arguments to `tsearch()` and `tdelete()`.

The *action* function supplied by the calling program is invoked at each node. This function is, in turn, called with three arguments.

```
 void action_routine (struct **node, visit order, int level)
```

The first argument is the address of the node being visited.

The second argument is a value from an enumeration data type:

```
 typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

defined in the `<search.h>` header file, depending on whether this is the first, second or third time that the node has been visited during a depth- first, left-to-right traversal of the tree, or whether the node is a leaf.

There are two nomenclatures used to refer to the order in which tree nodes are visited. This implementation uses preorder, postorder and endorder to respectively refer to visiting

1. *The Art of Computer Programming, Vol3 (Sorting and Searching)* by Donald Ervin Knuth (Reading, Mass.:Addison-Wesley, 1973).

a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder, and postorder to refer to the same visits, which results in some confusion over the meaning of postorder.

The third argument is the level of the node in the tree, with the root being level zero.

NOTE The `twalk` function and the header file `<search.h>` are not part of ANSI C. Using them makes your program less portable.

Examples

Refer to the example located in the `tsearch()` function description.

See Also

`tfind()`, `tsearch()`, `tdelete()`

tzset

Sets time zone conversion information.

Syntax

```
#include <time.h> /* proto */
void tzset (void);
```

Parameters

None.

Return Values

None.

Description

The `tzset` function uses the value of the user-defined environment variable TZ to set time conversion information used by `localtime()`, `ctime()`, `strftime()`, and `mktime()`.

The `tzset()` function sets the external variable `tzname`:

`extern char *tzname[2] = {"std", "dst"};`

where *`std`* and *`dst`* are described in the *HP C/iX Library Reference Manual*. If no TZ environment variable exists, Eastern Standard Time (EST) and Eastern Daylight Time (EDT) for the United States are used (EST5EDT is assumed for TZ).

See Also

`localtime()`, `ctime()`, `strftime()`, `mktime()`, POSIX.1 8.3.2

ungetc

Pushes back a single character onto an open stream.

Syntax

```
#include <stdio.h>
int ungetc (int c, FILE *stream);
```

Parameters

c	A single character to push back.
stream	A pointer to an open stream.

Return Values

x	The value of the argument *c*, indicating success.
EOF	An error occurred.

Description

The `ungetc` function pushes the character specified by *c* (converted to an `unsigned char`) back onto the input stream pointed to by *stream*. The pushed-back character is returned by subsequent reads on the stream in the reverse order of their pushing. A successful intervening call to *stream* to a file positioning function (`fseek()`, `fsetpos()`, or `rewind()`) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character pushback is guaranteed. If the `ungetc()` function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro `EOF`, the operation fails and the input stream is unchanged.

A successful call to `ungetc()` clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters is the same as it was before the characters were pushed back.

For a text stream, the value of its file position indicator after a successful call to `ungetc()` is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to `ungetc()`, if its value was zero before the call, it is indeterminate after the call.

Examples

The following program reads one character from `stdin`, pushes it back onto `stdin`, rereads the character, and checks to make sure that this character and the character originally pushed back are the same. A message is printed on `stdout` stating the outcome of the comparison.

```
#include <stdio.h>
main()
{
   int c1, c2;

   c1 = getchar();
   ungetc(c1, stdin);
   c2 = getchar();
   if(c1 == c2)
      printf("They're the same!\n");
   else

      printf("Oops!  They're different!\n");
}
```

At least one character may be pushed back if data has been read from the stream prior to the push-back attempt and if the stream is buffered.

See Also

`fseek()`, `fsetpos()`, `ftell()`, `fclose()`, `ferror()`, `fopen()`, `fread()`, `fgetc()`, `gets()`, `putc()`, `fputc()`, `scanf()`, `fscanf()`, ANSI C 4.9.7.11, POSIX.1 8.1

va_arg

Initializes a variable to the beginning of an argument list.

Syntax

```
#include <stdarg.h>
type va_arg (va_list ap, type);
```

Parameters

ap	A pointer to a double, as defined by type `va_list` in `<varargs.h>`. This variable must be initialized with the macro `va_start`.
type	A data type, either built-in or user-defined.

Return Values

x	The value of the next argument in the call, returned as a variable of type *type*.

Description

This macro returns the value of the next argument in the argument list of functions that can be called with a variable number of arguments.

The macros `va_start`, `va_arg`, and `va_end` determine the arguments of functions with variable-length argument lists. Functions with variable-length argument lists are indicated by the ellipsis in the function header.

Successive invocations of `va_arg` return the values of the remaining arguments in succession. The `ap` argument is used by the macro to maintain the context of each successive call. This argument must be initialized by a call to `va_start` prior to calling `va_arg`.

The macro assumes that the return value is of type *type*. No error checking is performed. If there is no next argument when `va_arg` is called , or if *type* is not compatible with the type of the actual next argument, the value returned is unpredictable.

When writing functions with a variable number of arguments, you should provide a method for the calling procedure to either pass the number of actual arguments, or signal the end of the argument list. In addition, the calling program and function being called must cooperate closely as to the data types of the items in the variable argument list.

NOTE The header `<varargs.h>` also contains this macro. However, `<varargs.h>` header is not defined by the ANSI C standard.

Examples

Refer to the example located in the `va_start` macro description.

See Also

`va_start`, va_end, ANSI C 4.8.1.2

va_end

Terminates access to a variable argument list.

Syntax

```
#include <stdarg.h>
void va_end (va_list ap);
```

Parameters

ap	A pointer to a double, as defined by type `va_list` in `<varargs.h>`. This variable must be initialized with the macro `va_start`.

Return Values

None.

Description

The `va_end` macro terminates access to the variable argument list by making *ap* unusable. It must be called at the end of accessing the variable argument list.

The macros `va_start`, `va_arg`, and `va_end` determine the arguments of a function that can be called with a variable number of arguments. The variable number of arguments are indicated by the ellipsis in the function header.

NOTE The header `<varargs.h>` also contains this macros described in this section. However, `<varargs.h>` is not defined by the ANSI C standard.

Examples

Refer to the example located in the `va_start` macro description.

See Also

`va_arg`, `va_start`, ANSI C 4.8.1.3

va_start

Initializes a variable to the beginning of an argument list.

Syntax

