Microchip Unified Standard Library Reference Guide

Notice to Development Tools Customers

Important:
All documentation becomes dated, and Development Tools manuals are no exception. Our tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our website (www.microchip.com/) to obtain the latest version of the PDF document.

Documents are identified with a DS number located on the bottom of each page. The DS format is DS<DocumentNumber><Version>, where <DocumentNumber> is an 8-digit number and <Version> is an uppercase letter.

For the most up-to-date information, find help for your tool at onlinedocs.microchip.com/.

Table of Contents

1. Preface

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation:

Table 1-1. Documentation Conventions

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	*File>Save*
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
N'Rnnnn	A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	4'b0010, 2'hF1
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier New font:		
Plain Courier New	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`c:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`
Italic Courier New	A variable argument	*`file.o`*, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`mcc18 [options] file [options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void) { ... }`

1.2 Recommended Reading

This document describes the behavior of and interface to the functions defined by the Microchip Unified Standard Library, as well as the intended use of the library types and macros. The following Microchip documents are available and recommended as supplemental reference resources.

MPLAB® XC8 C Compiler User's Guide for PIC® MCU
This guide describes the features and usage of the MPLAB XC8 C compiler when targeting 8-bit PIC devices.

MPLAB® XC8 C Compiler User's Guide for AVR® MCU
This guide describes the features and usage of the MPLAB XC8 C compiler when targeting 8-bit AVR devices.

MPLAB® XC16 C Compiler User's Guide
This guide describes the features and usage of the MPLAB XC16 C compiler for all 16-bit PIC MCUs and dsPIC® Digital Signal Controllers.

MPLAB® XC32 C/C++ Compiler User's Guide for PIC32C/SAM MCUs
This guide describes the features and usage of the MPLAB XC32 C compiler for all 32-bit PIC32C and SAM MCUs.

MPLAB® XC32 C/C++ Compiler User's Guide for PIC32M MCUs
This guide describes the features and usage of the MPLAB XC32 C compiler for all 32-bit PIC32M MCUs.

2. The Microchip Unified Standard Library

A library is a collection of functions, types, and preprocessor macros that are grouped for reference and ease of linking. This document describes the behavior of and interface to the functions defined by the Microchip Unified Standard Library, as well as the intended use of the library types and macros. This library is provided with all Microchip C Compilers, those being:

- MPLAB® XC8 C Compiler, for all 8-bit PIC® and AVR® MCUs
- MPLAB XC16 C Compiler, for all 16-bit PIC MCUs and dsPIC® Digital Signal Controllers
- MPLAB XC32 C/C++ Compiler, for all 32-bit PIC and SAM MCUs and MPUs

This library encompasses the functions defined by the standard C99 language specification headers, as well as any types and preprocessor macros needed to facilitate their use. Any additional resources permitted by the C99 language specification or additional functions and resources supplied by Microchip and which are related to the operation of the standard library are also documented.

This document groups the functions and related types and macros under the header that defines them. Not all the functions, types, and macros described are available for all compilers or for all devices. This document indicates when a resource has limited applicability or that a difference in definition is relevant in different contexts.

See your compiler's user guide for more information on compiler features referenced in this guide or the use of libraries in your project.

3. General Notes Concerning Library Code

The following points relate to the general usage of functions within the library.

MPLAB XC8 Functions With Floating-point Arguments

With the MPLAB XC8 compiler (for both PIC and AVR targets), the `double` and `long double` types are the same size and format as the 32-bit `float` type. Many math-related functions have separate implementations for each of the three floating-point types (e.g. `acos` for `double` types, `acosl` for `long double` types, and `acosf` for `float` types). In these cases, only the `float` instance of the function is fully implemented. Calls to either the `double` or `long double` variants of the functions will be mapped to the `float` implementation.

MPLAB XC16 and double Types

By default, the MPLAB XC16 compiler uses a `double` type equivalent to `float`, which is 32 bit wide. The `-fno-short-double` option makes `double` equivalent to `long double`, which is 64 bit wide. The characteristics of floating-point numbers, described in 5.6 <float.h> Floating-Point Characteristics, change with the use of this option, as indicated.

MPLAB XC8 For AVR Functions With Pointers Arguments

The MPLAB XC8 Compiler for AVR can store objects qualified `const` into program memory (rather than data memory) and allow pointers to `const` types to access such objects. The feature is only available for some devices and is enabled using the `-mconst-data-in-progmem` option. When enabled, pointers to a `const`-qualified type are treated differently to similar pointers without the `const` qualifier, so for example the addresses held by pointers of type `char *` and `const char *` are interpreted as belonging to different address spaces.

To accommodate this feature, a different library is linked in when the feature is enabled, and the type of any library function argument or return value that is a pointer to a non-`const`-qualified type becomes a `const`-qualified version of that type. For example, with the `-mconst-data-in-progmem` option specified, the prototype for the `strtof` function, which is specified in the C standard as:

```
float strtof(const char * __restrict, char ** __restrict);
```

becomes:

```
float strtof(const char * __restrict, const char ** __restrict);
```

This change might affect how your program should be defined.

4. Library Example Code

Example code is shown for most library functions. These examples illustrate how the functions can be called and might indicate other aspects of their usage, but they are not necessarily complete nor practical. The example code might be encoded differently on different target devices and might operate differently at runtime.

The examples can be run in a simulator, such as that in the MPLAB X IDE. Alternatively, they can be run on hardware, but they will require modification for the device and hardware setup that you are using. The device configuration bits, which are necessary for code to execute on hardware, are not shown in the examples, as these differ from device to device. If you are using the MPLAB X IDE, take advantage of its built-in tools to generate the code required to initialize the configuration bits, and which can be copied and pasted into your project's source. See the *MPLAB® X IDE User's Guide* for a description and use of the Configuration Bits window.

Many of the library examples use the `printf()` function. Code in addition to that shown in the examples might be necessary to have this function print to a peripheral of your choice.

When the examples are run in the MPLAB X IDE simulator, the `printf()` function can be made to have its output sent to a USART (for some devices, this peripheral is called a UART) and shown in a window. To do this, you must:

- Enable the USART IO feature in the MPLAB X IDE (the IDE might offer a choice of USARTs).
- Ensure that your project code initializes and enables the same USART used by the IDE.
- Ensure that your project code defines a 'print-byte' function that sends one byte to the relevant USART.
- Ensure that the `printf()` function will call the relevant print-byte function.

Some compilers might provide generic code that will already implement the USART initialization and print-byte functions, as itemized above. For other tools, you can often use the Microchip Code Configurator (MCC) to generate this code. Check to see if the MCC is available for your target device. Even if it is not, you may be able to adapt the MCC output for a similar device. Typically, the default USART settings in the MCC will work with the simulator, but these may not suit your final application. Once the USART is configured, you may use any of the standard IO library functions that write to `stdout`, in addition to `printf()`.

Some library examples might also use the `scanf()` function. Code in addition to that shown in the examples might be necessary to have this function read a peripheral of your choice.

When the examples are run in the MPLAB X IDE simulator, the `scanf()` function can be made to read from a USART that is taking input from a text file. To do this, you must:

- Enable the USART IO feature in the MPLAB X IDE.
- Ensure that your project code initializes and enables the same USART used by the IDE.
- Ensure that your project code defines a 'read-byte' function that reads one byte from the relevant USART.
- Ensure that the `scanf()` function will call the relevant read-byte function.
- Provide a text file containing the required input, and have the content of this file passed by register injection to the receive register associated with the USART used by the IDE.

Some compilers might provide generic code that will already implement the USART initialization and read-byte functions, as itemized above. For other tools, you can often use the Microchip Code Configurator (MCC) to generate this code. Typically, the default USART settings that MCC uses will work with the simulator, but these may not suit your final application. Once the USART is configured, you may use any of the standard IO library functions that read from `stdin`, in addition to `scanf()`.

For further information about the MPLAB X IDE, see the *MPLAB® X IDE User's Guide*; for the MCC tool, see the *MPLAB® Code Configurator v3.xx User's Guide*.

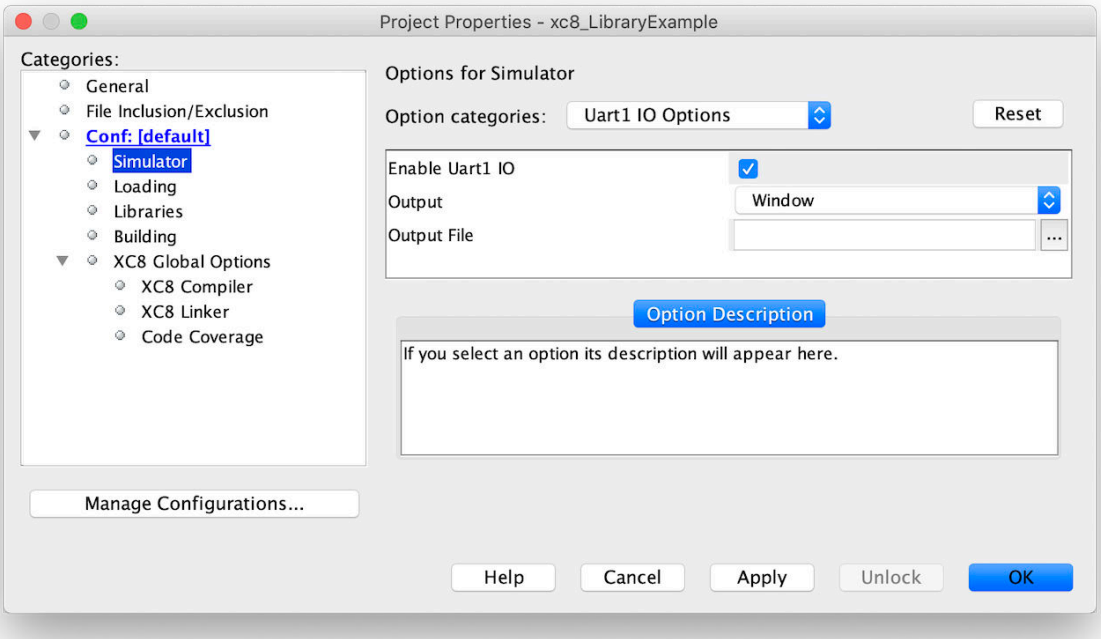
Compiler-specific implementations of the above are discussed in more detail in the following sections.

4.1 Example code for 8-bit PIC MCUs

If you want to run example code in the MPLAB X IDE simulator, the IDE's UART IO feature, available for most devices, allows you to view output from the `stdout` stream. Once properly configured, the output of `printf()` and other functions writing to `stdout` can then be viewed from the IDE when the program is run in the simulator.

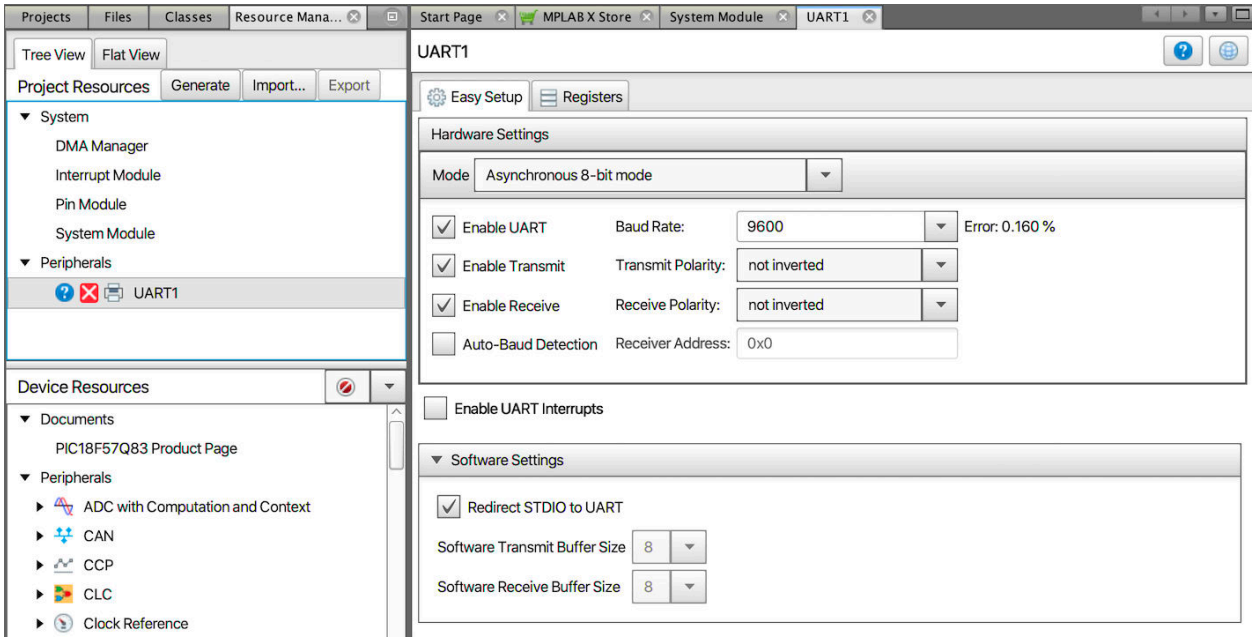
When available, this feature is enabled using the **Enable Uartx IO** checkbox in the **Project properties > Simulator > Uartx IO Options** dialog (shown below). You might have a choice of UARTs. Choose the UART that your code will write to. Output can be displayed in a window in the IDE or sent to a file on your host machine, based on the selections you make in the dialog.

Figure 4-1. Enabling the UART IO feature in the MPLAB X IDE



The UART initialization and print-byte functions generated by the MCC can be used by the simulator. If you enable the **Redirect STDIO to UART** checkbox in the **UARTx** pane (in the lower part of the pane shown below), MCC will ensure that the print-byte function used by `printf()` calls the MCC-generated function that sends data to the UART. The default communication settings should work in the simulator, but these may need to be changed if the UART is to be used on hardware.

Figure 4-2. Initializing the UART using MCC



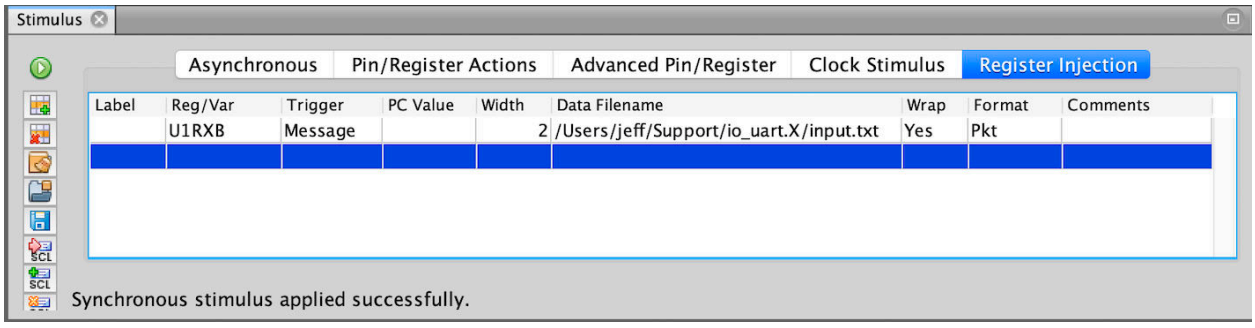
The same UART IO feature allows you to read input from the `stdin` stream. Once properly configured, `scanf()` and other functions reading from `stdin` can then take input from a file when the program is run in the simulator.

If you want to run example code that uses `scanf()` in the MPLAB X IDE simulator, you will need to set up a UART, as above. You can use the same UART for input as you do for output using `printf()`. Additionally, provide the input characters to the program by creating and adding to your project an empty file, using a suitable extension, such as `.txt`. Open this file in the editor and add the characters that you want to send to the program inside quotes, e.g. `"here is my input"` Commands that control how the input should be read can also be added to this file, as described in the *MPLAB® X IDE User's Guide*.

Next, open the **Window > Simulator > Stimulus** dialog. Select the **Register Injection** tab. Start typing the name of the receive register used by the USART in the **Reg/Var** field to see a pop-up list of matching register names. This register name will vary from device to device. Refer to your device data sheet, although you can often look to see the name of the register being returned by the `UARTx_Read()` function created by MCC. Ensure the **Trigger** field is set to **Message**, and the **Format** field is set to **Pkt**. In the **Data Filename** field, enter the name and path of the text file that contains your input. Finally, ensure that the stimulus has been applied by clicking the circular green button at the top left and checking the message shown at the bottom of the dialog. This button toggles the application of the stimulus.

The following image shows the stimulus correctly set up for the UART1 on a PIC18 device that uses a receive register called U1RXB.