```
#include <stdarg.h>
void va_start (va_list ap, parmN);
```

Parameters

ap	A pointer to a double, as defined by type va_list in <varargs.h>.
parmN	The identifier of the rightmost parameter in the variable parameter list in the function definition. This is the identifier just before the horizontal ellipsis.

Return Values

None.

Description

The va_start macro initializes *ap* (of type va_list) for subsequent use by va_arg and va_end. It must be invoked before va_arg can be used.

The macros va_start, va_arg, and va_end determine the arguments of a function that can be called with a variable number of arguments. The variable number of arguments are indicated by the ellipsis in the function header.

NOTE The header <varargs.h> also contains this macro. However, <varargs.h> is not defined by the ANSI C standard.

Examples

The following program uses <stdarg.h>:

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

enum arglisttype {NO_VAR_LIST, VAR_LIST_PRESENT};
enum argtype     {END_OF_LIST, CHAR, DOUB, INT, PINT};

int func (int a1, enum arglisttype a2, ...)
{
    va_list ap;

    enum argtype ptype;
    int i, *p;
    char c;
    double d;

    printf ("arg count = %d\n", a1);

    if (a2 == VAR_LIST_PRESENT) {
 /* Initialize the varargs mechanism */
 va_start(ap, a2);   /* pass a2 as an anchor */

 /* pick up all the arguments */
 do {
     /* get the type of the argument */
     ptype = va_arg (ap, enum argtype);

     /* retrieve the argument based on the type */
     switch (ptype) {
  case CHAR:  c = va_arg (ap, char);
       printf ("char = %c\n", c);
       break;

  case DOUB:  d = va_arg (ap, double);
       printf ("double = %f\n", d);
       break;

  case PINT:  p = va_arg (ap, int *);
       printf ("pointer = %x\n", p);
       break;

  case INT :  i = va_arg (ap, int);
       printf ("int = %d\n", i);
       break;

  case END_OF_LIST :
       break;

  default:    printf ("bad argument type %d\n", ptype);
       ptype = END_OF_LIST; /* to break loop */
       break;
     } /* switch */
 } while (ptype != END_OF_LIST);

 /* Clean up */
 va_end (ap);
    } /* if */
}

main()
{
    int x = 99;

    func (1, NO_VAR_LIST);
    func (2, VAR_LIST_PRESENT, DOUB, 3.0, PINT, &x, END_OF_LIST);
}
```

See Also

`va_arg`, `va_end`, ANSI C 4.8.1.1

vfprintf

Writes data in formatted form to an open stream using a variable argument list.

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf (FILE *stream, const char *format,
             va_list arg);
```

Parameters

stream	A pointer to an open stream where data is to be written.
format	A pointer to a character string defining the format (or the character string itself enclosed in double quotes).
arg	A variable argument list initialized by `va_start` (defined in the header `<stdarg.h>`).

Return Values

≥0	The number of characters written.
<0	An error occurred.

Description

The `vfprintf` function enables you to output data in formatted form to an open stream. In the `vfprintf` function, *string* points to an open stream, and *format* points to a character string (or the character string itself enclosed in double quotes) that specifies the format and content of the data to be written. The *arg* parameter is a variable argument list containing variables or expressions specifying the data to be written. *arg* must be initialized by the `va_start` macro prior to a call to `vfprintf()`.

The *arg* parameter specifies conversion specifications and literal characters. Literal characters are all characters that are not part of a conversion specification. Literal characters are written to the open stream exactly as they appear in the format.

Conversion Specifications

The following list shows the different components of a conversion specification in their correct sequence:

1. A percent sign (`%`), which signals the beginning of a conversion specification; to output a literal percent sign, you must type two percent signs (`%%`).
2. Zero or more flags, which affect the way a value is written (see below).
3. An optional decimal digit string which specifies a minimum `field width`.

4. An optional *`precision`* consisting of a dot (`.`) followed by a decimal digit string.
5. An optional `l, h,` or `L` indicating that the argument is of an alternate type. When used in conjunction with an integer conversion character, an `l` or `h` indicates a long or short integer argument, respectively. When used in conjunction with a floating-point conversion character, an `L` indicates a long double argument.
6. A conversion character, which indicates the type of data to be converted and printed.

A one-to-one correlation must exist between each specification encountered and each item in the item list.

The available *`flags`* are:

`-`	Causes the data to be left-justified within its output field. Normally, the data is right-justified.
`+`	Causes all signed data to begin with a sign `(+` or `-)`. Normally, only negative values have signs.
`blank`	Causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the `blank` and `+` flags both appear, the `blank` flag is ignored.
`#`	Causes the data to be written in an alternate form. Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.
`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 is ignored. The `0` flag is also ignored for `d`, `i`, `o`, `u`, `x`, and `X` conversions if a precision is specified.

A *`field width`*, if specified, determines the minimum number of spaces allocated to the output field for the particular piece of data being printed. If the data happens to be smaller than the field width, the data is blank- padded on the left (or on the right, if the `-` flag is specified) to fill the field. If the data is larger than the *`field width`*, the *`field width`* is simply expanded to accommodate the data. An insufficient *`field width`* never causes data to be truncated. If *`field width`* is not specified, the resulting field is made just large enough to hold the data.

The *`precision`* is a value which means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

NOTE A *`field width`* or *`precision`* may be replaced by an asterisk `(*)`. If so, the next item in the item list is fetched, and its value is used as the *`field width`* or *`precision`*. The *`item`* fetched must be an integer.

Conversion Characters

Conversion characters specify the type of data to expect in the item list and cause the data to be formatted and printed appropriately. The integer conversion characters are:

`d, i`	An integer *`item`* is converted to signed decimal. The *`precision`*, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the *`precision`*, the value is expanded with leading zeros. The default *`precision`* is 1. A null string results if a zero value is printed with a zero *`precision`*. The # flag has no effect.
`u`	An integer *`item`* is converted to unsigned decimal. The effects of the *`precision`* and the # flag are the same as for `d`.
`o`	An integer *`item`* is converted to unsigned octal. The # flag, if specified, causes the *`precision`* to be expanded, and the octal value is printed with a leading zero (a C convention). The *`precision`* behaves the same as in `d` above, except that writing a zero value with a zero *`precision`* results in only the leading zero being written, if the # flag is specified.
`x`	An integer *`item`* is converted to hexadecimal. The letters `abcdef` are used in writing hexadecimal values. The # flag, if specified, causes the *`precision`* to be expanded, and the hexadecimal value is written with a leading `"0x"` (a C convention). The *`precision`* behaves as in `d` above, except that writing a zero value with a zero *`precision`* results in only the leading `"0x"` being written, if the # flag is specified.
`X`	Same as `x` above, except that the letters `ABCDEF` are used to write the hexadecimal value, and the # flag causes the value to be written with a leading `"0X"`.

The character conversion characters are as follows:

`c`	The character specified by the `char` *`item`* is written. The *`precision`* is meaningless, and the # flag has no effect.
`s`	The string pointed to by the character pointer *`item`* is written. If a *`precision`* is specified, characters from the string are written until the number of characters indicated by the *`precision`* is reached, or until a null character is encountered, whichever comes first. If the *`precision`* is omitted, all characters up to the first null character are written. The # flag has no effect.

The floating-point conversion characters are:

`f` The `float` or `double` *`item`* is converted to decimal notation in style `f`; that is, in the form