Figure 4-3. Specifying a file to be read by the UART



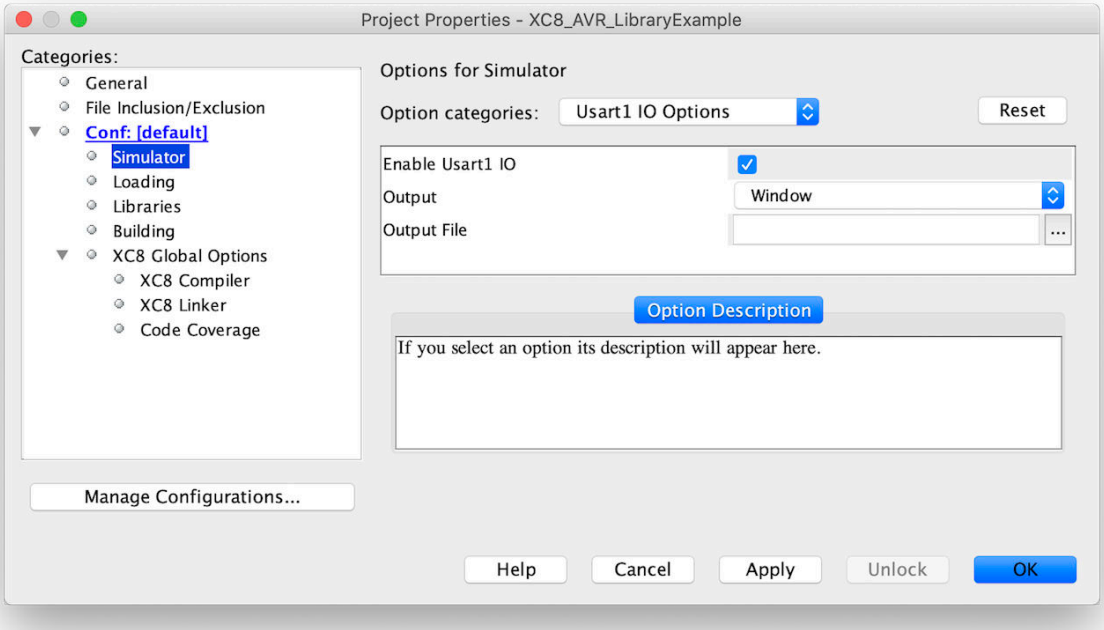
With the stimulus correct configured, `scanf()` and other functions reading from `stdin` will take input from the file specified.

4.2 Example code for 8-bit AVR MCUs

The UART IO feature in the MPLAB X IDE allows you to view output from the `stdout` stream. Once properly configured, the output of `printf()` and other functions can then be viewed from the IDE when the program is run in the simulator.

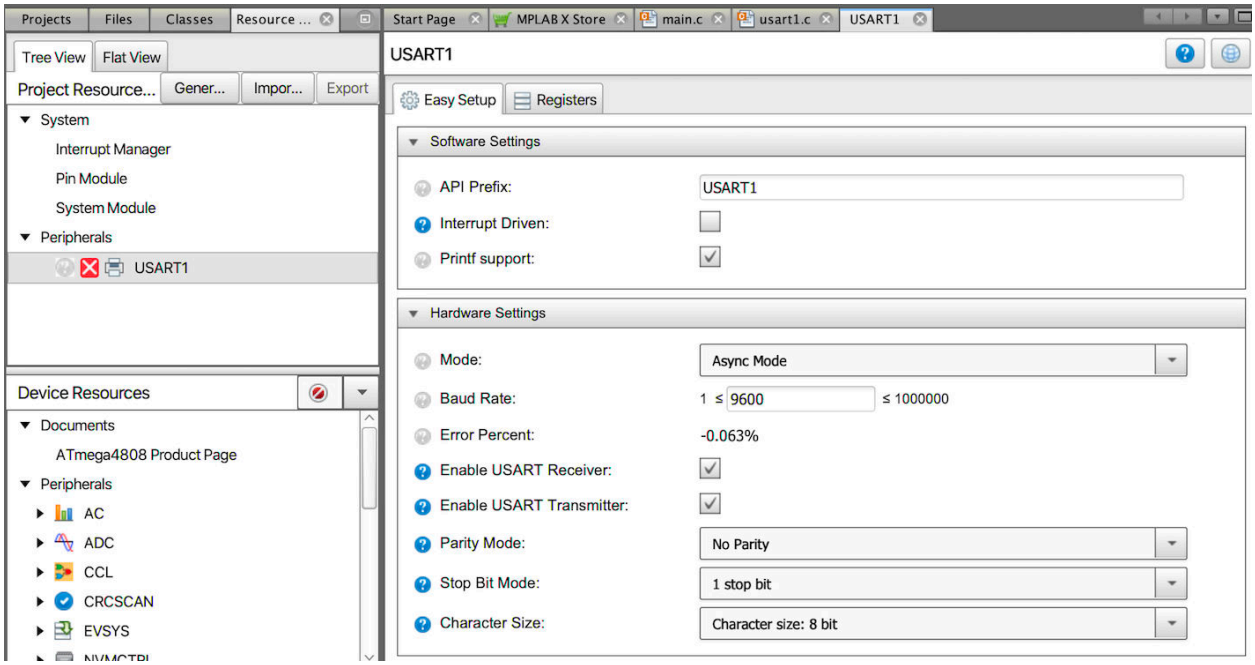
This feature is enabled from **Project properties > Simulator > USARTx IO Options**. You might have a choice of USARTs. Choose the USART that your code will write to. Output can be displayed in a window in the IDE or sent to a file on your host machine, based on the selections you make in the dialog.

Figure 4-4. Enabling the USART IO feature in the MPLAB X IDE



The USART initialization and print-byte function generated by the MCC will allow `printf()` to work in the simulator. If you enable the **Printf support** checkbox in the USART dialog, MCC will generate the necessary mapping via the `FDEV_SETUP_STREAM()` macro to ensure that `printf()` calls the correct print-byte function. The default communication settings should work in the simulator, but these may need to be changed if the USART is to be used on hardware.

Figure 4-5. Initializing the USART using MCC

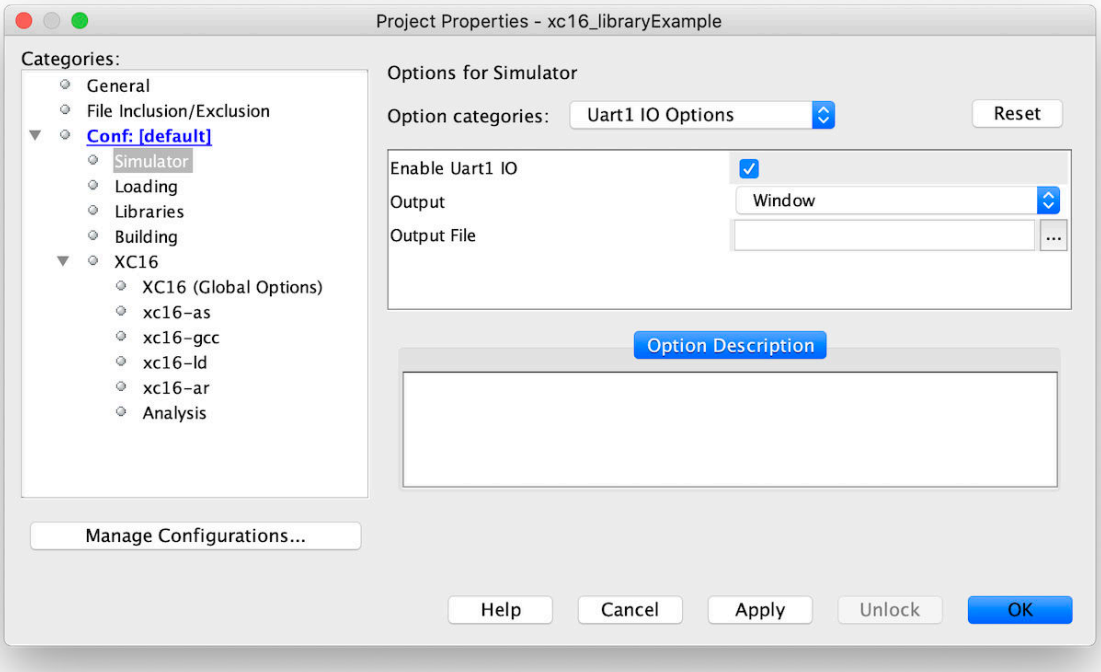


4.3 Example code for 16-bit PIC MCUs and dsPIC® DSCs

If you want to run example code in the MPLAB X IDE simulator, the IDE's UART IO feature, available for most devices, allows you to view output from the `stdout` stream. Once properly configured, the output of `printf()` and other functions writing to `stdout` can then be viewed from the IDE when the program is run in the simulator.

When available, this feature is enabled using the **Enable Uartx IO** checkbox in the **Project properties > Simulator > Uartx IO Options** dialog (shown below). You might have a choice of UARTs. Choose the UART that your code will write to. Output can be displayed in a window in the IDE or sent to a file on your host machine, based on the selections you make in the dialog.

Figure 4-6. Enabling the UART IO feature in the MPLAB X IDE



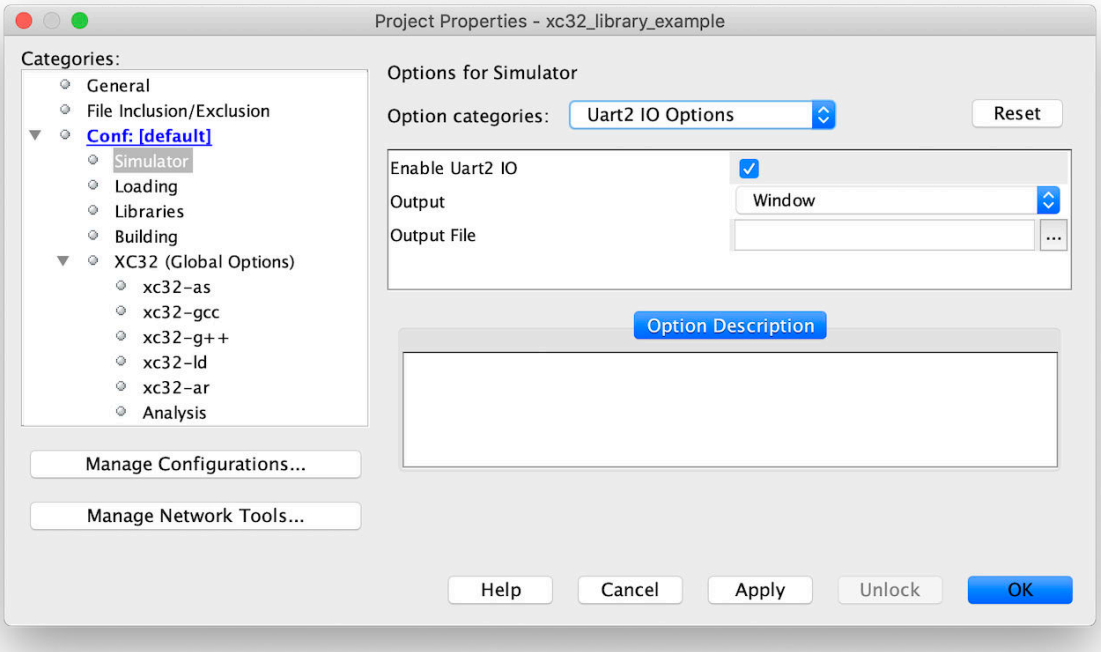
The IO helper functions provided by the MPLAB XC16 compiler will work with the simulator's USART IO feature without any modification; however, remember that you might need to modify these helper functions to suit applications running on your own hardware.

4.4 Example code for 32-bit PIC MCUs

If you want to run example code in the MPLAB X IDE simulator, the IDE's UART IO feature, available for PIC32 devices, allows you to view output from the `stdout` stream. Once properly configured, the output of `printf()` and other functions writing to `stdout` can then be viewed from the IDE when the program is run in the simulator. This IDE feature is not available for SAM MCUs.

When available, this feature is enabled using the **Enable Uart*x* IO** checkbox in the **Project properties > Simulator > Uart*x* IO Options** dialog (shown below). You might have a choice of UARTs. Choose the UART that your code will write to. Output can be displayed in a window in the IDE or sent to a file on your host machine, based on the selections you make in the dialog.

Figure 4-7. Enabling the UART IO feature in the MPLAB X IDE



When using 32-bit PIC devices, the IO helper functions provided by the MPLAB XC32 compiler will work with the simulator's UART IO feature without any modification; however, remember that you might need to modify these helper functions to suit applications running on your own hardware. All that is required in your project is code to select which UART output should be sent to.

The following code shows UART2 being selected for output (using the `__XC_UART` variable) before text is printed to the IDE's UART IO feature.

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

int main(void) {

    __XC_UART = 2; // set UART 2 for the simulator
    printf("The sine of 0.5 is %f\n", sin(0.5));

    return (EXIT_SUCCESS);
}
```

5. Standard C Library Headers

The following are the standard C99 language headers that make up the Microchip Unified Standard Library.

5.1 <assert.h> Diagnostics

The content of the header file `assert.h` is useful for debugging logic errors in programs. By using these features in critical locations where certain conditions should be true, the logic of the program may be tested.

5.1.1 NDEBUG Macro

A macro used by `<assert.h>` to specify the operation of certain debugging features.

Value

This macro is *not* defined by `<assert.h>` and must be defined by the user.

Remarks

`NDEBUG` must be defined as a macro by the user if and when required. Use the `#define` preprocessor directive to define the macro. It does not need to define any replacement string.

If the macro is defined at the point where `<assert.h>` is included, it disables certain debugging functionality of the header content supplied by that inclusion, in particular, the `assert()` macro (5.1.2 assert Macro). The `NDEBUG` macro may be undefined to allow `<assert.h>` to be included in subsequent code and with the full debugging functionality provided.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
int main(void)
{
  int a;

  a = 4;
#define NDEBUG        /* negate debugging - disable assert() functionality */
#include <assert.h>
  assert(a == 6);     /* no action performed, even though expression is false */

#undef NDEBUG         /* ensure assert() is active */
#include <assert.h>
  a = 7;
  assert(a == 7);     /* true - no action performed */
  assert(a == 8);     /* false - print message and abort */
}
```

Example Output

```
sampassert.c:14 a == 8 -- assertion failed
ABRT
```

5.1.2 assert Macro

If the argument is false, an assertion failure message is printed and the program is aborted.

Include

`<assert.h>`

Prototype

`void assert(`*`scalar`* **`expression`**`);`

Argument

expression	The expression to test.

Remarks

The expression evaluates to zero or non-zero. If zero, the assertion fails, a message is printed and `abort()` is called, terminating program execution. In the case of MPLAB XC8, the message is printed to `stdout`; for other compilers, this is printed to `stderr`.

The message includes the source file name (`__FILE__`), the source line number (`__LINE__`), and the expression being evaluated.

If the macro `NDEBUG` (see 5.1.1 NDEBUG Macro) is defined at the point where `<assert.h>` is included, the `assert()` macro will evaluate to a void expression, `((void)0)`, and not print any assertion message, nor abort program execution. Inclusion of `<assert.h>` can occur multiple times, even in the same source file, and the action of the `assert()` macro for each inclusion will be based on the state of `NDEBUG` at the point at which that inclusion takes place.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
int main(void)
{
  int a;

  a = 4;
#define NDEBUG        /* negate debugging - disable assert() functionality */
#include <assert.h>
  assert(a == 6);     /* no action performed, even though expression is false */

#undef NDEBUG         /* ensure assert() is active */
#include <assert.h>
  a = 7;
  assert(a == 7);     /* true - no action performed */
  assert(a == 8);     /* false - print message and abort */
}
```

Example Output

```
sampassert.c:14 a == 8 -- assertion failed
ABRT
```

5.1.3 __conditional_software_breakpoint Macro

Trigger a software breakpoint if the argument is false.

Include

`<assert.h>`

Prototype

`void __conditional_software_breakpoint(`*scalar* **expression**`);`

Argument

expression	The expression to test.

Remarks

The expression evaluates to zero or non-zero. If non-zero, the macro evaluates to the do-nothing expression, `((void)0)`. If zero, a software breakpoint is triggered and execution will be suspended. If the target device does not support software breakpoints, a compiler error is triggered.

If the macro `NDEBUG` (see 5.1.1 NDEBUG Macro) is defined or the macro `__DEBUG` is not defined at the point where `<assert.h>` is included, the `__conditional_software_breakpoint()` macro will evaluate to a void expression, `((void)0)`, and not suspend program execution. Inclusion of `<assert.h>` can occur multiple times,

even in the same source file, and the action of the `__conditional_software_breakpoint()` macro for each inclusion will be based on the state of `NDEBUG` and `__DEBUG` at the point at which that inclusion takes place. Note that the `__DEBUG` macro is automatically set by the MPLAB X IDE when performing a debug (as opposed to a production) build.

Example

```
#include <xc.h>

int main(void)
{
  int a;

  a = 4;
#define NDEBUG        /* negate debugging - disable __conditional_software_breakpoint()
functionality */
#include <assert.h>
  __conditional_software_breakpoint(a == 6);     /* no action performed, even though
expression is false */

#undef NDEBUG         /* ensure __conditional_software_breakpoint() is active */
#include <assert.h>
  a = 7;
  __conditional_software_breakpoint(a == 7);     /* true - no action performed */
  __conditional_software_breakpoint(a == 8);     /* false - suspend execution for debug
builds */
}
```

5.2 <complex.h> Complex Arithmetic

The header file `complex.h` implements macros and functions that support complex number arithmetic.

Attention: This header is implemented only by MPLAB XC32 C compilers.

5.2.1 complex Macro

A more readable representation for complex number types.

Attention: This macro is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Definition

This macro expands to the `_Complex` type specifier.

5.2.2 _Complex_I Macro

A representation of the imaginary unit, i.

Attention: This macro is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Definition

This macro expands to a constant expression that has a `const float _Complex` type and the value of the imaginary unit, *i*.

5.2.3 imaginary Macro

A more readable representation for imaginary number types.

Attention: This macro is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Definition

This macro expands to the `_Imaginary` type specifier.

5.2.4 _Imaginary_I Macro

A representation of the imaginary unit, *i*.

Attention: This macro is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Definition

This macro expands to a constant expression that has a `const float _Imaginary` type and the value of the imaginary unit, *i*.

5.2.5 I Macro

A representation of the imaginary unit, *i*.

Attention: This macro is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Definition

When using MPLAB XC32, this macro expands to `_Imaginary_I`, which expands to a constant expression that has a `const float _Imaginary` type and the value of the imaginary unit, *i*.

5.2.6 cabs Function

Calculates the complex absolute value of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex cabs(double complex z);
```

Argument

z value for which to return the complex absolute value

Return Value

Returns the complex absolute value (also known as the magnitude, modulus, or norm) of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = cabs(x);
  printf("The complex absolute value of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex absolute value of -2.000000 + 1.000000i is 2.236068 + 0.000000i
```

5.2.7 cabsf Function

Calculates the complex absolute value of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex cabsf(float complex z);
```

Argument

z value for which to return the complex absolute value

Return Value

Returns the complex absolute value (also known as the magnitude, modulus, or norm) of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = cabsf(x);
  printf("The complex absolute value of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The complex absolute value of -2.000000 + 1.000000i is 2.236068 + 0.000000i
```

5.2.8 cabsl Function

Calculates the complex absolute value of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

`long double complex cabsl(long double complex z);`

Argument

z value for which to return the complex absolute value

Return Value

Returns the complex absolute value (also known as the magnitude, modulus, or norm) of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = cabsl(x);
  printf("The complex absolute value of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex absolute value of -2.000000 + 1.000000i is 2.236068 + 0.000000i
```

5.2.9 cacos Function

Calculates the complex arc cosine function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex cacos(double complex z);
```

Argument

z value for which to return the complex arc cosine

Return Value

Returns the complex arc cosine of `z` with no bound along the imaginary axis, in the interval $[0, \pi]$ along the real axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 0.0*I;
  y = cacos(x);
  printf("The complex arccosine of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex arccosine of -2.000000 + 0.000000i is 3.141593 + -1.316958i
```

5.2.10 cacosf Function

Calculates the complex arc cosine function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex cacosf(float complex z);
```

Argument

z value for which to return the complex arc cosine

Return Value

Returns the complex arc cosine of `z` with no bound along the imaginary axis, in the interval $[0, \pi]$ along the real axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 0.0*I;
  y = cacosf(x);
  printf("The complex arccosine of %f + %fi is %f + %fi\n", crealf(x), cimagf(x), crealf(y),
cimagf(y));
}
```

Example Output

```
The complex arccosine of -2.000000 + 0.000000i is 3.141593 + -1.316958i
```

5.2.11 cacosl Function

Calculates the complex arc cosine function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex cacosl(long double complex z);
```

Argument

z value for which to return the complex arc cosine

Return Value

Returns the complex arc cosine of `z` with no bound along the imaginary axis, in the interval $[0, \pi]$ along the real axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 0.0*I;
  y = cacosl(x);
```

```
  printf("The complex arccosine of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex arccosine of -2.000000 + 0.000000i is 3.141593 + -1.316958i
```

5.2.12 cacosh Function

Calculates the complex arc hyperbolic cosine function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex cacosh(double complex z);
```

Argument

z value for which to return the complex arc hyperbolic cosine

Return Value

Returns the complex arc hyperbolic cosine of `z` with non-negative values along the real axis, in the interval $[-i\pi, +i\pi]$ along the imaginary axis, and with a branch cut at values less than 1 along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 0.0*I;
  y = cacosh(x);
  printf("The complex arc hyperbolic cosine of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex arc hyperbolic cosine of -2.000000 + 1.000000i is 1.469352 + 2.634236i
```

5.2.13 cacoshf Function

Calculates the complex arc hyperbolic cosine function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex cacoshf(float complex z);
```

Argument

z value for which to return the complex arc hyperbolic cosine

Return Value

Returns the complex arc hyperbolic cosine of `z` with non-negative values along the real axis, in the interval $[-i\pi, +i\pi]$ along the imaginary axis, and with a branch cut at values less than 1 along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 0.0*I;
  y = cacoshf(x);
  printf("The complex arc hyperbolic cosine of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The complex arc hyperbolic cosine of -2.000000 + 1.000000i is 1.469352 + 2.634236i
```

5.2.14 cacoshl Function

Calculates the complex arc hyperbolic cosine function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex cacoshl(long double complex z);
```

Argument

z value for which to return the complex arc hyperbolic cosine

Return Value

Returns the complex arc hyperbolic cosine of `z` with non-negative values along the real axis, in the interval $[-i\pi, +i\pi]$ along the imaginary axis, and with a branch cut at values less than 1 along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 0.0*I;
  y = cacoshl(x);
  printf("The complex arc hyperbolic cosine of %Lf + %Lfi is %Lf + %Lfi\n", creall(x),
cimagl(x), creall(y), cimagl(y));
}
```

Example Output

```
The complex arc hyperbolic cosine of -2.000000 + 1.000000i is 1.469352 + 2.634236i
```

5.2.15 carg Function

Calculates the argument of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex carg(double complex z);
```

Argument

z value for which to return the argument

Return Value

Returns the argument (or phase angle) of `z` in the interval $[-\pi, +\pi]$ and with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = carg(x);
  printf("The complex argument of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex argument of -2.000000 + 1.000000i is 2.677945 + 0.000000i
```

5.2.16 cargf Function

Calculates the argument of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex cargf(float complex z);
```

Argument

z value for which to return the argument

Return Value

Returns the argument (or phase angle) of `z` in the interval $[-\pi, +\pi]$ and with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = cargf(x);
  printf("The complex argument of %f + %fi is %f + %fi\n", crealf(x), cimagf(x), crealf(y),
cimagf(y));
}
```

Example Output

```
The complex argument of -2.000000 + 1.000000i is 2.677945 + 0.000000i
```

5.2.17 cargl Function

Calculates the argument of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex cargl(long double complex z);
```

Argument

z value for which to return the argument

Return Value

Returns the argument (or phase angle) of `z` in the interval $[-\pi, +\pi]$ and with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = cargl(x);
  printf("The complex argument of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex argument of -2.000000 + 1.000000i is 2.677945 + 0.000000i
```

5.2.18 casin Function

Calculates the complex arc sine function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex casin(double complex z);
```

Argument

z value for which to return the complex arc sine

Return Value

Returns the complex arc sine of `z` with no bound along the imaginary axis, in the interval $[-\pi/2, +\pi/2]$ along the real axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 0.0*I;
  y = casin(x);
```

```
  printf("The complex arcsine of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex arcsin of -2.000000 + 0.000000i is -1.570796 + 1.316958i
```

5.2.19 casinf Function

Calculates the complex arc sine function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex casinf(float complex z);
```

Argument

z value for which to return the complex arc sine

Return Value

Returns the complex arc sine of `z` with no bound along the imaginary axis, in the interval $[-\pi/2, +\pi/2]$ along the real axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 0.0*I;
  y = casinf(x);
  printf("The complex arcsine of %f + %fi is %f + %fi\n", crealf(x), cimagf(x), crealf(y),
cimagf(y));
}
```

Example Output

```
The complex arcsin of -2.000000 + 0.000000i is -1.570796 + 1.316958i
```

5.2.20 casinl Function

Calculates the complex arc sine function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex casinl(long double complex z);
```

Argument

z value for which to return the complex arc sine

Return Value

Returns the complex arc sine of `z` with no bound along the imaginary axis, in the interval $[-\pi/2, +\pi/2]$ along the real axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 0.0*I;
  y = casinl(x);
  printf("The complex arcsine of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex arcsin of -2.000000 + 0.000000i is -1.570796 + 1.316958i
```

5.2.21 casinh Function

Calculates the complex arc hyperbolic sine function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex casinh(double complex z);
```

Argument

z value for which to return the complex arc hyperbolic sine

Return Value

Returns the complex arc hyperbolic sine of `z` with no bounds on the real axis, in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis, and with branch cuts outside the interval $[-i, +i]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 0.0*I;
  y = casinh(x);
  printf("The complex arc hyperbolic cosine of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex arc hyperbolic sine of -2.000000 + 1.000000i is -1.528571 + 0.427079i
```

5.2.22 casinhf Function

Calculates the complex arc hyperbolic sine function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

`float complex casinhf(float complex z);`

Argument

z value for which to return the complex arc hyperbolic sine

Return Value

Returns the complex arc hyperbolic sine of `z` with no bounds on the real axis, in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis, and with branch cuts outside the interval $[-i, +i]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 0.0*I;
  y = casinhf(x);
  printf("The complex arc hyperbolic cosine of %f + %fi is %f + %fi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex arc hyperbolic sine of -2.000000 + 1.000000i is -1.528571 + 0.427079i
```

5.2.23 casinhl Function

Calculates the complex arc hyperbolic sine function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex casinhl(long double complex z);
```

Argument

z value for which to return the complex arc hyperbolic sine

Return Value

Returns the complex arc hyperbolic sine of `z` with no bounds on the real axis, in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis, and with branch cuts outside the interval $[-i, +i]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 0.0*I;
  y = casinhl(x);
  printf("The complex arc hyperbolic cosine of %Lf + %Lfi is %Lf + %Lfi\n", creall(x),
cimagl(x), creall(y), cimagl(y));
}
```

Example Output

```
The complex arc hyperbolic sine of -2.000000 + 1.000000i is -1.528571 + 0.427079i
```

5.2.24 catan Function

Calculates the complex arc tangent function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex catan(double complex z);
```

Argument

z value for which to return the complex arc tangent

Return Value

Returns the complex arc tangent of z with no bound along the imaginary axis, in the interval $[-\pi/2, +\pi/2]$ along the real axis, and with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 0.0*I;
  y = catan(x);
  printf("The complex arctan of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex arctan of -2.000000 + 0.000000i is -1.107149 + 0.000000i
```

5.2.25 catanf Function

Calculates the complex arc tangent function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex catanf(float complex z);
```

Argument

z value for which to return the complex arc tangent

Return Value

Returns the complex arc tangent of z with no bound along the imaginary axis, in the interval $[-\pi/2, +\pi/2]$ along the real axis, and with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 0.0*I;
  y = catanf(x);
```

```
  printf("The complex arctan of %f + %fi is %f + %fi\n", crealf(x), cimagf(x), crealf(y),
cimagf(y));
}
```

Example Output

```
The complex arctan of -2.000000 + 0.000000i is -1.107149 + 0.000000i
```

5.2.26 catanl Function

Calculates the complex arc tangent function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex catanl(long double complex z);
```

Argument

z value for which to return the complex arc tangent

Return Value

Returns the complex arc tangent of `z` with no bound along the imaginary axis, in the interval $[-\pi/2, +\pi/2]$ along the real axis, and with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 0.0*I;
  y = catanl(x);
  printf("The complex arctan of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x), creall(y),
cimagl(y));
}
```

Example Output

```
The complex arctan of -2.000000 + 0.000000i is -1.107149 + 0.000000i
```

5.2.27 catanh Function

Calculates the complex arc hyperbolic tangent function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex catanh(double complex z);
```

Argument

z value for which to return the complex arc hyperbolic tangent

Return Value

Returns the complex arc hyperbolic tangent of `z` with no bounds on the real axis, in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 0.0*I;
  y = catanh(x);
  printf("The complex arc hyperbolic tangent of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex arc hyperbolic tangent of -2.000000 + 1.000000i is -0.402359 + 1.338973i
```

5.2.28 catanhf Function

Calculates the complex arc hyperbolic tangent function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex catanhf(float complex z);
```

Argument

z value for which to return the complex arc hyperbolic tangent

Return Value

Returns the complex arc hyperbolic tangent of `z` with no bounds on the real axis, in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 0.0*I;
  y = catanhf(x);
  printf("The complex arc hyperbolic tangent of %f + %fi is %f + %fi\n", crealf(x),
cimagf(x), crealf(y), cimagf(y));
}
```

Example Output

```
The complex arc hyperbolic tangent of -2.000000 + 1.000000i is -0.402359 + 1.338973i
```

5.2.29 catanhl Function

Calculates the complex arc hyperbolic tangent function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex catanhl(long double complex z);
```

Argument

z value for which to return the complex arc hyperbolic tangent

Return Value

Returns the complex arc hyperbolic tangent of `z` with no bounds on the real axis, in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis, and with branch cuts outside the interval $[-1, +1]$ along the real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 0.0*I;
  y = catanhl(x);
  printf("The complex arc hyperbolic tangent of %Lf + %Lfi is %Lf + %Lfi\n", creall(x),
cimagl(x), creall(y), cimagl(y));
}
```

Example Output

```
The complex arc hyperbolic tangent of -2.000000 + 1.000000i is -0.402359 + 1.338973i
```

5.2.30 ccos Function

Calculates the complex cosine function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex ccos(double complex z);
```

Argument

z value for which to return the complex cosine

Return Value

Returns the complex cosine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = ccos(x);
  printf("The complex cosine of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex cosine of -2.000000 + 1.000000i is -0.642148 + 1.068607i
```

5.2.31 ccosf Function

Calculates the complex cosine function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex ccosf(float complex z);
```

Argument

z value for which to return the complex cosine

Return Value

Returns the complex cosine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = ccosf(x);
  printf("The complex cosine of %f + %fi is %f + %fi\n", crealf(x), cimagf(x), crealf(y),
cimagf(y));
}
```

Example Output

```
The complex cosine of -2.000000 + 1.000000i is -0.642148 + 1.068607i
```

5.2.32 ccosl Function

Calculates the complex cosine function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex ccosl(long double complex z);
```

Argument

z value for which to return the complex cosine

Return Value

Returns the complex cosine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = ccosl(x);
  printf("The complex cosine of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x), creall(y),
cimagl(y));
}
```

Example Output

```
The complex cosine of -2.000000 + 1.000000i is -0.642148 + 1.068607i
```

5.2.33 ccosh Function

Calculates the complex hyperbolic cosine function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex ccosh(double complex z);
```

Argument

z value for which to return the complex hyperbolic cosine

Return Value

Returns the complex hyperbolic cosine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = ccosh(x);
  printf("The complex hyperbolic cosine of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex hyperbolic cosine of -2.000000 + 1.000000i is 2.032723 + -3.051898i
```

5.2.34 ccoshf Function

Calculates the complex hyperbolic cosine function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex ccoshf(float complex z);
```

Argument

z value for which to return the complex hyperbolic cosine

Return Value

Returns the complex hyperbolic cosine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = ccoshf(x);
  printf("The complex hyperbolic cosine of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The complex hyperbolic cosine of -2.000000 + 1.000000i is 2.032723 + -3.051898i
```

5.2.35 ccoshl Function

Calculates the complex hyperbolic cosine function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex ccoshl(long double complex z);
```

Argument

z value for which to return the complex hyperbolic cosine

Return Value

Returns the complex hyperbolic cosine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = ccoshl(x);
```

```
  printf("The complex hyperbolic cosine of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex hyperbolic cosine of -2.000000 + 1.000000i is 2.032723 + -3.051898i
```

5.2.36 cexp Function

Calculates the complex base-*e* exponential of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex cexp(double complex z);
```

Argument

z value for which to return the complex base-*e* exponential

Return Value

Returns the complex base-*e* exponential of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = cexp(x);
  printf("The complex base-e exponential  of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex base-e exponential of -2.000000 + 1.000000i is 0.073122 + 0.113881i
```

5.2.37 cexpf Function

Calculates the complex base-*e* exponential of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex cexpf(float complex z);
```

Argument

z value for which to return the complex base-*e* exponential

Return Value

Returns the complex base-*e* exponential of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = cexpf(x);
  printf("The complex base-e exponential  of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The complex base-e exponential of -2.000000 + 1.000000i is 0.073122 + 0.113881i
```

5.2.38 cexpl Function

Calculates the complex base-*e* exponential of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex cexpl(long double complex z);
```

Argument

z value for which to return the complex base-*e* exponential

Return Value

Returns the complex base-*e* exponential of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
```

```
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = cexpl(x);
  printf("The complex base-e exponential  of %Lf + %Lfi is %Lf + %Lfi\n", creall(x),
cimagl(x), creall(y), cimagl(y));
}
```

Example Output

```
The complex base-e exponential of -2.000000 + 1.000000i is 0.073122 + 0.113881i
```

5.2.39 cimag Function

Calculates the imaginary part of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex cimag(double complex z);
```

Argument

z value for which to return the imaginary part

Return Value

Returns the imaginary part of the double precision complex value `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x;

  x = -2.0 + 1.0*I;
  printf("The imaginary part of %f + %fi is %f\n", creal(x), cimag(x), cimag(x));
}
```

Example Output

```
The imaginary part of -2.000000 + 1.000000i is 1.000000
```

5.2.40 cimagf Function

Calculates the imaginary part of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex cimagf(float complex z);
```

Argument

z value for which to return the imaginary part

Return Value

Returns the imaginary part of the single precision complex value `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x;

  x = -2.0 + 1.0*I;
  printf("The imaginary part of %f + %fi is %f\n", crealf(x), cimagf(x), cimagf(x));
}
```

Example Output

```
The imaginary part of -2.000000 + 1.000000i is 1.000000
```

5.2.41 cimagl Function

Calculates the imaginary part of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex cimagl(double complex z);
```

Argument

z value for which to return the imaginary part

Return Value

Returns the imaginary part of the double precision complex value `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x;
```

```
  x = -2.0 + 1.0*I;
  printf("The imaginary part of %Lf + %Lfi is %Lf\n", creall(x), cimagl(x), cimagl(x));
}
```

Example Output

```
The imaginary part of -2.000000 + 1.000000i is 1.000000
```

5.2.42 clog Function

Calculates the complex base-*e* logarithm of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex clog(double complex z);
```