```
[-]ddd.ddd
```

where the number of digits after the decimal point is equal to the *`precision`*. If *`precision`* is not specified, six digits are written after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. If the # flag is specified, a decimal point always appears, even if no digits follow the decimal point.

`e`	The `float` or `double` *`item`* is converted to scientific notation in style `e`; that is, in the form `[-]d.ddde±ddd` where there is always one digit before the decimal point. The number of digits after the decimal point is equal to the *`precision`*. If *`precision`* is not given, six digits are written after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. The exponent always contains exactly three digits. If the # flag is specified, the result always contains a decimal point, even if no digits follow the decimal point.
`E`	Same as `e` above, except that `E` is used to introduce the exponent instead of `e` (style `E`).
`g`	The `float` or `double` *`item`* is converted to either style `f` or style `e`, depending on the size of the exponent. If the exponent resulting from the conversion is less than -4 or greater than the *`precision`*, style `e` is used. Otherwise, style `f` is used. The *`precision`* specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the # flag is specified, the result always has a decimal point, even if no digits follow the decimal point, and trailing zeros are *not* removed.
`G`	Same as the `g` conversion above, except that style `E` is used instead of style `e`.

Other conversion characters are:

`p`	The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters.
`n`	The argument is a pointer to an integer into which is written the number of characters written to the output stream so far by this call to `fprintf()`. No argument is converted.
`%`	A `%` is written. No argument is converted. The complete conversion specification is &%&%.

The *`item`*s in the item list may be variable names or expressions. Note that, with the exception of the `s` conversion, pointers are *not* required in the item list. If the `s` conversion is used, a pointer to a character string must be specified.

See Also

`setlocale()`, `putc()`, `scanf()`, `fprintf()`, `printf()`, `vprintf()`, `vsprintf()`, `stdio()`, ANSI C 4.9.6.7

vprintf

Writes data in formatted form to the standard output stream `stdout` using a variable argument list.

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vprintf (const char *format, va_list arg);
```

Parameters

`format`	A pointer to a character string defining the format (or the character string itself enclosed in double quotes).
`arg`	A variable argument list initialized by `va_start()` (defined in the header `<stdarg.h>`.)

Return Values

≥0	The number of characters written.
<0	An error occurred.

Description

The `vprintf` function enables you to output data in formatted form to `stdout`. In the `vprintf` function, *`string`* is a pointer to `stdout`, and *`format`* is a pointer to a character string (or the character string itself enclosed in double quotes) that specifies the format and content of the data to be written. The *`arg`* parameter is a variable argument list containing variables or expressions specifying the data to be written. The *`arg`* parameter must be initialized by the `va_start` macro prior to a call to `vprintf()`.

The *`arg`* parameter specifies conversion specifications and literal characters. Literal characters are all characters that are not part of a conversion specification. Literal characters are written to `stdout` exactly as they appear in the format.

Conversion Specifications

The following list shows the different components of a conversion specification in their correct sequence:

1. A percent sign `(%),` that signals the beginning of a conversion specification; to output a literal percent sign, you must type two percent signs (`%%`).
2. Zero or more flags, that affect the way a value is printed (see below).
3. An optional decimal digit string that specifies a minimum `field width`.
4. An optional *`precision`*, consisting of a dot (`.`) followed by a decimal digit string.

5. An optional `l`, `h`, or `L` indicating that the argument is of an alternate type. When used in conjunction with an integer conversion character, an `l` or `h` indicates a long or short integer argument, respectively. When used in conjunction with a floating-point conversion character, an `L` indicates a long double argument.
6. A conversion character, which indicates the type of data to be converted and printed.

A one-to-one correlation must exist between each specification encountered and each item in the item list.

The available *`flags`* are:

`-`	Causes the data to be left-justified within its output field. Normally, the data is right-justified.
`+`	Causes all signed data to begin with a sign (`+` or `-`). Normally, only negative values have signs.
`blank`	Causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the `blank` and `+` flags both appear, the `blank` flag is ignored.
`#`	Causes the data to be printed in an *alternate form*. Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.
`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 is ignored. The `0` flag is also ignored for `d`, `i`, `o`, `u`, `x`, and `X` conversions if a precision is specified.