Argument

z value for which to return the complex base-*e* logarithm

Return Value

Returns the complex base-*e* (natural) logarithm of `z`, with no bound along the real axis, in the interval $[-i\pi, +i\pi]$ along the imaginary axis, with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = clog(x);
  printf("The complex base-e logarithm of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex base-e logarithm of -2.000000 + 1.000000i is 0.804719 + 2.677945i
```

5.2.43 clogf Function

Calculates the complex base-*e* logarithm of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex clogf(float complex z);
```

Argument

z value for which to return the complex base-*e* logarithm

Return Value

Returns the complex base-*e* (natural) logarithm of `z`, with no bound along the real axis, in the interval $[-i\pi, +i\pi]$ along the imaginary axis, with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = clogf(x);
  printf("The complex base-e logarithm of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The complex base-e logarithm of -2.000000 + 1.000000i is 0.804719 + 2.677945i
```

5.2.44 clogl Function

Calculates the complex base-*e* logarithm of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex clogl(long double complex z);
```

Argument

z value for which to return the complex base-*e* logarithm

Return Value

Returns the complex base-*e* (natural) logarithm of `z`, with no bound along the real axis, in the interval $[-i\pi, +i\pi]$ along the imaginary axis, with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = clogl(x);
  printf("The complex base-e logarithm of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex base-e logarithm of -2.000000 + 1.000000i is 0.804719 + 2.677945i
```

5.2.45 conj Function

Calculates the complex conjugate of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex conj(double complex z);
```

Argument

z value for which to return the complex conjugate

Return Value

Returns the complex conjugate of `z`, being the value of the argument with the sign of the imaginary part reversed.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = conj(x);
  printf("The complex conjugate of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex conjugate of -2.000000 + 1.000000i is -2.000000 + -1.000000i
```

5.2.46 conjf Function

Calculates the complex conjugate of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex conjf(float complex z);
```

Argument

z value for which to return the complex conjugate

Return Value

Returns the complex conjugate of `z`, being the value of the argument with the sign of the imaginary part reversed.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = conjf(x);
  printf("The complex conjugate of %f + %fi is %f + %fi\n", crealf(x), cimagf(x), crealf(y),
cimagf(y));
}
```

Example Output

```
The complex conjugate of -2.000000 + 1.000000i is -2.000000 + -1.000000i
```

5.2.47 conjl Function

Calculates the complex conjugate of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex conjl(long double complex z);
```

Argument

z value for which to return the complex conjugate

Return Value

Returns the complex conjugate of `z`, being the value of the argument with the sign of the imaginary part reversed.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = conjl(x);
  printf("The complex conjugate of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex conjugate of -2.000000 + 1.000000i is -2.000000 + -1.000000i
```

5.2.48 cpow Function

Calculates the complex power function x^y for double precision complex values.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex cpow(double complex x, double complex y);
```

Arguments

x	the base value
y	the exponent, or power, value

Return Value

Returns the complex power function x^y, with a branch cut for the base value (`x`) along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y, z;
  x = -2.0 + 1.0*I;
  y = 0.2 - 1.3*I;
  z = cpow(x, y);
  printf("The complex value %f + %fi raised to the value %f + %fi is %f + %fi\n", creal(x),
cimag(x), creal(y), cimag(y), creal(z), cimag(z));
```

```
}
```

Example Output

```
The complex value -2.000000 + 1.000000i raised to the value 0.200000 + -1.300000i is
33.309896 + -18.656049i
```

5.2.49 cpowf Function

Calculates the complex power function x^y for single precision complex values.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex cpowf(float complex x, float complex y);
```

Arguments

x	the base value
y	the exponent, or power, value

Return Value

Returns the complex power function x^y, with a branch cut for the base value (x) along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y, z;
  x = -2.0 + 1.0*I;
  y = 0.2 - 1.3*I;
  z = cpowf(x, y);
  printf("The complex value %f + %fi raised to the value %f + %fi is %f + %fi\n", crealf(x),
cimagf(x), crealf(y), cimagf(y), crealf(z), cimagf(z));

}
```

Example Output

```
The complex value -2.000000 + 1.000000i raised to the value 0.200000 + -1.300000i is
33.309896 + -18.656049i
```

5.2.50 cpowl Function

Calculates the complex power function x^y for long double precision complex values.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex cpowl(long double complex x, long double complex y);
```

Arguments

x	the base value
y	the exponent, or power, value

Return Value

Returns the complex power function x^y, with a branch cut for the base value (`x`) along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y, z;
  x = -2.0 + 1.0*I;
  y = 0.2 - 1.3*I;
  z = cpowl(x, y);
  printf("The complex value %Lf + %Lfi raised to the value %Lf + %Lfi is %Lf + %Lfi\n",
creall(x), cimagl(x), creall(y), cimagl(y), creall(z), cimagl(z));

}
```

Example Output

```
The complex value -2.000000 + 1.000000i raised to the value 0.200000 + -1.300000i is
33.309896 + -18.656049i
```

5.2.51 cproj Function

Calculates the a projection of a double precision complex value onto the Riemann sphere.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex cproj(double complex z);
```

Argument

z	value for which to calculate the projection

Return Value

Returns the a projection of a double precision complex value, `z`, onto the Riemann sphere. The projection is such that `z` is returned, except if `z` is any complex infinity, in which case a positive infinity is returned.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x;

  x = -INFINITY + 1.0*I;
  y = cproj(x);
  printf("The Riemann plane projection of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The Riemann plane projection of -inf + 1.000000i is inf + 0.000000i
```

5.2.52 cprojf Function

Calculates the a projection of a single precision complex value onto the Riemann sphere.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex cprojf(float complex z);
```

Argument

z value for which to calculate the projection

Return Value

Returns the a projection of a single precision complex value, `z`, onto the Riemann sphere. The projection is such that `z` is returned, except if `z` is any complex infinity, in which case a positive infinity is returned.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x;

  x = -INFINITY + 1.0*I;
  y = cprojf(x);
  printf("The Riemann plane projection of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The Riemann plane projection of -inf + 1.000000i is inf + 0.000000i
```

5.2.53 cprojl Function

Calculates the a projection of a long double precision complex value onto the Riemann sphere.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex cprojl(long double complex z);
```

Argument

z value for which to calculate the projection

Return Value

Returns the a projection of a long double precision complex value, `z`, onto the Riemann sphere. The projection is such that `z` is returned, except if `z` is any complex infinity, in which case a positive infinity is returned.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x;

  x = -INFINITY + 1.0*I;
  y = cprojl(x);
  printf("The Riemann plane projection of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The Riemann plane projection of -inf + 1.000000i is inf + 0.000000i
```

5.2.54 creal Function

Calculates the real part of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex creal(double complex z);
```

Argument

z value for which to return the real part

Return Value

Returns the real part of the double precision complex value `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x;

  x = -2.0 + 1.0*I;
  printf("The real part of %f + %fi is %f\n", creal(x), cimag(x), creal(x));
}
```

Example Output

```
The real part of -2.000000 + 1.000000i is -2.000000
```

5.2.55 crealf Function

Calculates the real part of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex crealf(float complex z);
```

Argument

z value for which to return the real part

Return Value

Returns the real part of the single precision complex value `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x;

  x = -2.0 + 1.0*I;
```

```
  printf("The real part of %f + %fi is %f\n", crealf(x), cimagf(x), crealf(x));
}
```

Example Output

```
The real part of -2.000000 + 1.000000i is -2.000000
```

5.2.56 creall Function

Calculates the real part of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex creall(long double complex z);
```

Argument

z value for which to return the real part

Return Value

Returns the real part of the long double precision complex value `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x;

  x = -2.0 + 1.0*I;
  printf("The real part of %Lf + %Lfi is %Lf\n", creall(x), cimagl(x), creall(x));
}
```

Example Output

```
The real part of -2.000000 + 1.000000i is -2.000000
```

5.2.57 csin Function

Calculates the complex sine function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex csin(double complex z);
```

Argument

z value for which to return the complex sine

Return Value

Returns the complex sine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = csin(x);
  printf("The complex sine of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex sine of -2.000000 + 1.000000i is -1.403119 + -0.489056i
```

5.2.58 csinf Function

Calculates the complex sine function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex csinf(float complex z);
```

Argument

z value for which to return the complex sine

Return Value

Returns the complex sine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;
```

```
  x = -2.0 + 1.0*I;
  y = csinf(x);
  printf("The complex sine of %f + %fi is %f + %fi\n", crealf(x), cimagf(x), crealf(y),
cimagf(y));
}
```

Example Output

```
The complex sine of -2.000000 + 1.000000i is -1.403119 + -0.489056i
```

5.2.59 csinl Function

Calculates the complex sine function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex csinl(long double complex z);
```

Argument

z value for which to return the complex sine

Return Value

Returns the complex sine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = csinl(x);
  printf("The complex sine of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x), creall(y),
cimagl(y));
}
```

Example Output

```
The complex sine of -2.000000 + 1.000000i is -1.403119 + -0.489056i
```

5.2.60 csinh Function

Calculates the complex hyperbolic sine function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex csinh(double complex z);
```

Argument

z value for which to return the complex hyperbolic sine

Return Value

Returns the complex hyperbolic sine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = csinh(x);
  printf("The complex hyperbolic sine of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex hyperbolic sine of -2.000000 + 1.000000i is -1.959601 + 3.165779i
```

5.2.61 csinhf Function

Calculates the complex hyperbolic sine function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex csinhf(float complex z);
```

Argument

z value for which to return the complex hyperbolic sine

Return Value

Returns the complex hyperbolic sine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
```

```
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = csinhf(x);
  printf("The complex hyperbolic sine of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The complex hyperbolic sine of -2.000000 + 1.000000i is -1.959601 + 3.165779i
```

5.2.62 csinhl Function

Calculates the complex hyperbolic sine function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex csinhl(long double complex z);
```

Argument

z value for which to return the complex hyperbolic sine

Return Value

Returns the complex hyperbolic sine of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = csinhl(x);
  printf("The complex hyperbolic sine of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex hyperbolic sine of -2.000000 + 1.000000i is -1.959601 + 3.165779i
```

5.2.63 csqrt Function

Calculates the complex square root of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex csqrt(double complex z);
```

Argument

z value for which to return the complex square root

Return Value

Returns the complex square root of `z`, in the right half plane, including the imaginary axis, and with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = csqrt(x);
  printf("The complex square root of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex square root of -2.000000 + 1.000000i is 0.343561 + 1.455347i
```

5.2.64 csqrtf Function

Calculates the complex square root of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex csqrtf(float complex z);
```

Argument

z value for which to return the complex square root

Return Value

Returns the complex square root of `z`, in the right half plane, including the imaginary axis, and with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = csqrtf(x);
  printf("The complex square root of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The complex square root of -2.000000 + 1.000000i is 0.343561 + 1.455347i
```

5.2.65 csqrtl Function

Calculates the complex square root of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex csqrtl(long double complex z);
```

Argument

z value for which to return the complex square root

Return Value

Returns the complex square root of `z`, in the right half plane, including the imaginary axis, and with a branch cut along the negative real axis.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = csqrtl(x);
  printf("The complex square root of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex square root of -2.000000 + 1.000000i is 0.343561 + 1.455347i
```

5.2.66 ctan Function

Calculates the complex tangent function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
double complex ctan(double complex z);
```

Argument

z value for which to return the complex tangent

Return Value

Returns the complex tangent of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = ctan(x);
  printf("The complex tangent of %f + %fi is %f + %fi\n", creal(x), cimag(x), creal(y),
cimag(y));
}
```

Example Output

```
The complex tangent of -2.000000 + 1.000000i is 0.243458 + 1.166736i
```

5.2.67 ctanf Function

Calculates the complex tangent function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex ctanf(float complex z);
```

Argument

z value for which to return the complex tangent

Return Value

Returns the complex tangent of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = ctanf(x);
  printf("The complex tangent of %f + %fi is %f + %fi\n", crealf(x), cimagf(x), crealf(y),
cimagf(y));
}
```

Example Output

```
The complex tangent of -2.000000 + 1.000000i is 0.243458 + 1.166736i
```

5.2.68 ctanl Function

Calculates the complex tangent function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex ctanl(long double complex z);
```

Argument

z value for which to return the complex tangent

Return Value

Returns the complex tangent of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = ctanl(x);
```

```
  printf("The complex tangent of %Lf + %Lfi is %Lf + %Lfi\n", creall(x), cimagl(x),
creall(y), cimagl(y));
}
```

Example Output

```
The complex tangent of -2.000000 + 1.000000i is 0.243458 + 1.166736i
```

5.2.69 ctanh Function

Calculates the complex hyperbolic tangent function of a double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

`double complex ctanh(double complex z);`

Argument

z value for which to return the complex hyperbolic tangent

Return Value

Returns the complex hyperbolic tangent of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double complex x, y;

  x = -2.0 + 1.0*I;
  y = ctanh(x);
  printf("The complex hyperbolic tangent of %f + %fi is %f + %fi\n", creal(x), cimag(x),
creal(y), cimag(y));
}
```

Example Output

```
The complex hyperbolic tangent of -2.000000 + 1.000000i is -1.014794 + 0.033813i
```

5.2.70 ctanhf Function

Calculates the complex hyperbolic tangent function of a single precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
float complex ctanhf(float complex z);
```

Argument

z value for which to return the complex hyperbolic tangent

Return Value

Returns the complex hyperbolic tangent of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float complex x, y;

  x = -2.0 + 1.0*I;
  y = ctanhf(x);
  printf("The complex hyperbolic tangent of %f + %fi is %f + %fi\n", crealf(x), cimagf(x),
crealf(y), cimagf(y));
}
```

Example Output

```
The complex hyperbolic tangent of -2.000000 + 1.000000i is -1.014794 + 0.033813i
```

5.2.71 ctanhl Function

Calculates the complex hyperbolic tangent function of a long double precision complex value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<complex.h>`

Prototype

```
long double complex ctanhl(long double complex z);
```

Argument

z value for which to return the complex hyperbolic tangent

Return Value

Returns the complex hyperbolic tangent of `z`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
```

```
  long double complex x, y;

  x = -2.0 + 1.0*I;
  y = ctanhl(x);
  printf("The complex hyperbolic tangent of %Lf + %Lfi is %Lf + %Lfi\n", creall(x),
cimagl(x), creall(y), cimagl(y));
}
```

Example Output

```
The complex hyperbolic tangent of -2.000000 + 1.000000i is -1.014794 + 0.033813i
```

5.3 <ctype.h> Character Handling

The header file `ctype.h` consists of functions that are useful for classifying and mapping characters. Characters are interpreted according to the Standard `"C"` locale.

5.3.1 isalnum Function

Test for an alphanumeric character.

Include

`<ctype.h>`

Prototype

`int isalnum(int c);`

Argument

c	The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is alphanumeric; otherwise, returns a zero.

Remarks

Alphanumeric characters are included within the ranges `'A'-'Z'`, `'a'-'z'` or `'0'-'9'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = '3';
  if (isalnum(ch))
    printf("3 is an alphanumeric\n");
  else
    printf("3 is NOT an alphanumeric\n");

  ch = '#';
  if (isalnum(ch))
    printf("# is an alphanumeric\n");
  else
    printf("# is NOT an alphanumeric\n");
}
```

Example Output

```
3 is an alphanumeric
# is NOT an alphanumeric
```

5.3.2 isalpha Function

Test for an alphabetic character.

Include

`<ctype.h>`

Prototype

```
int isalpha(int c);
```

Argument

c	The character to test.

Return Value

Returns a non-zero integer value if the character is alphabetic; otherwise, returns zero.

Remarks

Alphabetic characters are included within the ranges `'A'-'Z'` or `'a'-'z'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = 'B';
  if (isalpha(ch))
    printf("B is alphabetic\n");
  else
    printf("B is NOT alphabetic\n");

  ch = '#';
  if (isalpha(ch))
    printf("# is alphabetic\n");
  else
    printf("# is NOT alphabetic\n");
}
```

Example Output

```
B is alphabetic
# is NOT alphabetic
```

5.3.3 isblank Function

Test for a space or tab character.

Include

`<ctype.h>`

Prototype

```
int isblank (int c);
```

Argument

c The character to test.

Return Value

Returns a non-zero integer value if the character is a space or tab character; otherwise, returns zero.

Remarks

A character is considered to be a white-space character if it is one of the following: space (`' '`) or horizontal tab (`'\t'`).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = '&';
  if (isblank(ch))
    printf("& is a white-space character\n");
  else
    printf("& is NOT a white-space character\n");

  ch = '\t';
  if (isblank(ch))
    printf("A tab is a white-space character\n");
  else
    printf("A tab is NOT a white-space character\n");
}
```

Example Output

```
& is NOT a white-space character
A tab is a white-space character
```

5.3.4 iscntrl Function

Test for a control character.

Include

`<ctype.h>`

Prototype

```
int iscntrl(int c);
```

Argument

c The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is a control character; otherwise, returns zero.

Remarks

A character is considered to be a control character if its ASCII value is in the range 0x00 to 0x1F inclusive, or 0x7F.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

```
int main(void)
{
  char ch;

  ch = 'B';
  if (iscntrl(ch))
    printf("B is a control character\n");
  else
    printf("B is NOT a control character\n");

  ch = '\t';
  if (iscntrl(ch))
    printf("A tab is a control character\n");
  else
    printf("A tab is NOT a control character\n");
}
```

Example Output

```
B is NOT a control character
A tab is a control character
```

5.3.5 isdigit Function

Test for a decimal digit.