A *`field width`*, if specified, determines the *minimum* number of spaces allocated to the output field for the particular piece of data being printed. If the data happens to be smaller than the field width, the data is blank-padded on the left (or on the right, if the `-` flag is specified) to fill the field. If the data is larger than the *`field width`*, the *`field width`* is simply expanded to accommodate the data. An insufficient *`field width`* never causes data to be truncated. If no *`field width`* is specified, the resulting field is made just large enough to hold the data.

The *`precision`* is a value that means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

NOTE A *`field width`* or *`precision`* may be replaced by an asterisk (`*`). If so, the next item in the item list is fetched, and its value is used as the *`field width`* or *`precision`*. The *`item`* fetched must be an integer.

Conversion Characters

Conversion characters specify the type of data to expect in the item list and cause the data

to be formatted and printed appropriately. The integer conversion characters are:

`d`, `i`	An integer *`item`* is converted to signed decimal. The *`precision`*, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the *`precision`*, the value is expanded with leading zeros. The default *`precision`* is 1. A null string results if a zero value is printed with a zero *`precision`*. The # flag has no effect.
`u`	An integer *`item`* is converted to unsigned decimal. The effects of the *`precision`* and the # flag are the same as for `d.`
`o`	An integer *`item`* is converted to unsigned octal. The # flag, if specified, causes the *`precision`* to be expanded, and the octal value is printed with a leading zero (a C convention). The *`precision`* behaves the same as in `d` above, except that printing a zero value with a zero *`precision`* results in only the leading zero being printed, if the # flag is specified.
`x`	An integer *`item`* is converted to hexadecimal. The letters `abcdef` are used in printing hexadecimal values. The # flag, if specified, causes the *`precision`* to be expanded, and the hexadecimal value is printed with a leading `"0x"` (a C convention). The *`precision`* behaves as in `d` above, except that printing a zero value with a zero *`precision`* results in only the leading `"0x"` being printed, if the # flag is specified.
`X`	Same as `x` above, except that the letters `ABCDEF` are used to print the hexadecimal value, and the # flag causes the value to be printed with a leading `"0X".`

The character conversion characters are as follows:

`c`	The character specified by the `char` *`item`* is printed. The *`precision`* is meaningless, and the # flag has no effect.
`s`	The string pointed to by the character pointer *`item`* is printed. If a *`precision`* is specified, characters from the string are printed until the number of characters indicated by the *`precision`* is reached, or until a null character is encountered, whichever comes first. If the *`precision`* is omitted, all characters up to the first null character are printed. The # flag has no effect.

The floating-point conversion characters are:

`f` The `float` or `double` *`item`* is converted to decimal notation in style `f`; that is, in the form

```
[-]ddd.ddd
```

where the number of digits after the decimal point is equal to the *`precision`*. If no *`precision`* is specified, six digits are printed after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. If the # flag is specified, a decimal point always appears, even if no digits follow the decimal point.

`e` The `float` or `double` *`item`* is converted to scientific notation in style `e`;

that is, in the form

```
[-]d.ddde±ddd
```

where there is always one digit before the decimal point. The number of digits after the decimal point is equal to the *`precision`*. If no *`precision`* is given, six digits are printed after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. The exponent always contains exactly three digits. If the # flag is specified, the result always contains a decimal point, even if no digits follow the decimal point.

`E`	Same as `e` above, except that `E` is used to introduce the exponent instead of `e` (style `E`).
`g`	The `float` or `double` *`item`* is converted to either style `f` or style `e`, depending on the size of the exponent. If the exponent resulting from the conversion is less than -4 or greater than the *`precision`*, style `e` is used. Otherwise, style `f` is used. The *`precision`* specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the # flag is specified, the result always has a decimal point, even if no digits follow the decimal point, and trailing zeros are *not* removed.
`G`	Same as the `g` conversion above, except that style `E` is used instead of style `e.`

Other conversion characters are:

`p`	The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters.
`n`	The argument is a pointer to an integer into which is written the number of characters written to the output stream so far by this call to `fprintf()`. No argument is converted.
`%`	A `%` is written. No argument is converted. The complete conversion specification is &%&%.

The *`item`*s in the item list may be variable names or expressions. Note that, with the exception of the `s` conversion, pointers are *not* required in the item list. If the `s` conversion is used, a pointer to a character string must be specified.

See Also

`setlocale()`, `putc()`, `scanf()`, `vfprintf()`, `vsprintf()`, `stdio()`, ANSI C 4.9.6.8

vsprintf

Writes formatted data to a character string in memory using a variable argument list.