Include

`<ctype.h>`

Prototype

`int isdigit(int c);`

Argument

c	The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is a digit; otherwise, returns zero.

Remarks

A character is considered to be a digit character if it is in the range of `'0'-'9'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = '3';
  if (isdigit(ch))
    printf("3 is a digit\n");
  else
    printf("3 is NOT a digit\n");

  ch = '#';
  if (isdigit(ch))
    printf("# is a digit\n");
  else
    printf("# is NOT a digit\n");
}
```

Example Output

```
3 is a digit
# is NOT a digit
```

5.3.6 isgraph Function

Test for a graphical character.

Include

`<ctype.h>`

Prototype

```
int isgraph (int c);
```

Argument

c The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is a graphical character; otherwise, returns zero.

Remarks

A character is considered to be a graphical character if it is any printable character except a space.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = '3';
  if (isgraph(ch))
    printf("3 is a graphical character\n");
  else
    printf("3 is NOT a graphical character\n");

  ch = '#';
  if (isgraph(ch))
    printf("# is a graphical character\n");
  else
    printf("# is NOT a graphical character\n");

  ch = ' ';
  if (isgraph(ch))
    printf("A space is a graphical character\n");
  else
    printf("A space is NOT a graphical character\n");
}
```

Example Output

```
3 is a graphical character
# is a graphical character
A space is NOT a graphical character
```

5.3.7 islower Function

Test for a lowercase alphabetic character.

Include

`<ctype.h>`

Prototype

```
int islower (int c);
```

Argument

c	The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is a lowercase alphabetic character; otherwise, returns zero.

Remarks

A character is considered to be a lowercase alphabetic character if it is in the range of `'a'-'z'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = 'B';
  if (islower(ch))
    printf("B is lowercase\n");
  else
    printf("B is NOT lowercase\n");

  ch = 'b';
  if (islower(ch))
    printf("b is lowercase\n");
  else
    printf("b is NOT lowercase\n");
}
```

Example Output

```
B is NOT lowercase
b is lowercase
```

5.3.8 isprint Function

Test for a printable character (includes a space).

Include

`<ctype.h>`

Prototype

```
int isprint(int c);
```

Argument

c	The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is printable; otherwise, returns zero.

Remarks

A character is considered to be a printable character if its ASCII value is in the range 0x20 to 0x7e inclusive.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = '&';
  if (isprint(ch))
    printf("& is a printable character\n");
  else
    printf("& is NOT a printable character\n");

  ch = '\t';
  if (isprint(ch))
    printf("A tab is a printable character\n");
  else
    printf("A tab is NOT a printable character\n");
}
```

Example Output

```
& is a printable character
A tab is NOT a printable character
```

5.3.9 ispunct Function

Test for a punctuation character.

Include

`<ctype.h>`

Prototype

```
int ispunct (int c);
```

Argument

c The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is a punctuation character; otherwise, returns zero.

Remarks

A character is considered to be a punctuation character if it is a printable character which is neither a space nor an alphanumeric character. Punctuation characters consist of the following:

```
'!'  '"'  '#'  '$'  '%'  '&'  '''  '('  ')'  '*'  '+'  ','  '-'  '.'  '/'  ':'
';'  '<'  '='  '>'  '?'  '@'  '['  '\'  ']'  '^'  '_'  '`'  '{'  '|'  '}'  '~'
```

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = '&';
  if (ispunct(ch))
    printf("& is a punctuation character\n");
  else
```

```
    printf("& is NOT a punctuation character\n");

  ch = '\t';
  if (ispunct(ch))
    printf("A tab is a punctuation character\n");
  else
    printf("A tab is NOT a punctuation character\n");
}
```

Example Output

```
& is a punctuation character
A tab is NOT a punctuation character
```

5.3.10 isspace Function

Test for a white-space character.

Include

`<ctype.h>`

Prototype

`int isspace (int c);`

Argument

c The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is a white-space character; otherwise, returns zero.

Remarks

A character is considered to be a white-space character if it is one of the following: space, `' '`, form feed, `'\f'`, newline, `'\n'`, carriage return, `'\r'`, horizontal tab, `'\t'`, or vertical tab, `'\v'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = '&';
  if (isspace(ch))
    printf("& is a white-space character\n");
  else
    printf("& is NOT a white-space character\n");

  ch = '\t';
  if (isspace(ch))
    printf("A tab is a white-space character\n");
  else
    printf("A tab is NOT a white-space character\n");
}
```

Example Output

```
& is NOT a white-space character
A tab is a white-space character
```

5.3.11 isupper Function

Test for an uppercase letter.

Include

`<ctype.h>`

Prototype

```
int isupper (int c);
```

Argument

c	The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is an uppercase alphabetic character; otherwise, returns zero.

Remarks

A character is considered to be an uppercase alphabetic character if it is in the range of `'A'-'Z'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = 'B';
  if (isupper(ch))
    printf("B is uppercase\n");
  else
    printf("B is NOT uppercase\n");

  ch = 'b';
  if (isupper(ch))
    printf("b is uppercase\n");
  else
    printf("b is NOT uppercase\n");
}
```

Example Output

```
B is uppercase
b is NOT uppercase
```

5.3.12 isxdigit Function

Test for a hexadecimal digit.

Include

`<ctype.h>`

Prototype

```
int isxdigit (int c);
```

Argument

c	The character to test.

Return Value

Returns a non-zero integer value if the character, `c`, is a hexadecimal digit; otherwise, returns zero.

Remarks

A character is considered to be a hexadecimal digit character if it is in the range of `'0'-'9'`, `'A'-'F'`, or `'a'-'f'`.

Note: The list does not include the `x` character, which is used when writing a hexadecimal *value*, such as `0x7FFE`. This function looks purely to see if the character is a hexadecimal *digit*.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = 'B';
  if (isxdigit(ch))
    printf("B is a hexadecimal digit\n");
  else
    printf("B is NOT a hexadecimal digit\n");

  ch = 't';
  if (isxdigit(ch))
    printf("t is a hexadecimal digit\n");
  else
    printf("t is NOT a hexadecimal digit\n");
}
```

Example Output

```
B is a hexadecimal digit
t is NOT a hexadecimal digit
```

5.3.13 tolower Function

Convert a character to a lowercase alphabetical character.

Include

`<ctype.h>`

Prototype

```
int tolower (int c);
```

Argument

c The character to convert to lowercase.

Return Value

Returns the corresponding lowercase alphabetical character if the argument, `c`, was originally uppercase; otherwise, returns the original character.

Remarks

Only uppercase alphabetical characters may be converted to lowercase.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
```

```
  int ch;

  ch = 'B';
  printf("B changes to lowercase %c\n",
          tolower(ch));

  ch = 'b';
  printf("b remains lowercase %c\n",
          tolower(ch));

  ch = '@';
  printf("@ has no lowercase, ");
  printf("so %c is returned\n", tolower(ch));
}
```

Example Output

```
B changes to lowercase b
b remains lowercase b
@ has no lowercase, so @ is returned
```

5.3.14 toupper Function

Convert a character to an uppercase alphabetical character.

Include

`<ctype.h>`

Prototype

```
int toupper (int c);
```

Argument

c The character to convert to uppercase.

Return Value

Returns the corresponding uppercase alphabetical character if the argument, `c`, was originally lowercase; otherwise, returns the original character.

Remarks

Only lowercase alphabetical characters may be converted to uppercase.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int ch;

  ch = 'b';
  printf("b changes to uppercase %c\n",
          toupper(ch));

  ch = 'B';
  printf("B remains uppercase %c\n",
          toupper(ch));

  ch = '@';
  printf("@ has no uppercase, ");
  printf("so %c is returned\n", toupper(ch));
}
```

Example Output

```
b changes to uppercase B
B remains uppercase B
@ has no uppercase, so @ is returned
```

5.4 <errno.h> Errors

The header file `errno.h` consists of macros that provide error codes that are reported by certain library functions. The variable `errno` might be assigned any value greater than zero by these library functions. Preprocessor macros are defined to represent these values. Library functions will never set `errno` to zero. At program start-up, `errno` is zero.

To test if a library function encounters an error, the program should store the zero value in `errno` immediately before calling the library function. The value should be checked after the call, before subsequent code can change the value.

5.4.1 errno Macro/Variable

Contains the value of an error when an error occurs in several library functions.

Include

`<errno.h>`

Remarks

This is implemented as a macro that aliases the `int` object `__errno_val` with all MPLAB XC implementations except for XC8 PIC, where it is defined as an `int` object in its own right.

The value of `errno` will be zero at program startup. It is set to a non-zero integer value by certain library functions when an error occurs. The `errno` object should be reset to zero prior to calling a function that may set it.

5.4.2 EDOM Macro

Represents a domain error.

Include

`<errno.h>`

Remarks

`EDOM` represents a domain error, which occurs when an input argument is outside the domain in which the function is defined.

5.4.3 EILSEQ Macro

Represents a wide character encoding error.

Include

`<errno.h>`

Remarks

`EILSEQ` represents a wide character encoding error, when the character sequence presented to the underlying `mbrtowc` function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying `wcrtomb` does not correspond to a valid (generalized) overflow or underflow error, which occurs when a result is too large or too small to be stored.

5.4.4 ERANGE Macro

Represents an overflow or underflow error.

Include

`<errno.h>`

Remarks

ERANGE represents an overflow or underflow error, which occurs when a result is too large or too small to be stored.

5.4.5 Implementation-defined Error Conditions

The following are error condition macros in addition to those specified by the C language specification that might be assigned to errno by the Microchip Unified Standard Library. They expand to positive integer constant expressions with type int, suitable for use in #if preprocessing directives

Table 5-1. Implementation-defined error condition macros

Macro	Error condition
E2BIG	Argument list too long
EACCES	Permission denied
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Address not available
EADV	Advertise error
EAFNOSUPPORT	Address family not supported by protocol family
EAGAIN	No more processes
EALREADY	Socket already connected
EBADE	Invalid exchange
EBADF	Bad file number
EBADFD	File descriptor invalid for this operation
EBADMSG	Bad message
EBADR	Invalid request descriptor
EBADRQC	Invalid request code
EBADSLT	Invalid slot
EBFONT	Bad font file fmt
EBUSY	Device or resource busy
ECANCELED	Operation canceled
ECHILD	No children
ECHRNG	Channel number out of range
ECOMM	Communication error on send
ECONNABORTED	Software caused connection abort
ECONNREFUSED	Connection refused
ECONNRESET	Connection reset by peer
EDEADLK	Deadlock
EDEADLOCK	File locking deadlock error
EDESTADDRREQ	Destination address required
EDOTDOT	Cross mount point (not really error)
EDQUOT	Disc quota exceeded
EEXIST	File exists

.....continued

Macro	Error condition
EFAULT	Bad address
EFBIG	File too large
EHOSTDOWN	Host is down
EHOSTUNREACH	Host is unreachable
EHWPOISON	Memory page has hardware error
EIDRM	Identifier removed
EINPROGRESS	Connection already in progress
EINTR	Interrupted system call
EINVAL	Invalid argument
EIO	I/O error
EISCONN	Socket is already connected
EISDIR	Is a directory
EISNAM	Is a named type file
EKEYEXPIRED	Key has expired
EKEYREJECTED	Key was rejected by service
EKEYREVOKED	Key has been revoked
EL2HLT	Level 2 halted
EL2NSYNC	Level 2 not synchronized
EL3HLT	Level 3 halted
EL3RST	Level 3 reset
ELIBACC	Can't access a needed shared lib
ELIBBAD	Accessing a corrupted shared lib
ELIBEXEC	Attempting to exec a shared library
ELIBMAX	Attempting to link in too many libs
ELIBSCN	`.lib` section in `a.out` corrupted
ELNRNG	Link number out of range
ELOOP	Too many symbolic links
EL3RST	Level 3 reset
EMEDIUMTYPE	Wrong medium type
EMFILE	File descriptor value too large
EMLINK	Too many links
EMSGSIZE	Message too long
EMULTIHOP	Multihop attempted
ENAMETOOLONG	File or path name too long
ENAVAIL	No XENIX semaphores available

.....continued

Macro	Error condition
ENETDOWN	Network interface is not configured
ENETRESET	Connection aborted by network
ENETUNREACH	Network is unreachable
ENFILE	Too many open files in system
ENOANO	No anode
ENOBUFS	No buffer space available
ENOCSI	No CSI structure available
ENODATA	No data (for no delay io)
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOKEY	Required key not available
ENOLCK	No lock
ENOLINK	Virtual circuit is gone
ENOMEDIUM	No medium found
ENOMEM	Not enough space
ENOMSG	No message of desired type
ENONET	Machine is not on the network
ENOPKG	Package not installed
ENOPROTOOPT	Protocol not available
ENOSPC	No space left on device
ENOSR	No stream resources
ENOSTR	Not a stream
ENOSYS	Function not implemented
ENOTBLK	Block device required
ENOTCONN	Socket is not connected
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTNAM	Not a XENIX named type file
ENOTRECOVERABLE	State not recoverable
ENOTSOCK	Socket operation on non-socket
ENOTSUP	Not supported
ENOTTY	Not a character device
ENOTUNIQ	Given log. name not unique
ENXIO	No such device or address

.....continued

Macro	Error condition
EOPNOTSUPP	Operation not supported on socket
EOVERFLOW	Value too large for defined data type
EOWNERDEAD	Previous owner died
EPERM	Not owner
EPFNOSUPPORT	Protocol family not supported
EPIPE	Broken pipe
EPROTO	Protocol error
EPROTONOSUPPORT	Unknown protocol
EPROTOTYPE	Protocol wrong type for socket
EREMCHG	Remote address changed
EREMOTE	The object is remote
EREMOTEIO	Remote I/O error
ERESTART	Interrupted system call should be restarted
ERFKILL	Operation not possible due to RF-kill
EROFS	Read-only file system
ESHUTDOWN	Can't send after socket shutdown
ESOCKTNOSUPPORT	Socket type not supported
ESPIPE	Illegal seek
ESRCH	No such process
ESRMNT	Srmount error
ESTALE	Stale file handle
ESTRPIPE	Streams pipe error
ETIME	Stream ioctl timeout
ETIMEDOUT	Connection timed out
ETOOMANYREFS	Too many references: cannot splice
ETXTBSY	Text file busy
EUCLEAN	Structure needs cleaning
EUNATCH	Protocol driver not attached
EUSERS	Too many users
EWOULDBLOCK	Operation would block
EXDEV	Cross-device link
EXFULL	Exchange full

5.5 <fenv.h> Floating-point Environment

The header file `fenv.h` implements macros, types, and functions that provide access to any floating-point exception status flags and direction-rounding control modes supported by the implementation. A floating-point status flag is a system variable whose value is set (but never cleared) when a floating-point exception is raised.

Attention: This header is implemented only by MPLAB XC32 C compilers.

5.5.1 FENV_ACCESS Pragma

Attention: This pragma is implemented only by MPLAB XC32 compilers.

This pragma informs the implementation that the program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes. When set to `ON`, compiler optimizations that affect floating-point status flags tests might be disabled.

Include

`<fenv.h>`

Usage

```
#pragma STDC FENV_ACCESS ON|OFF|DEFAULT.
```

Remarks

When placed outside external declarations or preceding all explicit declarations and statements inside a compound statement, the pragma takes effect from its occurrence until another `FENV_ACCESS` pragma is encountered, or until the end of the translation unit. When placed inside a compound statement, the pragma takes effect from its occurrence until another `FENV_ACCESS` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to that just before the compound statement. In any other context. the behavior of this pragma is undefined.

5.5.2 fenv Types

5.5.2.1 fenv_t Type

This type represents the entire floating-point environment.

Attention: This type is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

5.5.2.2 fexcept_t Type

This type represents the floating-point status flags collectively, including any status the implementation associates with the flags.

Attention: This type is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

5.5.3 fenv Exception Macros

Attention: These macros are implemented only by MPLAB XC32 C compilers.

The following macros relate to floating-point exceptions.

Table 5-2. Floating-point exception macros

Macro	Description
FE_DIVBYZERO	The operation on finite numbers has produced a result that is infinity.
FE_INEXACT	The rounded result of an operation differs to the infinite precision result.
FE_INVALID	The result of the operation cannot be expressed as a value.
FE_OVERFLOW	The absolute value of the floating-point result is too large to be represented.
FE_UNDERFLOW	The absolute value of the floating-point result is smaller than the smallest positive normalized floating-point number.
FE_ALL_EXCEPT	The bitwise OR of the above floating-point exception macros.

5.5.4 fenv Rounding Direction Macros

Attention: These macros are implemented only by MPLAB XC32 C compilers.

The following macros relate to the rounding direction of floating-point values that cannot be exactly represented in the significand.

Table 5-3. Floating-point rounding direction macros

Macro	Description
FE_DOWNWARD	Round down, towards negative infinity.
FE_TONEAREST	Round to the nearest value.
FE_TOWARDZERO	Round towards zero.
FE_UPWARD	Round up, towards positive infinity.

5.5.5 FE_DFL_ENV Macro

Attention: This macro is implemented only by MPLAB XC32 C compilers.

This macro represents the default floating-point environment installed at program startup and can be used as an argument to those `<fenv.h>` functions that manage the floating-point environment.

Include

`<fenv.h>`

Definition

A pointer with type `(const fenv_t *)`.

5.5.6 feclearexcept Function

Clears the supported floating-point exceptions represented by its argument.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

`void feclearexcept(int excepts);`

Argument

`excepts` a value being the bitwise OR of one or more floating-point exception macros, representing the exceptions to clear

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

volatile double result;

int main(void)
{
  feclearexcept(FE_ALL_EXCEPT);       // clear all exceptions
  result = sqrt(-1);                  // possibly generate an exeception
  if(fetestexcept(FE_INVALID))        // was an exception raised?
    printf("FE_INVALID exception raised by sqrt() function\n");
}
```

Example Output

```
FE_INVALID exception raised by sqrt() function
```

5.5.7 fegetenv Function

Stores the floating-point environment into an object.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

```
void fegetenv(fenv_t * envp);
```

Argument

envp a pointer to the object in which the environment should be stored

Remarks

This function gets the current floating-point environment and stores it into the object pointed to by the pointer argument.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  fenv_t envp;
  volatile double x, y=1E30;

  fegetenv(&envp);              // save the current environment
  fesetround(FE_UPWARD);        // change the environment
  x = 1 / y;
  printf("The small result rounded up is %g\n", x);
  fesetround(FE_DOWNWARD);
  x = 1 / y;
  printf("The small result rounded down is %g\n", x);
  fesetenv(&envp);              // restore the environemnt
}
```

Example Output

```
The small result rounded up is 1.00001e-30
The small result rounded down is 9.99999e-31
```

5.5.8 fegetexceptflag Function

Stores the floating-point exception flags represented by the argument excepts into an object.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

```
void fegetexceptflag(fexcept_t * flagp, int excepts);
```

Arguments

flagp a pointer to the object in which the flag representation will be stored

excepts a value being the bitwise OR of one or more floating-point exception macros, representing the exceptions to store

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void showExceptions(void)
{
  printf("Exceptions currently raised: ");
  if(fetestexcept(FE_DIVBYZERO))
    printf("FE_DIVBYZERO ");
  if(fetestexcept(FE_INEXACT))
    printf("FE_INEXACT ");
  if(fetestexcept(FE_INVALID))
    printf("FE_INVALID ");
  if(fetestexcept(FE_OVERFLOW))
    printf("FE_OVERFLOW ");
  if(fetestexcept(FE_UNDERFLOW))
    printf("FE_UNDERFLOW ");
  printf("\n");
}

volatile double result = 0.0;

int main(void)
{
  fexcept_t excepts;

  result = 1.0 / result;                        // raise FE_DIVBYZERO
  showExceptions();
  fegetexceptflag(&excepts, FE_ALL_EXCEPT);     // save state
  feraiseexcept(FE_INVALID|FE_OVERFLOW);        // raise exceptions without operation
  showExceptions();
  fesetexceptflag(&excepts, FE_ALL_EXCEPT);     // restore state
  showExceptions();
}
```

Example Output

```
Exceptions currently raised: FE_DIVBYZERO
Exceptions currently raised: FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW
Exceptions currently raised: FE_DIVBYZERO
```

5.5.9 fegetround Function

Gets the current rounding direction.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

```
int fegetround(void);
```

Return Value

This function returns the a value representing the current floating-point rounding direction, which can be compared to the rounding-direction macros.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <fenv.h>
#include <stdio.h>
#include <assert.h>

int main(void)
{
  int save_round, setround_ok;
  volatile double x, y=1E30;

  save_round = fegetround();            // save the rounding state
  setround_ok = fesetround(FE_UPWARD);
  assert(setround_ok == 0);
  x = 1 / y;
  printf("the small result rounded up is %.40f\n", x);
  setround_ok = fesetround(FE_DOWNWARD);
  assert(setround_ok == 0);
  x = 1 / y;
  printf("the small result rounded down is %.40f\n", x);
  fesetround(save_round);               // restore the rounding state
}
```

Example Output

```
the small result rounded up is 0.0000000000000000000000000000010000000001
the small result rounded down is 0.0000000000000000000000000000009999999999
```

5.5.10 feholdexcept Function

Stores the floating-point environment into an object then installs a non-stop exception mode.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

```
int feholdexcept(fenv_t * envp);
```

Argument

`envp` a pointer to the object in which the environment should be stored

Return Value

This function returns zero if and only if the non-stop exception mode was successfully installed.

Remarks

This function saves the current floating-point environment in the object pointed to by `envp`, clears the floating-point status flags, and then installs a non-stop exception mode, if available. Once installed, the non-stop mode will allow execution to continue on a floating-point exception and can be used to hide spurious floating-point exceptions.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <fenv.h>
#include <stdio.h>
#include <assert.h>

int main(void)
{
  int nonStopFail;
  fenv_t envp;
  volatile double x, y=1E30;

  nonStopFail = feholdexcept(&envp);      // save the current environment
  assert(nonStopFail == 0);
  fesetround(FE_UPWARD);
  x = 1 / y;
  printf("the small result rounded up is %g\n", x);
  fesetround(FE_DOWNWARD);
  x = 1 / y;
  printf("the small result rounded down is %g\n", x);
  feupdateenv(&envp);                     // restore the environment
}
```

Example Output

```
The small result rounded up is 1.00001e-30
The small result rounded down is 9.99999e-31
```

5.5.11 feraiseexcept Function

Raises the supported floating-point exceptions represented by its argument.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

`void feraiseexcept(int excepts);`

Argument

`excepts` a value being the bitwise OR of one or more floating-point exception macros, representing the exceptions to raise

Remarks

The order in which these floating-point exceptions are raised is unspecified, except if the argument represents "overflow" and "inexact", or "underflow" and "inexact", in which case "overflow" or "underflow" is raised before "inexact".

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void showExceptions(void)
{
  printf("Exceptions currently raised: ");
```

```
  if(fetestexcept(FE_DIVBYZERO))
    printf("FE_DIVBYZERO ");
  if(fetestexcept(FE_INEXACT))
    printf("FE_INEXACT ");
  if(fetestexcept(FE_INVALID))
    printf("FE_INVALID ");
  if(fetestexcept(FE_OVERFLOW))
    printf("FE_OVERFLOW ");
  if(fetestexcept(FE_UNDERFLOW))
    printf("FE_UNDERFLOW ");
  printf("\n");
}

volatile double result = 0.0;

int main(void)
{
  fexcept_t excepts;

  result = 1.0 / result;                          // raise FE_DIVBYZERO
  showExceptions();
  fegetexceptflag(&excepts, FE_ALL_EXCEPT);       // save state
  feraiseexcept(FE_INVALID|FE_OVERFLOW);          // raise exceptions without operation
  showExceptions();
  fesetexceptflag(&excepts, FE_ALL_EXCEPT);       // restore state
  showExceptions();
}
```

Example Output

```
Exceptions currently raised: FE_DIVBYZERO
Exceptions currently raised: FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW
Exceptions currently raised: FE_DIVBYZERO
```

5.5.12 fesetenv Function

Establishes the floating-point environment from an object.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

`void fesetenv(fenv_t * envp);`

Argument

`envp` a pointer to the object in which the environment should be obtained

Remarks

This function establishes the floating-point environment from the object pointed to by `envp`. This pointer should be pointing to an object obtained from a call to `fegetenv()` or `feholdexcept()`, or assigned a floating-point environment macro, such as `FE_DFL_ENV`. Although the state of the floating-point status flags are set, these floating-point exceptions are not raised.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

```
int main(void)
{
  fenv_t envp;
  volatile double x, y=1E30;

  fegetenv(&envp);              // save the current environment
  fesetround(FE_UPWARD);        // change the environment
  x = 1 / y;
  printf("The small result rounded up is %g\n", x);
  fesetround(FE_DOWNWARD);
  x = 1 / y;
  printf("The small result rounded down is %g\n", x);
  fesetenv(&envp);              // restore the environemnt
}
```

Example Output

```
The small result rounded up is 1.00001e-30
The small result rounded down is 9.99999e-31
```

5.5.13 fesetexceptflag Function

Sets the floating-point status flags to those represented by the state stored in the object.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

```
void fesetexceptflag(fexcept_t * flagp, int excepts);
```

Arguments

flagp	a pointer to the object from which the flag representation will be read
excepts	a value being the bitwise OR of one or more floating-point exception macros, representing the possible exceptions to set

Remarks

The pointer must hold an address obtained by a previous call to `fegetexceptflag()` whose second argument represented at least those floating-point exceptions being set by this call. Only those state flags in the second argument may be set by this function. This function does not raise floating-point exceptions; it only sets the state of the flags.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void showExceptions(void)
{
  printf("Exceptions currently raised: ");
  if(fetestexcept(FE_DIVBYZERO))
    printf("FE_DIVBYZERO ");
  if(fetestexcept(FE_INEXACT))
    printf("FE_INEXACT ");
  if(fetestexcept(FE_INVALID))
    printf("FE_INVALID ");
  if(fetestexcept(FE_OVERFLOW))
```

```
    printf("FE_OVERFLOW ");
  if(fetestexcept(FE_UNDERFLOW))
    printf("FE_UNDERFLOW ");
  printf("\n");
}

volatile double result = 0.0;

int main(void)
{
  fexcept_t excepts;

  result = 1.0 / result;                        // raise FE_DIVBYZERO
  showExceptions();
  fegetexceptflag(&excepts, FE_ALL_EXCEPT);     // save state
  feraiseexcept(FE_INVALID|FE_OVERFLOW);        // raise exceptions without operation
  showExceptions();
  fesetexceptflag(&excepts, FE_ALL_EXCEPT);     // restore state
  showExceptions();
}
```

Example Output

```
Exceptions currently raised: FE_DIVBYZERO
Exceptions currently raised: FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW
Exceptions currently raised: FE_DIVBYZERO
```

5.5.14 fesetround Function

Sets the current rounding direction.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

`int fesetround(int round);`

Argument

round	the requested rounding direction

Return Value

This function returns zero if and only if the requested rounding direction was established.

Remarks

This function attempts to set the current floating-point rounding direction to that specified by the argument, which should be the value of the required rounding direction macro.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <fenv.h>
#include <stdio.h>
#include <assert.h>

int main(void)
{
  int save_round, setround_ok;
  volatile double x, y=1E30;
```

```
  save_round = fegetround();            // save the rounding state
  setround_ok = fesetround(FE_UPWARD);
  assert(setround_ok == 0);
  x = 1 / y;
  printf("the small result rounded up is %.40f\n", x);
  setround_ok = fesetround(FE_DOWNWARD);
  assert(setround_ok == 0);
  x = 1 / y;
  printf("the small result rounded down is %.40f\n", x);
  fesetround(save_round);               // restore the rounding state
}
```

Example Output

```
the small result rounded up is 0.0000000000000000000000000000010000000001
the small result rounded down is 0.0000000000000000000000000000009999999999
```

5.5.15 fetestexcept Function

Determines the state of a specified subset of the currently set floating-point exceptions.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

```
int fetestexcept(int excepts);
```

Argument

`excepts` a value being the bitwise OR of one or more floating-point exception macros, representing the exceptions to test

Retrun Value

The function returns the bitwise OR of the floating-point exception macro values associated with those exceptions that are specified by the argument and that are currently set.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void showExceptions(void)
{
  printf("Exceptions currently raised: ");
  if(fetestexcept(FE_DIVBYZERO))
    printf("FE_DIVBYZERO ");
  if(fetestexcept(FE_INEXACT))
    printf("FE_INEXACT ");
  if(fetestexcept(FE_INVALID))
    printf("FE_INVALID ");
  if(fetestexcept(FE_OVERFLOW))
    printf("FE_OVERFLOW ");
  if(fetestexcept(FE_UNDERFLOW))
    printf("FE_UNDERFLOW ");
  printf("\n");
}

volatile double result = 0.0;
```

```
int main(void)
{
  fexcept_t excepts;

  result = 1.0 / result;                        // raise FE_DIVBYZERO
  showExceptions();
  fegetexceptflag(&excepts, FE_ALL_EXCEPT);     // save state
  feraiseexcept(FE_INVALID|FE_OVERFLOW);        // raise exceptions without operation
  showExceptions();
  fesetexceptflag(&excepts, FE_ALL_EXCEPT);     // restore state
  showExceptions();
}
```

Example Output

```
Exceptions currently raised: FE_DIVBYZERO
Exceptions currently raised: FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW
Exceptions currently raised: FE_DIVBYZERO
```

5.5.16 feupdateenv Function

Installs the environment from an object, preserving the state of the current floating-point exceptions.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<fenv.h>`

Prototype

`void feupdateenv(fenv_t * envp);`

Argument

envp a pointer to the object in which the environment should be stored

Remarks

This function saves the current floating-point environment in automatic storage, installs the floating-point environment represented by the object pointed to by `envp`, and then raises the saved floating-point exceptions. The pointer argument should be pointing to an object obtained from a call to `fegetenv()` or `feholdexcept()`, or assigned a floating-point environment macro, such as `FE_DFL_ENV`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <fenv.h>
#include <stdio.h>
#include <assert.h>

int main(void)
{
  int nonStopFail;
  fenv_t envp;
  volatile double x, y=1E30;

  nonStopFail = feholdexcept(&envp);        // save the current environment
  assert(nonStopFail == 0);
  fesetround(FE_UPWARD);
  x = 1 / y;
  printf("the small result rounded up is %g\n", x);
  fesetround(FE_DOWNWARD);
  x = 1 / y;
```

```
  printf("the small result rounded down is %g\n", x);
  feupdateenv(&envp);                     // restore the environment
}
```

Example Output

```
The small result rounded up is 1.00001e-30
The small result rounded down is 9.99999e-31
```

5.6 <float.h> Floating-Point Characteristics

The header file `float.h` consists of macros that specify various properties of floating-point types. These properties include number of significant figures, size limits and what rounding mode is used.

5.6.1 DBL_DIG Macro

Number of decimal digits of precision in a double precision floating-point value.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`6`
MPLAB XC16	`6/15` (See attention note)
MPLAB XC32	`15`

5.6.2 DBL_EPSILON Macro

The difference between 1.0 and the next larger representable double precision floating-point value.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`1.1920928955078125e-7F`
MPLAB XC16	`1.1920928955078125e-7F/2.2204460492503131e-16L` (See attention note)
MPLAB XC32	`2.2204460492503131e-16L`

5.6.3 DBL_MANT_DIG Macro

Number of base-`FLT_RADIX` digits in a double precision floating-point significand.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`24`
MPLAB XC16	`24/53` (See attention note)
MPLAB XC32	`53`

5.6.4 DBL_MAX Macro

Maximum finite double precision floating-point value.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`3.40282346638528859812e+38F`
MPLAB XC16	`3.4028234663852886e+38F/1.7976931348623157081e+308` (See attention note)
MPLAB XC32	`1.7976931348623157081e+308`

5.6.5 DBL_MAX_10_EXP Macro

Maximum integer value for a double precision floating-point exponent in base 10.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`38`
MPLAB XC16	`38/308` (See attention note)
MPLAB XC32	`308`

5.6.6 DBL_MAX_EXP Macro

Maximum integer value for a double precision floating-point exponent in base `FLT_RADIX`.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`128`
MPLAB XC16	`128/1024` (See attention note)
MPLAB XC32	`1024`

5.6.7 DBL_MIN Macro

Minimum normalized positive double precision floating-point value.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`1.17549435082228750797e-38F`
MPLAB XC16	`((double)1.1754943508222875e-38L)/` `((double)2.2250738585072014e-308L)` (See attention note)
MPLAB XC32	`((double)2.2250738585072014e-308L)`

5.6.8 DBL_MIN_10_EXP Macro

Minimum negative integer value for a double precision floating-point exponent in base 10.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`(-37)`

.....continued	
Compiler	**Definition**
MPLAB XC16	`(-37)/(-307)` (See attention note)
MPLAB XC32	`(-307)`

5.6.9 DBL_MIN_EXP Macro

Minimum negative integer value for a double precision floating-point exponent in base `FLT_RADIX`.

Attention: The default size of the MPLAB XC16 `double` type is 32 bits, but this can be changed to 64 bits with the `-fno-short-double` option.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`(-125)`
MPLAB XC16	`(-125)/(-1021)` (See attention note)
MPLAB XC32	`(-1021)`

5.6.10 DECIMAL_DIG Macro

The number of decimal digits, *n*, such that any floating-point number in the widest supported floating type with the largest *XXX*`_MANT_DIG` radix 2 digits can be rounded to a floating-point number with *n* decimal digits and back again without change to the value.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	`9`
MPLAB XC16	`17`
MPLAB XC32	`17`

5.6.11 FLT_DIG Macro

Number of decimal digits of precision in a single precision floating-point value.

Include

`<float.h>`

Definition

The value `6` for all compilers.

5.6.12 FLT_EPSILON Macro

The difference between 1.0 and the next larger representable single precision floating-point value.

Include

`<float.h>`

Value

The value `1.1920928955078125e-07F` for all compilers.

5.6.13 FLT_EVAL_METHOD Macro

Specifies the range and precision of floating operands and values subject to the usual arithmetic conversions and of floating constants. The possible values and meanings are as follows.

Include

`<float.h>`

Definition

The value `0` for all compilers, implying that all operations and constants are evaluated just to the range and precision of the type.

Remarks

The meaning of values for this macro as defined by the C standard are as follows.

-1 indeterminable

0 evaluate all operations and constants just to the range and precision of the type

1 evaluate operations and constants of type `float` and `double` to the range and precision of the `double` type, evaluate `long double` operations and constants to the range and precision of the `long double` type

2 evaluate all operations and constants to the range and precision of the `long double` type

5.6.14 FLT_MANT_DIG Macro

Number of base-`FLT_RADIX` digits in a single precision floating-point significand.

Include

`<float.h>`

Value

The value 24.

5.6.15 FLT_MAX Macro

Maximum finite single precision floating-point value.