Syntax

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf (char *string, const char *format,
              va_list arg);
```

Parameters

string	A pointer to a buffer in memory where the data is to be written.
format	A pointer to a character string defining the format (or the character string itself enclosed in double quotes).
arg	A variable argument list initialized by `va_start` (defined in the header `<stdarg.h>`).

Return Values

≥0	The number of characters written.
<0	An error occurred.

Description

The *arg* parameter is a variable argument list containing variables or expressions specifying the data to be written. The *arg* parameter must be initialized by the `va_start` macro prior to a call to `vsprintf()`.

This function returns the number of characters written to the string, not counting the terminating null character, or a negative value (if an error occurs).

The *arg* parameter specifies conversion specifications and literal characters. Literal characters are all characters that are not part of a conversion specification. Literal characters are written to the buffer in memory exactly as they appear in the format.

Conversion Specifications

The following list shows the different components of a conversion specification in their correct sequence:

1. A percent sign (`%`), which signals the beginning of a conversion specification; to output a literal percent sign, you must type two percent signs (`%%`);
2. Zero or more flags, which affect the way a value is written (see below).
3. An optional decimal digit string which specifies a minimum `field width`.
4. An optional *precision* consisting of a dot (`.`) followed by a decimal digit string.

5. An optional `l, h,` or `L`, indicating that the argument is of an alternate type. When used in conjunction with an integer conversion character, an `l` or `h` indicates a long or short integer argument, respectively. When used in conjunction with a floating-point conversion character, an `L` indicates a long double argument.
6. A conversion character, which indicates the type of data to be converted and written.

A one-to-one correlation must exist between each specification encountered and each item in the item list.

The available *`flags`* are:

`-`	Causes the data to be left-justified within its output field. Normally, the data is right-justified.
`+`	Causes all signed data to begin with a sign `(+` or `-).` Normally, only negative values have signs.
`blank`	Causes a blank to be inserted before a positive signed value. This is used to line up positive and negative values in columnar data. Otherwise, the first digit of a positive value is lined up with the negative sign of a negative value. If the `blank` and `+` flags both appear, the `blank` flag is ignored.
`#`	Causes the data to be written in an alternate form. Refer to the descriptions of the conversion characters below for details concerning the effects of this flag.
`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 is ignored. The `0` flag is also ignored for `d`, `i`, `o`, `u`, `x`, and `X` conversions if *`precision`* is specified.

A *`field width`*, if specified, determines the minimum number of spaces allocated to the output field for the particular piece of data being written. If the data happens to be smaller than the field width, the data is blank-padded on the left (or on the right, if the `-` flag is specified) to fill the field. If the data is larger than the *`field width`*, the *`field width`* is simply expanded to accommodate the data. An insufficient *`field width`* never causes data to be truncated. If *`field width`* is not specified, the resulting field is made just large enough to hold the data.

The *`precision`* is a value which means different things depending on the conversion character specified. Refer to the descriptions of the conversion characters below for more details.

NOTE A *`field width`* or *`precision`* may be replaced by an asterisk `(*).` If so, the next item in the item list is fetched, and its value is used as the *`field width`* or *`precision`*. The *`item`* fetched must be an integer.

Conversion Characters

Conversion characters specify the type of data to expect in the item list and cause the data

to be formatted and written appropriately. The integer conversion characters are:

`d`	An integer *`item`* is converted to signed decimal. The *`precision`*, if given, specifies the minimum number of digits to appear. If the value has fewer digits than that specified by the *`precision`*, the value is expanded with leading zeros. The default *`precision`* is 1. A null string results if a zero value is written with a zero *`precision`*. The # flag has no effect.
`u`	An integer *`item`* is converted to unsigned decimal. The effects of the *`precision`* and the # flag are the same as for `d`.
`o`	An integer *`item`* is converted to unsigned octal. The # flag, if specified, causes the *`precision`* to be expanded, and the octal value is written with a leading zero (a C convention). The *`precision`* behaves the same as in `d` above, except that writing a zero value with a zero *`precision`* results in only the leading zero being written, if the # flag is specified.
`x`	An integer *`item`* is converted to hexadecimal. The letters `abcdef` are used in writing hexadecimal values. The # flag, if specified, causes the *`precision`* to be expanded, and the hexadecimal value is written with a leading `"0x"` (a C convention). The *`precision`* behaves as in `d` above, except that writing a zero value with a zero *`precision`* results in only the leading `"0x"` being written, if the # flag is specified.
`X`	Same as `x` above, except that the letters `ABCDEF` are used to write the hexadecimal value, and the # flag causes the value to be written with a leading `"0X"`.

The character conversion characters are as follows:

`c`	The character specified by the `char` *`item`* is written. The *`precision`* is meaningless, and the # flag has no effect.
`s`	The string pointed to by the character pointer *`item`* is written. If a *`precision`* is specified, characters from the string are written until the number of characters indicated by the *`precision`* is reached, or until a null character is encountered, whichever comes first. If the *`precision`* is omitted, all characters up to the first null character are written. The # flag has no effect.

The floating-point conversion characters are:

`f`	The `float` or `double` *`item`* is converted to decimal notation in style `f`; that is, in the form `[-]ddd.ddd` where the number of digits after the decimal point is equal to the *`precision`*. If *`precision`* is not specified, six digits are written after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. If the # flag is specified, a decimal point always appears, even if no digits follow the decimal point.
`e`	The `float` or `double` *`item`* is converted to scientific notation in style `e`; that is, in the form `[-]d.ddde±ddd`

	where there is always one digit before the decimal point. The number of digits after the decimal point is equal to the *`precision`*. If *`precision`* is not given, six digits are written after the decimal point. If the *`precision`* is explicitly zero, the decimal point is eliminated entirely. The exponent always contains exactly three digits. If the # flag is specified, the result always contains a decimal point, even if no digits follow the decimal point.
`E`	Same as `e` above, except that `E` is used to introduce the exponent instead of `e` (style `E`).
`g`	The `float` or `double` *`item`* is converted to either style `f` or style `e`, depending on the size of the exponent. If the exponent resulting from the conversion is less than -4 or greater than the *`precision`*, style `e` is used. Otherwise, style `f` is used. The *`precision`* specifies the number of significant digits. Trailing zeros are removed from the result, and a decimal point appears only if it is followed by a digit. If the # flag is specified, the result always has a decimal point, even if no digits follow the decimal point, and trailing zeros are *not* removed.
`G`	Same as the `g` conversion above, except that style `E` is used instead of style `e.`

Other conversion characters are:

`p`	The argument is a pointer to `void`. The value of the pointer is converted to a sequence of printable characters.
`n`	The argument is a pointer to an integer into which is written the number of characters written to the output stream so far by this call to `fprintf()`. No argument is converted.
`%`	A `%` is written. No argument is converted. The complete conversion specification is &%&%.

The *`item`*s in the item list may be variable names or expressions. Note that, with the exception of the `s` conversion, pointers are not required in the item list. If the `s` conversion is used, a pointer to a character string must be specified.

See Also

`setlocale(), putc()`, `scanf()`, `vfprintf()`, `vprintf()`, `stdio()`, ANSI C 4.9.6.9

wcstombs

Converts a sequence of wide character codes to a sequence of multibyte characters.

Syntax

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Parameters

s	A pointer to a character array to which the converted multibyte characters are returned.
pwcs	A pointer to the sequence of wide characters to be converted.
n	A variable of type `size_t` indicating the maximum number of bytes to return.

Return Values

x	The number of array elements modified, not including a terminating zero.
–1	Invalid wide character found.

Description

The sequence of wide character codes from the array pointed to by *pwcs* are converted into a sequence of multibyte characters and stored in the array pointed to by *s*. The conversion ends when a null character is stored or *n* is reached, whichever occurs first.

If a code is encountered that does not correspond to a valid multibyte character, `wcstombs` returns `(size_t)-1`. Otherwise, this function returns the number of bytes modified (not including a terminating null character, if any).

See Also

`mblen()`, `mbstowcs()`, `mbtowc()`, `wctomb()`, ANSI C 4.10.8.2

wctomb

Converts a single wide character value to its multibyte character representation.

Syntax

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Parameters

s	A pointer to a character array to which the multibyte character is returned.
wchar	The wide character value to be converted.

Return Values

>0	The length of the multibyte character in bytes.
–1	The *wchar* parameter does not point to a valid wide character.
0	The *wchar* parameter is a null character.

Description

The `wctomb` function converts the wide character *wchar* to multibyte representation and stores the result in the array pointed to by *s* (if *s* is not a null pointer).

This function retains state information. Multibyte encodings can be state-dependent, employing "shift characters" to alter the meaning of subsequent characters. The shift state is persistent between calls to the routines for processing extended character sets unless the LC_CTYPE category of the locale is changed.

Calling this function with the *s* argument set to NULL resets the function to its initial state. When using a NULL pointer to clear the shift state, zero is returned if the multibyte shift state was previously clear. A nonzero value is returned if the locale-specific shift state was previously set.

If the value of *wchar* is zero, `wctomb` is left in the initial shift state.

If *s* is not a null pointer, `wctomb` returns –1 if the value of *wchar* does not correspond to a valid multibyte character, or returns the number of bytes in the multibyte character corresponding to the value of *wchar*

The value returned cannot be greater than the value of the MB_CUR_MAX macro.

See Also

`wchar_t`, MB_CUR_MAX, `mbtowc()`, `wcstomb()`, `mbstowcs()`, ANSI C 4.10.7.3

write

Writes data to a file.

Syntax