Include

`<float.h>`

Value

The value 3.402823e+38.

5.6.16 FLT_MAX_10_EXP Macro

Maximum integer value for a single precision floating-point exponent in base 10.

Include

`<float.h>`

Value

The value 38.

5.6.17 FLT_MAX_EXP Macro

Maximum integer value for a single precision floating-point exponent in base `FLT_RADIX`.

Include

`<float.h>`

Value

The value 128.

5.6.18 FLT_MIN Macro

Minimum single precision floating-point value.

Include

`<float.h>`

Value

The value 1.175494e-38.

5.6.19 FLT_MIN_10_EXP Macro

Minimum negative integer value for a single precision floating-point exponent in base 10.

Include

`<float.h>`

Value

The value -37.

5.6.20 FLT_MIN_EXP Macro

Minimum negative integer value for a single precision floating-point exponent in base `FLT_RADIX`.

Include

`<float.h>`

Value

The value -125.

5.6.21 FLT_RADIX Macro

Radix of the exponent representation.

Include

`<float.h>`

Value

2

Remarks

The value 2 (binary).

5.6.22 FLT_ROUNDS Macro

Represents the rounding mode for floating-point operations.

Include

`<float.h>`

Definition

The value `1` for all compilers, implying that rounding is to the nearest representable value.

Remarks

The meaning of values for this macro as defined by the C standard are as follows.

-1	indeterminable
0	toward zero

1	to nearest representable value
2	toward positive infinity
3	toward negative infinity

5.6.23 LDBL_DIG Macro

Number of decimal digits of precision in a long double precision floating-point value.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	6
MPLAB XC16	15
MPLAB XC32	15

5.6.24 LDBL_EPSILON Macro

The difference between 1.0 and the next larger representable long double precision floating-point value.

Include

`<float.h>`

Value

Compiler	Definition
MPLAB XC8	1.1920928955078125e-07F
MPLAB XC16	2.2204460492503131e-16L
MPLAB XC32	2.2204460492503131e-16L

5.6.25 LDBL_MANT_DIG Macro

Number of base-`FLT_RADIX` digits in a long double precision floating-point significand.

Include

`<float.h>`

Value

Compiler	Definition
MPLAB XC8	24
MPLAB XC16	53
MPLAB XC32	53

5.6.26 LDBL_MAX Macro

Maximum finite long double precision floating-point value.

Include

`<float.h>`

Definition

Compiler	Definition
MPLAB XC8	3.4028234663852885981e+38F
MPLAB XC16	1.7976931348623157e+308L
MPLAB XC32	1.7976931348623157e+308L

5.6.27 LDBL_MAX_10_EXP Macro

Maximum integer value for a long double precision floating-point exponent in base 10.

Include

<float.h>

Definition

Compiler	Definition
MPLAB XC8	38
MPLAB XC16	308
MPLAB XC32	308

5.6.28 LDBL_MAX_EXP Macro

Maximum integer value for a long double precision floating-point exponent in base FLT_RADIX.

Include

<float.h>

Value

Compiler	Definition
MPLAB XC8	128
MPLAB XC16	1024
MPLAB XC32	1024

5.6.29 LDBL_MIN Macro

Minimum normalized positive long double precision floating-point value.

Include

<float.h>

Definition

Compiler	Definition
MPLAB XC8	1.1754943508222875079e-38F
MPLAB XC16	2.2250738585072014e-308L
MPLAB XC32	2.2250738585072014e-308L

5.6.30 LDBL_MIN_10_EXP Macro

Minimum negative integer value for a long double precision floating-point exponent in base 10.

Include

<float.h>

Definition

Compiler	Definition
MPLAB XC8	(-37)
MPLAB XC16	(-307)
MPLAB XC32	(-307)

5.6.31 LDBL_MIN_EXP Macro

Minimum negative integer value for a long double precision floating-point exponent in base FLT_RADIX.

Include

<float.h>

Definition

Compiler	Definition
MPLAB XC8	(-125)
MPLAB XC16	(-1021)
MPLAB XC32	(-1021)

5.7 <inttypes.h> Integer Format Conversion

The content of the header file inttypes.h consists of functions that work with greatest-width integer types, and format specifiers, which can be used with printf() and scanf() functions and their wide-character counterparts.

This header includes <stdint.h>

5.7.1 inttypes.h types

imaxdiv_t
A type that holds a quotient and remainder of a signed integer division with greatest width integer operands.

Definition

```
typedef struct { intmax_t quot, rem; } imaxdiv_t;
```

Remarks

This is the structure type returned by the function, imaxdiv.

5.7.2 Print Format Macros for Signed Integers

Placeholder macros that can be used with the printf family of functions when printing signed integer values.

Attention: The format macros for 64-bit quantities are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

<inttypes.h>

Remarks

The macros in following table expand to character string literals representing conversion specifier characters that can be used with the printf family of functions when printing signed integer values.

Macro Name	Description
PRId8	Decimal placeholder string for a signed 8-bit integer type (`int8_t`).
PRId16	Decimal placeholder string for a signed 16-bit integer type (`int16_t`).
PRId32	Decimal placeholder string for a signed 32-bit integer type (`int32_t`).
PRId64	Decimal placeholder string for a signed 64-bit integer type (`int64_t`), where supported (see Attention note).
PRIi8	Integer placeholder string for a signed 8-bit integer type (`int8_t`).
PRIi16	Integer placeholder string for a signed 16-bit integer type (`int16_t`).
PRIi32	Integer placeholder string for a signed 32-bit integer type (`int32_t`).
PRIi64	Integer placeholder string for a signed 64-bit integer type (`int64_t`), where supported (see Attention note).
PRIdFAST8	Decimal placeholder string for the fastest signed integer type with a width of at least 8 bits (`int_fast8_t`).
PRIdFAST16	Decimal placeholder string for the fastest signed integer type with a width of at least 16 bits (`int_fast16_t`).
PRIdFAST32	Decimal placeholder string for the fastest signed integer type with a width of at least 32 bits (`int_fast32_t`).
PRIdFAST64	Decimal placeholder string for the fastest signed integer type with a width of at least 64 bits (`int_fast64_t`), where supported (see Attention note).
PRIiFAST8	Integer placeholder string for the fastest signed integer type with a width of at least 8 bits (`int_fast8_t`).
PRIiFAST16	Integer placeholder string for the fastest signed integer type with a width of at least 16 bits (`int_fast16_t`).
PRIiFAST32	Integer placeholder string for the fastest signed integer type with a width of at least 32 bits (`int_fast32_t`).
PRIiFAST64	Integer placeholder string for the fastest signed integer type with a width of at least 64 bits (`int_fast64_t`), where supported (see Attention note).
PRIdLEAST8	Decimal placeholder string for a signed integer type with a width of at least 8 bits (`int_least8_t`).
PRIdLEAST16	Decimal placeholder string for a signed integer type with a width of at least 16 bits (`int_least16_t`).
PRIdLEAST32	Decimal placeholder string for a signed integer type with a width of at least 32 bits (`int_least32_t`).
PRIdLEAST64	Decimal placeholder string for a signed integer type with a width of at least 64 bits (`int_least64_t`), where supported (see Attention note).
PRIiLEAST8	Integer placeholder string for a signed integer type with a width of at least 8 bits (`int_least8_t`).
PRIiLEAST16	Integer placeholder string for a signed integer type with a width of at least 16 bits (`int_least16_t`).
PRIiLEAST32	Integer placeholder string for a signed integer type with a width of at least 32 bits (`int_least32_t`).

.....continued	
Macro Name	**Description**
PRIiLEAST64	Integer placeholder string for a signed integer type with a width of at least 64 bits (`int_least64_t`), where supported (see Attention note).
PRIdMAX	Decimal placeholder string for a signed integer type with maximum width (`intmax_t`).
PRIiMAX	Integer placeholder string for a signed integer type with maximum width (`intmax_t`).
PRIdPTR	Decimal placeholder string for the `intptr_t` type.
PRIiPTR	Integer placeholder string for the `intptr_t` type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int8_t s8 = 23;
int_least32_t l32 = -324;

int main(void)
{
  printf("s8 value: %" PRId8 "\n", s8);
  printf("l32 value: %" PRIiLEAST32 "\n", l32);
}
```

5.7.3 Print Format Macros for Unsigned Integers

Placeholder macros that can be used with the `printf` family of functions when printing unsigned integer values.

Attention: The format macros for 64-bit quantities are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<inttypes.h>`

Remarks

The macros in following table expand to character string literals representing placeholders that can be used with the `printf` family of functions when printing unsigned integer values.

Macro Name	Description
PRIo8	Octal placeholder string for an unsigned 8-bit integer type (`uint8_t`).
PRIo16	Octal placeholder string for an unsigned 16-bit integer type (`uint16_t`).
PRIo32	Octal placeholder string for an unsigned 32-bit integer type (`uint32_t`).
PRIo64	Octal placeholder string for an unsigned 64-bit integer type (`uint64_t`), where supported (see Attention note).
PRIu8	Unsigned decimal placeholder string for an unsigned 8-bit integer type (`uint8_t`).

.....continued

Macro Name	Description
PRIu16	Unsigned decimal placeholder string for an unsigned 16-bit integer type (`uint16_t`).
PRIu32	Unsigned decinmal placeholder string for an unsigned 32-bit integer type (`uint32_t`).
PRIu64	Unsigned decimal placeholder string for an unsigned 64-bit integer type (`uint64_t`), where supported (see Attention note).
PRIx8/ PRIX8	Hexadecimal placeholder string for an unsigned 8-bit integer type (`uint8_t`).
PRIx16/ PRIX16	Hexadecimal placeholder string for an unsigned 16-bit integer type (`uint16_t`).
PRIx32/ PRIX32	Hexadecimal placeholder string for an unsigned 32-bit integer type (`uint32_t`).
PRIx64/ PRIX64	Hexadecimal placeholder string for an unsigned 64-bit integer type (`uint64_t`), where supported (see Attention note).
PRIoFAST8	Octal placeholder string for the fastest unsigned integer type with a width of at least 8 bits (`uint_fast8_t`).
PRIoFAST16	Octal placeholder string for the fastest unsigned integer type with a width of at least 16 bits (`uint_fast16_t`).
PRIoFAST32	Octal placeholder string for the fastest unsigned integer type with a width of at least 32 bits (`uint_fast32_t`).
PRIoFAST64	Octal placeholder string for the fastest unsigned integer type with a width of at least 64 bits (`uint_fast64_t`), where supported (see Attention note).
PRIuFAST8	Unsigned decimal placeholder string for the fastest unsigned integer type with a width of at least 8 bits (`uint_fast8_t`).
PRIuFAST16	Unsigned decimal placeholder string for the fastest unsigned integer type with a width of at least 16 bits (`uint_fast16_t`).
PRIuFAST32	Unsigned decimal placeholder string for the fastest unsigned integer type with a width of at least 32 bits (`uint_fast32_t`).
PRIuFAST64	Unsigned decimal placeholder string for the fastest unsigned integer type with a width of at least 64 bits (`uint_fast64_t`), where supported (see Attention note).
PRIxFAST8/PRIXFAST8	Hexadecimal placeholder string for the fastest unsigned integer type with a width of at least 8 bits (`uint_fast8_t`).
PRIxFAST16/PRIXFAST16	Hexadecimal placeholder string for the fastest unsigned integer type with a width of at least 16 bits (`uint_fast16_t`).
PRIxFAST32/PRIXFAST32	Hexadecimal placeholder string for the fastest unsigned integer type with a width of at least 32 bits (`uint_fast32_t`).
PRIxFAST64/PRIXFAST64	Octal placeholder string for the fastest unsigned integer type with a width of at least 64 bits (`uint_fast64_t`), where supported (see Attention note).
PRIoLEAST8	Octal placeholder string for an unsigned integer type with a width of at least 8 bits (`uint_least8_t`).

.....continued

Macro Name	Description
PRIoLEAST16	Octal placeholder string for an unsigned integer type with a width of at least 16 bits (`uint_least16_t`).
PRIoLEAST32	Octal placeholder string for an unsigned integer type with a width of at least 32 bits (`uint_least32_t`).
PRIoLEAST64	Octal placeholder string for an unsigned integer type with a width of at least 64 bits (`uint_least64_t`), where supported (see Attention note).
PRIuLEAST8	Unsigned decimal placeholder string for an unsigned integer type with a width of at least 8 bits (`uint_least8_t`).
PRIuLEAST16	Unsigned decimal placeholder string for an unsigned integer type with a width of at least 16 bits (`uint_least16_t`).
PRIuLEAST32	Unsigned decimal placeholder string for an unsigned integer type with a width of at least 32 bits (`uint_least32_t`).
PRIuLEAST64	Unsigned decimal placeholder string for an unsigned integer type with a width of at least 64 bits (`uint_least64_t`), where supported (see Attention note).
PRIxLEAST8/PRIXLEAST8	Hexadecimal placeholder string for an unsigned integer type with a width of at least 8 bits (`uint_least8_t`).
PRIxLEAST16/PRIXLEAST16	Hexadecimal placeholder string for an unsigned integer type with a width of at least 16 bits (`uint_least16_t`).
PRIxLEAST32/PRIXLEAST32	Hexadecimal placeholder string for an unsigned integer type with a width of at least 32 bits (`uint_least32_t`)
PRIxLEAST64/PRIXLEAST64	Hexadecimal placeholder string for an unsigned integer type with a width of at least 64 bits (`uint_least64_t`).
PRIoMAX	Octal placeholder string for an unsigned integer type with maximum width (`uintmax_t`).
PRIuMAX	Unsigned decimal placeholder string for an unsigned integer type with maximum width (`uintmax_t`).
PRIxMAX/PRIXMAX	Hexadecimal placeholder string for an unsigned integer type with maximum width (`uintmax_t`).
PRIoPTR	Octal placeholder string for the `uintptr_t`type.
PRIuPTR	Unsigned decimal placeholder string for the `uintptr_t` type.
PRIxPTR/PRIXPTR	Hexadecimal placeholder string for the `uintptr_t` type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

uint8_t u8 = 23;
uint_least32_t ul32 = 0x345678;

int main(void)
{
  printf("u8 value: %" PRIo8 "\n", u8);
```

```
  printf("ul32 value: %" PRIxLEAST32 "\n", ul32);
}
```

5.7.4 Scan Format Macros for Signed Integers

Placeholder macros that can be used with the `scanf` family of functions when reading in signed integer values.

Attention: The format macros for 64-bit quantities are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<inttypes.h>`

Remarks

The macros in following table expand to character string literals representing placeholders that can be used with the `scanf` family of functions when reading in signed integer values.

Macro Name	Description
SCNd8	Decimal placeholder string for a signed 8-bit integer type (`int8_t`).
SCNd16	Decimal placeholder string for a signed 16-bit integer type (`int16_t`).
SCNd32	Decimal placeholder string for a signed 32-bit integer type (`int32_t`).
SCNd64	Decimal placeholder string for a signed 64-bit integer type (`int64_t`), where supported (see Attention note).
SCNi8	Integer placeholder string for a signed 8-bit integer type (`int8_t`).
SCNi16	Integer placeholder string for a signed 16-bit integer type (`int16_t`).
SCNi32	Integer placeholder string for a signed 32-bit integer type (`int32_t`).
SCNi64	Integer placeholder string for a signed 64-bit integer type (`int64_t`), where supported (see Attention note).
SCNdFAST8	Decimal placeholder string for the fastest signed integer type with a width of at least 8 bits (`int_fast8_t`).
SCNdFAST16	Decimal placeholder string for the fastest signed integer type with a width of at least 16 bits (`int_fast16_t`).
SCNdFAST32	Decimal placeholder string for the fastest signed integer type with a width of at least 32 bits (`int_fast32_t`).
SCNdFAST64	Decimal placeholder string for the fastest signed integer type with a width of at least 64 bits (`int_fast64_t`), where supported (see Attention note).
SCNiFAST8	Integer placeholder string for the fastest signed integer type with a width of at least 8 bits (`int_fast8_t`).
SCNiFAST16	Integer placeholder string for the fastest signed integer type with a width of at least 16 bits (`int_fast16_t`).
SCNiFAST32	Integer placeholder string for the fastest signed integer type with a width of at least 32 bits (`int_fast32_t`).
SCNiFAST64	Integer placeholder string for the fastest signed integer type with a width of at least 64 bits (`int_fast64_t`), where supported (see Attention note).