```
int write (int fildes, char *buffer, unsigned nbyte);
```

Parameters

fildes	The file descriptor of the file to write to.
buffer	A pointer to a buffer containing data to write.
nbyte	The number of bytes to write.

Return Values

≥0	The number of bytes written.
–1	An error occurred. The file position indicator remains unchanged and `errno` is set to one of the following values:

EBADF	The *fildes* parameter is not a valid file descriptor open for writing.
EFBIG	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
ESYSERR	A call to a system intrinsic failed.

Description

The `write` function writes *nbyte* bytes from the buffer pointed to by *buffer* to the file associated with *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file position indicator. Upon return from `write()`, the file position indicator is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the device's current position. The value of a file position indicator associated with such a device is undefined.

If the file is opened for append mode, the file position indicator is set to the end of the file prior to each write.

If `write()` requests that more bytes be written than there is room for, `write()` fails and –1 is returned.

NOTE If linking with the POSIX/iX library, refer to the description of `write()` located in the *MPE/iX Developer's Kit Reference Manual*.

See Also

`read()`, `open()`

A Time Zones

This appendix contains a list of commonly used time zones and the TZ environment variable strings that correspond to these time zones. The TZ strings are used by the time and date library functions for adjustment to specific time zones. Refer to the description of the `ctime` function in chapter 5 for details.

The first line of each entry contains the time zone name followed by the Daylight Savings Time zone name, if appropriate. The next few lines contain the geographic locations associated with this time zone. The last line contains the TZ environment variable string that corresponds to this time zone.

Hawaiian Standard Time, Hawaiian Daylight Time
United States: Hawaii
`HST10`

Aleutian Standard Time, Aleutian Daylight Time
United States: Alaska (parts)
`AST10ADT`

Yukon Standard Time, Yukon Daylight Time
United States: Alaska (parts)
`YST9YDT`

Pacific Standard Time, Pacific Daylight Time
Canada: British Columbia.
`PST8PDT - Canada`

Pacific Standard Time, Pacific Daylight Time
United States: California, Idaho (parts), Nevada, Oregon (parts), Washington.
`PST8PDT`

Mountain Standard Time, Mountain Daylight Time
Canada: Alberta, Saskatchewan (parts).
`MST7MDT - Canada`

Mountain Standard Time, Mountain Daylight Time
United States: Colorado, Idaho (parts), Kansas (parts), Montana, Nebraska (parts), New Mexico, North Dakota (parts), Oregon (parts), South Dakota (parts), Texas (parts), Utah, Wyoming.
`MST7MDT`

Mountain Standard Time
United States: Arizona
`MST7`

Central Standard Time, Central Daylight Time

Canada: Manitoba, Ontario (parts), Saskatchewan (parts).
`CST6CDT - Canada`

Central Standard Time, Central Daylight Time
United States: Alabama, Arkansas, Florida (parts), Illinois, Iowa, Kansas, Kentucky (parts), Louisiana, Michigan (parts), Minnesota, Mississippi, Missouri, Nebraska, North Dakota, Oklahoma, South Dakota, Tennessee (parts), Texas, Wisconsin.
`CST6CDT`

Eastern Standard Time, Central Daylight Time
United States: Indiana (most)
`EST6CDT`

Eastern Standard Time, Eastern Daylight Time
Canada: Ontario (parts), Quebec (parts).
`EST5EDT - Canada`

Eastern Standard Time, Eastern Daylight Time
United States: Connecticut, Delaware, District of Columbia, Florida, Georgia, Kentucky, Maine, Maryland, Massachusetts, Michigan, New Hampshire, New Jersey, New York, North Carolina, Ohio, Pennsylvania, Rhode Island, South Carolina, Tennessee (parts), Vermont, Virginia, West Virginia.
`EST5EDT`

Atlantic Standard Time, Atlantic Daylight Time
Canada: Newfoundland (parts), Nova Scotia, Prince Edward Island, Quebec (parts).
`AST4ADT`

Newfoundland Standard Time, Newfoundland Daylight Time
Canada: Newfoundland (parts).
`NST3:30NDT`

Western European Time, Western European Time Daylight Savings Time
Great Britain, Ireland
`WET0WETDST`

Portuguese Winter Time, Portuguese Summer Time
`PWT0PST`

Mitteleuropaeische Zeit, Mitteleuropaeische Sommerzeit
`MEZ-1MESZ`

Middle European Time, Middle European Time Daylight Savings Time
Belgium, Luxembourg, Netherlands, Denmark, Norway, Austria, Poland, Czechoslovakia, Sweden, Switzerland, DDR, DBR, France, Spain, Hungary, Italy, Yugoslavia
`MET-1METDST`

South Africa Standard Time, South Africa Daylight Time
`SAST-2SADT`

Japan Standard Time
Japan
`JST-9`

Australian Western Standard Time
Australia: Western Australia
`WST-8:00`

Australian Central Standard Time
Australia: Northern Territory
`CST-9:30`

Australian Central Standard Time, Australian Central Daylight Time
Australia: South Australia
`CST-9:30CDT`

Australian Eastern Standard Time
Australia: Queensland
`EST-10`

Australian Eastern Standard Time, Australian Eastern Daylight Time
Australia: New South Wales, Victoria
`EST-10EDT`

Australian Eastern Standard Time, Australian Eastern Daylight Time
Australia: Tasmania
`EST-10EDT - Tasmania`

New Zealand Standard Time, New Zealand Daylight Time
`NZST-12NZDT`

TZTAB Time Zone Adjustment Table

The differences between Coordinated Universal Time (UTC) and local time are described in table form in the file `TZTAB.LIB.SYS`. This table can be used in conjunction with historical information to represent several local areas simultaneously. This file is also used by `mktime()` to compute the UTC from the local time.

The `TZTAB` file contains one or more time zone adjustment entries. The first line of the entry contains a unique string that is compared to the value of the `TZ` environment variable.

The format of the first line of a time zone adjustment entry is:

`tzname diff dstzname`

where:

`tzname`	Is the time zone name or abbreviation.
`diff`	Is the difference in hours from UTC. Fractional values of *`diff`* are expressed in minutes preceded by a colon. For example, `1:15` equals one hour and fifteen minutes.
`dstzname`	Is the name or abbreviation of the Daylight Savings Time zone.

The first line of the entry always begins with an alphabetic character.

The second and subsequent lines of each entry contain details about the adjustments for a time zone. Each line contains seven fields. The seventh field specifies a particular time zone adjustment. The first through sixth fields describe the time range for the adjustment. The fields are separated by blanks or tabs. The meanings of each field are as follows:

First field (0-59)	Specifies the minute at which a time zone adjustment takes effect.
Second field (0-23)	Specifies the hour at which a time zone adjustment takes effect.
Third field (1-31)	Specifies the day of the month at which a time zone adjustment takes effect.
Fourth field (1-12)	Specifies the month in which a time zone adjustment takes effect.
Fifth field (1970-1999)	Specifies the year in which a time zone adjustment takes effect.
Sixth field (0-6)	Specifies the day of the week in which a time zone adjustment takes effect, with 0 equal to Sunday, 1 to Monday, etc.

The minute, hour, and month fields contain a single number within the range given above for each. The day of the month, year, and day of the week fields may contain a single number or a range of numbers separated by a minus sign. Either the day of the month or the day of the week field must be a range, and the other must be a single number.

The seventh field is a string that describes the time zone adjustment in the following format:

`tznamediff`

where:

tzname	An alphabetic string containing the time zone name or abbreviation. The *tzname* value must match either the *tzname* field in the first line of the time zone adjustment entry or the *dstzname* field in the first line of the entry.
diff	The difference in hours from GMT. Any fractional value of *diff* is shown in minutes.

Comments are allowed within time zone adjustment entries. They begin with a pound sign (#) and include all characters up to a new line. Comments are ignored.

Example

The time zone adjustment entry for the Eastern Time Zone in the United States is as follows:

```
EST5EDT
0 3 6     1  1974      0-6 EDT4
0 3 22-28 2  1975      0   EDT4
0 3 24-30 4  1976-1986 0   EDT4
0 3 1-7   4  1987-1999 0   EDT4
0 1 24-30 11 1974      0   EST5
0 1 25-31 10 1975-1999 0   EST5
```

Normally, Eastern Standard Time (EST) is five hours earlier than Coordinated Universal Time. This is indicated in the first line. However, during Eastern Daylight Time, the difference is four hours. The first time Eastern Daylight Time took effect was on January 6, 1974 at 3:00 am EDT. This information is given in the second line. Note that the minute before was 1:59 am EST. The change back to standard time took effect on the last Sunday in November of the same year. This information is given on the sixth line. At that point, the time changed from 1:59 am EDT to 1:00 am EST. The transition to Eastern Daylight Time since then has gone from the last Sunday in February (indicated on the third line) to the last Sunday in April (fourth line) to the first Sunday in April (fifth line). The return to standard time for the same period has consistently occurred on the last Sunday in October (seventh line).

The TZTAB file was developed by Hewlett-Packard. It can support and is compatible with Native Language Support (NLS).

B Restrictions and Special Considerations

This appendix addresses restrictions and considerations that are not in the range of this manual.

Identifier Names

Function names beginning with an underscore (_) are reserved for library use. Therefore, you should not specify identifiers that begin with an underscore.

File Access Restrictions

You can open the following *special files* for read-only or write-only, but not for update:

- variable record length files
- circular files
- RIO files
- message files
- KSAM files

Attempting to open one of these files with update mode will result in an `open` error. Random access to these files using `fseek`, `lseek`, or `rewind` is allowed only on files opened with read access only.

If linking with the POSIX/iX lbrary, only files whose underlying format is byte stream can be created or opened.

Mixed I/O from the C System and Other Systems

With one exception, concurrent use of the HP C I/O system and another I/O system to output data to the same disk file is not supported. The one exception is interleaved output using another I/O system and the HP C I/O system through the standard C streams `stdout` and `stderr`. In this case, if you want the C output to appear in the file after a call to an output routine is made, you must call the `fflush` function immediately after the call or change the buffering scheme for the `stdout` and `stderr` streams to completely unbuffered or line buffered by calling the C library function `setvbuf`. Refer to the description of the `setvbuf` function in chapter 5 for details.

C System-Dependent Information

This appendix briefly summarizes the differences between the HP C/iX library as it is implemented on HP 3000 Series 900 computers and HP 9000 Series 700/800 computers. Because the HP 9000 Series 700/800 are UNIX-based systems, the summary of differences given will usually apply to other UNIX-based systems as well. Refer to chapter 5 for detailed descriptions of the HP C/iX library functions. Refer to the *HP C/HP-UX Reference Manual* for complete descriptions of the HP C/HP-UX functions.

Additional differences between POSIX/iX library functions and HP C/HP-UX library functions are described in the *MPE/iX Developer's Kit Reference Manual*.

This appendix is organized alphabetically by function name. For each function, a description of the behavior on both systems is provided.

abort

HP 9000 Series 700/800: The `abort` function sends a signal to the calling process to terminate it. If this signal is caught or ignored, `abort` returns without terminating the process. If this signal is neither caught nor ignored, a core dump is produced and a message is issued.

HP 3000 Series 900: The `abort` function uses the `QUIT` intrinsic to terminate a process. This always results in process termination. No core dump is produced.

access

HP 9000 Series 700/800: The following *amode* parameter bit pattern is supported:

```
01 execute (search)
```

This option checks whether a file may be executed.

HP 3000 Series 900: The `execute` *amode* parameter bit pattern is not supported.

brk and sbrk

HP 9000 Series 700/800: Newly allocated space obtained from `brk` and `sbrk` is set to zero.

HP 3000 Series 900: Newly allocated space obtained from `brk` and `sbrk` is not set to zero.

malloc

HP 9000 Series 700/800: The default memory allocation package provided in the C library is not the *fast* memory allocation package. This package is known as the `malloc(3C)` package.

HP 3000 Series 900: The memory allocation package provided in the C library is the *fast* memory allocation package. This package is known as the `malloc(3X)` package on the HP 9000 Series 700/800. It is available on HP 9000 Series 700/800 by using the `-lmalloc` linker option.

mktemp

HP 9000 Series 700/800: The string of X's in the argument to `mktemp` are replaced by the current process identification number.

HP 3000 Series 900: The string of X's in the argument to `mktemp` are replaced by a randomly generated number.

`open`

HP 9000 Series 700/800: The following options are available:

`O_NDELAY:`	This option controls whether a process blocks on an I/O request until the request is completed.
`O_EXCL:`	If `O_EXCL` and `O_CREAT` are set, `open()` fails if the file exists.
`O_SYNCIO:`	If a file is opened with `O_SYNCIO`, file system writes for that file are done through the cache to the disk as soon as possible, and the process blocks until this is completed.

The `O_MPEOPTS` option that allows you to specify MPE-like file attributes is not available.

HP 3000 Series 900: The `O_MPEOPTS` option is available. The `O_NDELAY`, `O_EXCL` and `O_SYNCIO` options are not available.

`read`

HP 9000 Series 700/800: An `open` option is available to specify whether or not a process should block until the `read` request is complete.

HP 3000 Series 900: No `open` option is available to specify whether or not a process should block until the `read` request is completed. A process always blocks until the `read` request is complete.

`setjmp` and `longjmp`

HP 9000 Series 700/800: The `setjmp` and `longjmp` functions save and restore a signal mask while `_setjmp` and `_longjmp` manipulate only the stack and registers. The `setjmp` and `longjmp` functions may be able to detect a condition in which the environment of the `setjmp` no longer exists and recover.

HP 3000 Series 900: The `setjmp` and `longjmp` functions do not save and restore a signal mask; they manipulate only the stack and registers. The `setjmp` and `longjmp` functions are not able to detect a condition in which the environment of the `setjmp` no longer exists.

`sleep`

HP 9000 Series 700/800: The `sleep` function is implemented using signals. These signals may cause the time slept to be more or less than the requested sleep time. If the actual sleep time is less than the requested sleep time, `sleep` returns the difference in these two times.

Seconds must be less than 2^{32}.

HP 3000 Series 900: The `sleep` function is implemented by calling the `PAUSE` intrinsic. Signals will not interfere with the amount of time slept. `Sleep` returns its argument if an error occurs, zero if no error occurs.

Seconds must be less than 2,147,485.

`write`

HP 9000 Series 700/800: If a `write` requests more bytes to be written than there is room for, the `write` fails and -1 is returned.

An `open` option is available to specify whether or not a process should block until the `write` request has completed.

HP 3000 Series 900: If a `write` requests more bytes to be written than the file size limit, only as many bytes as there is room for are written. For example, if there is space for 20 bytes more in a file before reaching a limit, a `write` of 512 bytes returns 20. The next `write` of a non-zero number of bytes gives a failure return.

No `open` option is available to specify whether or not a process should block until the `write` request has completed. A process blocks until the `write` request is completed.