.....continued

Macro Name	Description
SCNdLEAST8	Decimal placeholder string for a signed integer type with a width of at least 8 bits (`int_least8_t`).
SCNdLEAST16	Decimal placeholder string for a signed integer type with a width of at least 16 bits (`int_least16_t`).
SCNdLEAST32	Decimal placeholder string for a signed integer type with a width of at least 32 bits (`int_least32_t`).
SCNdLEAST64	Decimal placeholder string for a signed integer type with a width of at least 64 bits (`int_least64_t`), where supported (see Attention note).
SCNiLEAST8	Integer placeholder string for a signed integer type with a width of at least 8 bits (`int_least8_t`).
SCNiLEAST16	Integer placeholder string for a signed integer type with a width of at least 16 bits (`int_least16_t`).
SCNiLEAST32	Integer placeholder string for a signed integer type with a width of at least 32 bits (`int_least32_t`).
SCNiLEAST64	Integer placeholder string for a signed integer type with a width of at least 64 bits (`int_least64_t`), where supported (see Attention note).
SCNdMAX	Decimal placeholder string for a signed integer type with maximum width (`intmax_t`).
SCNiMAX	Integer placeholder string for a signed integer type with maximum width (`intmax_t`).
SCNdPTR	Decimal placeholder string for the `intptr_t` type.
SCNiPTR	Integer placeholder string for the `intptr_t` type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int_least16_t l16;
intmax_t smax;

int main(void)
{
  scanf("%" SCNdLEAST16, &l16);
  scanf("%" SCNiMAX, &smax);
}
```

5.7.5 Scan Format Macros for Unsigned Integers

Placeholder macros that can be used with the `scanf` family of functions when reading in unsigned integer values.

Attention: The format macros for 64-bit quantities are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

<inttypes.h>

Remarks

The macros in following table expand to character string literals representing placeholders that can be used with the printf family of functions when reading in unsigned integer values.

Macro Name	Description
SCNo8	Octal placeholder string for an unsigned 8-bit integer type (uint8_t).
SCNo16	Octal placeholder string for an unsigned 16-bit integer type (uint16_t).
SCNo32	Octal placeholder string for an unsigned 32-bit integer type (uint32_t).
SCNo64	Octal placeholder string for an unsigned 64-bit integer type (uint64_t), where supported (see Attention note).
SCNu8	Unsigned decimal placeholder string for an unsigned 8-bit integer type (uint8_t).
SCNu16	Unsigned decimal placeholder string for an unsigned 16-bit integer type (uint16_t).
SCNu32	Unsigned decimal placeholder string for an unsigned 32-bit integer type (uint32_t).
SCNu64	Unsigned decimal placeholder string for an unsigned 64-bit integer type (uint64_t), where supported (see Attention note).
SCNx8	Hexadecimal placeholder string for an unsigned 8-bit integer type (uint8_t).
SCNx16	Hexadecimal placeholder string for an unsigned 16-bit integer type (uint16_t).
SCNx32	Hexadecimal placeholder string for an unsigned 32-bit integer type (uint32_t).
SCNx64	Hexadecimal placeholder string for an unsigned 64-bit integer type (uint64_t), where supported (see Attention note).
SCNoFAST8	Octal placeholder string for the fastest unsigned integer type with a width of at least 8 bits (uint_fast8_t).
SCNoFAST16	Octal placeholder string for the fastest unsigned integer type with a width of at least 16 bits (uint_fast16_t).
SCNoFAST32	Octal placeholder string for the fastest unsigned integer type with a width of at least 32 bits (uint_fast32_t).
SCNoFAST64	Octal placeholder string for the fastest unsigned integer type with a width of at least 64 bits (uint_fast64_t), where supported (see Attention note).
SCNuFAST8	Unsigned decimal placeholder string for the fastest unsigned integer type with a width of at least 8 bits (uint_fast8_t).
SCNuFAST16	Unsigned decimal placeholder string for the fastest unsigned integer type with a width of at least 16 bits (uint_fast16_t).
SCNuFAST32	Unsigned decimal placeholder string for the fastest unsigned integer type with a width of at least 32 bits (uint_fast32_t).
SCNuFAST64	Unsigned decimal placeholder string for the fastest unsigned integer type with a width of at least 64 bits (uint_fast64_t), where supported (see Attention note).

.....continued

Macro Name	Description
SCNxFAST8	Hexadecimal placeholder string for the fastest unsigned integer type with a width of at least 8 bits (`uint_fast8_t`).
SCNxFAST16	Hexadecimal placeholder string for the fastest unsigned integer type with a width of at least 16 bits (`uint_fast16_t`).
SCNxFAST32	Hexadecimal placeholder string for the fastest unsigned integer type with a width of at least 32 bits (`uint_fast32_t`).
SCNxFAST64	Octal placeholder string for the fastest unsigned integer type with a width of at least 64 bits (`uint_fast64_t`), where supported (see Attention note).
SCNoLEAST8	Octal placeholder string for a unsigned integer type with a width of at least 8 bits (`uint_least8_t`).
SCNoLEAST16	Octal placeholder string for a unsigned integer type with a width of at least 16 bits (`uint_least8_t`).
SCNoLEAST32	Octal placeholder string for a unsigned integer type with a width of at least 32 bits (`uint_least8_t`).
SCNoLEAST64	Octal placeholder string for a unsigned integer type with a width of at least 64 bits (`uint_least8_t`), where supported.
SCNuLEAST8	Unsigned decimal placeholder string for a unsigned integer type with a width of at least 8 bits (`uint_least8_t`).
SCNuLEAST16	Unsigned decimal placeholder string for a unsigned integer type with a width of at least 16 bits (`uint_least8_t`).
SCNuLEAST32	Unsigned decimal placeholder string for a unsigned integer type with a width of at least 32 bits (`uint_least8_t`).
SCNuLEAST64	Unsigned decimal placeholder string for a unsigned integer type with a width of at least 64 bits (`uint_least8_t`), where supported.
SCNxLEAST8	Hexadecimal placeholder string for a unsigned integer type with a width of at least 8 bits (`uint_least8_t`).
SCNxLEAST16	Hexadecimal placeholder string for a unsigned integer type with a width of at least 16 bits (`uint_least8_t`).
SCNxLEAST32	Hexadecimal placeholder string for a unsigned integer type with a width of at least 32 bits (`uint_least8_t`).
SCNxLEAST64	Hexadecimal placeholder string for a unsigned integer type with a width of at least 64 bits (`uint_least8_t`).
SCNoMAX	Octal placeholder string for a unsigned integer type with maximum width (`uintmax_t`).
SCNuMAX	Unsigned decimal placeholder string for a unsigned integer type with maximum width (`uintmax_t`).
SCNxMAX	Hexadecimal placeholder string for a unsigned integer type with maximum width (`uintmax_t`).
SCNoPTR	Octal placeholder string for the `uintptr_t` type.
SCNuPTR	Unsigned decimal placeholder string for the `uintptr_t` type.
SCNxPTR	Hexadecimal placeholder string for the `uintptr_t` type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

uint_least16_t ul16;
uintmax_t umax;

int
main(void)
{
  scanf("%" SCNxLEAST16, &ul16);
  scanf("%" SCNuMAX, &umax);
}
```

5.7.6 imaxabs Function

Calculate the absolute value of a greatest-width integer.

Include

`<inttypes.h>`

Prototype

`intmax_t imaxabs(intmax_t j);`

Argument

j The value whose absolute value is required.

Return Value

The `imaxabs` function computes the absolute value of an integer `j`.

Remarks

If the result cannot be represented, the behavior is undefined. The absolute value of the most negative number cannot be represented in two's compliment.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  intmax_t val;

  val = -10000;
  printf("The absolute value of %" PRIdMAX " is %" PRIdMAX "\n", val, imaxabs(val));
}
```

Example Output

```
The absolute value of -10000 is 10000
```

5.7.7 imaxdiv Function

Compute division and remainder of a greatest-width integer in one operation.

Include

`<inttypes.h>`

Prototype

```
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Arguments

numer	The numerator argument
denom	The denominator argument

Return Value

The `imaxdiv` function computes the division, `numer / denom`, and remainder, `numer % denom`, of the arguments and returns the results in an `imaxdiv_t` structure.

Remarks

If either part of the result cannot be represented, the behavior is undefined.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  intmax_t numer, denom;
  imaxdiv_t result;

  numer = 400;
  denom = 3;
  result = imaxdiv(numer, denom);
  printf("The remainder of %" PRIdMAX " divided by %" PRIdMAX " is %" PRIdMAX "\n", numer,
denom, result.rem);
}
```

Example Output

```
The remainder of 400 divided by 3 is 1
```

5.7.8 strtoimax Function

Convert string to greatest-width integer integer value.

Include

`<inttypes.h>`

Prototype

```
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);
```

Arguments

nptr	the string to attempt to convert
endptr	pointer to the remainder of the string that was not converted
base	The base of the conversion

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The `strtoimax` function attempts to convert the first part of the string pointed to by `nptr` to an `intmax_t` integer value. If the value of `base` is unsupported, `errno` will be set to `EINVAL`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char * string = "-1234abcd";
  char * final;
  intmax_t result;

  result = strtoimax(string, &final, 10);
  printf("The integer conversion of the string \"%s\" is %" PRIdMAX "; final string part is
\"%s\"\n", string, result, final);
}
```

Example Output

```
The integer conversion of the string "-1234abcd" is -1234; final string part is "abcd"
```

5.7.9 strtoumax Function

Convert a string to a greatest-width unsigned integer value.

Include

`<inttypes.h>`

Prototype

`uintmax_t strtoumax(const char * restrict` **`nptr`**`, char ** restrict` **`endptr`**`, int` **`base`**`);`

Arguments

`nptr`	the string to attempt to convert
`endptr`	pointer to the remainder of the string that was not converted
`base`	The base of the conversion

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The `strtoumax` function attempts to convert the first part of the string pointed to by `nptr` to an `uintmax_t` integer value. If the value of `base` is unsupported, `errno` will be set to `EINVAL`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char * string = "-1234abcd";
  char * final;
  uintmax_t result;

  result = strtoumax(string, &final, 10);
  printf("The integer conversion of the string \"%s\" is %" PRIuMAX "; final string part is
\"%s\"\n", string, result, final);
}
```

Example Output

```
The integer conversion of the string "-1234abcd" is 18446744073709550382; final string part
is "abcd"
```

5.7.10 wcstoimax Function

Convert wide string to a maximum-length integer value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<inttypes.h>`

Prototype

```
intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int
base);
```

Arguments

nptr	the wide string to attempt to convert
endptr	pointer to store the address of the remainder of the wide string that was not converted
base	the radix in which the value of the wide string should be interpreted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, `INTMAX_MIN` or `INTMAX_MAX` is returned based on the sign of the correct value.

Remarks

The `wcstoimax` function attempts to convert a portion of the wide string pointed to by `nptr` to a `intmax_t`value.

The initial sequence consists of any white-space wide characters.

The subject sequence represents the integer constant to convert and whose radix is given by `base`.

If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant whose radix is determined by the sequence itself. For example, a leading `0x` implies a hexadecimal constant; a leading `0` implies an octal constant. The sequence may be preceded by a plus or minus sign, but not including an integer suffix.

If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters (`a` (or `A`) through `z` (or `Z`) ascribed the values 10 through 35) and digits representing an integer with the radix specified by `base`. Only letters and digits whose ascribed values are less than that of `base` are permitted. The sequence may be preceded by a plus or minus sign, but not including an integer suffix. If the value of `base` is 16, the wide characters `0x` or `0X` may prefix the sequence of letters and digits, following the sign wide character if present.

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial sequence and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The final sequence consists of wide characters in the wide string that were unrecognized in the subject sequence and which will include the terminating null wide character of the wide string. This sequence of the wide string will be accessible via the address stored to `endptr`, provided `endptr` is not a null pointer.

The value of the macro `ERANGE` is stored in `errno` if the correct value of the conversion is outside the range of representable values.

Example

```
#include <inttypes.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
  wchar_t ws[] = L"2001 60c0c0 -110111010011010010000 7000 7000";
  wchar_t * pEnd;
  intmax_t li1, li2, li3, li4, li5;

  li1 = wcstoimax(ws, &pEnd, 10);
  li2 = wcstoimax(pEnd, &pEnd, 16);
  li3 = wcstoimax(pEnd, &pEnd, 2);
  li4 = wcstoimax(pEnd, &pEnd, 16);
  li5 = wcstoimax(pEnd, NULL, 0);
  wprintf(L"The converted string values in decimal are: %"
    PRIdMAX ", %" PRIdMAX ", %" PRIdMAX ", %" PRIdMAX ", and %" PRIdMAX ".\n", li1, li2, li3,
li4, li5);
}
```

Example Output

```
The converted string values in decimal are: 2001, 6340800, -3624224, 28672, and 7000.
```

5.7.11 wcstoumax Function

Convert wide string to a maximum-length unsigned integer value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<inttypes.h>`

Prototype

`uintmax_t wcstoumax(const wchar_t * restrict` **`nptr`**`, wchar_t ** restrict` **`endptr`**`, int` **`base`**`);`

Arguments

`nptr`	the wide string to attempt to convert
`endptr`	pointer to store the address of the remainder of the wide string that was not converted
`base`	the radix in which the value of the wide string should be interpreted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, `UINTMAX_MIN` or `UINTMAX_MAX` is returned based on the sign of the correct value.

Remarks

The `wcstoumax` function attempts to convert a portion of the wide string pointed to by `nptr` to a `uintmax_t`value.

The initial sequence consists of any white-space wide characters.

The subject sequence represents the integer constant to convert and whose radix is given by `base`.

If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant whose radix is determined by the sequence itself. For example, a leading `0x` implies a hexadecimal constant; a leading `0` implies an octal constant. The sequence may be preceded by a plus or minus sign, but not including an integer suffix.

If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters (a (or A) through z (or Z) ascribed the values 10 through 35) and digits representing an integer with the radix specified by base. Only letters and digits whose ascribed values are less than that of base are permitted. The sequence may be preceded by a plus or minus sign, but not including an integer suffix. If the value of base is 16, the wide characters 0x or 0X may prefix the sequence of letters and digits, following the sign wide character if present.

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial sequence and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

The final sequence consists of wide characters in the wide string that were unrecognized in the subject sequence and which will include the terminating null wide character of the wide string. This sequence of the wide string will be accessible via the address stored to endptr, provided endptr is not a null pointer.

The value of the macro ERANGE is stored in errno if the correct value of the conversion is outside the range of representable values.

Example

```
#include <inttypes.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
  wchar_t ws[] = L"2001 60c0c0 -110111010011010010000 7000 7000";
  wchar_t * pEnd;
  uintmax_t ui1, ui2, ui3, ui4, ui5;

  ui1 = wcstoumax(ws, &pEnd, 10);
  ui2 = wcstoumax(pEnd, &pEnd, 16);
  ui3 = wcstoumax(pEnd, &pEnd, 2);
  ui4 = wcstoumax(pEnd, &pEnd, 16);
  ui5 = wcstoumax(pEnd, NULL, 0);
  wprintf(L"The converted string values in decimal are: %"
        PRIuMAX ", %" PRIuMAX ", %" PRIuMAX ", %" PRIuMAX ", and %" PRIuMAX ".\n", ui1, ui2,
ui3, ui4, ui5);
}
```

Example Output

```
The converted string values in decimal are: 2001, 6340800, 18446744073705927392, 28672, and
7000.
```

5.8 <iso646.h> Alternate Spellings

The <iso646.h> header file consists of macros that can be used to replace the logical and bitwise operators.

5.8.1 iso6464 Alternate Spelling Macros

Macro Name	Definition
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=

.....continued

Macro Name	Definition
or	||
or_eq	|=
xor	^
xor_eq	^=

5.9 <limits.h> Implementation-Defined Limits

The header file limits.h consists of macros that define the size-related values associated with integer types.

5.9.1 Implementation-Defined Limit Macros

Include

<limits.h>

Remarks

The macros in following table relate to sizes of the integer types and expand to constant expressions that can be used in #if preprocessor directives.

The Baseline and non-enhanced Mid-range PIC devices do not support 64-bit integer types, and all the long long int integer types are 32 bits wide for these devices. For all other devices, these types are 64-bit wide, hence the macro values defined by <limits.h> associated with these types will be different, as indicated in the table.

Table 5-4. Declarations Provided by <limits.h>

Macro Name	Description	Value
CHAR_BIT	Number of bits to represent type char	8
CHAR_MAX	Maximum value of a char	When -funsigned-char is in effect (or using XC8 for PIC devices and no plain char type option has been used): 255 When -fsigned-char is in effect (or using XC8 for AVR devices, XC16, or XC32 and no plain char type option has been used): 127
CHAR_MIN	Minimum value of a char	When -funsigned-char is in effect (or using XC8 for PIC devices and no plain char type option has been used): 0 When -fsigned-char is in effect (or using XC8 for AVR devices, XC16, or XC32 and no plain char type option has been used): -128
UCHAR_MAX	Maximum value of an unsigned char	255
SCHAR_MAX	Maximum value of a signed char	127
SCHAR_MIN	Minimum value of a signed char	-128
USHRT_MAX	Maximum value of an unsigned short int	65535
SHRT_MAX	Maximum value of a short int	32767
SHRT_MIN	Minimum value of a short int	-32768

.....continued

Macro Name	Description	Value
UINT_MAX	Maximum value of an `unsigned int`	`65535`
INT_MAX	Maximum value of a `int`	MPLAB XC8: `32767` MPLAB XC16: `32767` MPLAB XC32: `2147483647`
INT_MIN	Minimum value of a `int`	MPLAB XC8: `-32768` MPLAB XC16: `-32768` MPLAB XC32: `-2147483648`
ULONG_MAX	Maximum value of a `long unsigned int`	All: `4294967295`
LONG_MAX	Maximum value of a `long int`	All: `2147483647`
LONG_MIN	Minimum value of a `long int`	All: `-2147483648`
ULLONG_MAX	Maximum value of a `long long unsigned int`	Devices lacking 64 bit support (see Remarks): `4294967295` All other devices: `18446744073709551615`
LLONG_MAX	Maximum value of a `long long int`	Devices lacking 64 bit support (see Remarks): `2147483647` All other devices: `9223372036854775807`
LLONG_MIN	Minimum value of a `long long int`	Devices lacking 64 bit support (see Remarks): `-2147483648` All other devices: `-9223372036854775808`
MB_LEN_MAX	Maximum number of bytes in a multibyte character	MPLAB XC8: `1` MPLAB XC16: `4` MPLAB XC32: `16`

5.10 <locale.h> Localization

A compiler can define one or more sets of parameters, called locales, that define local conventions of nationality, culture, and language and that affect such things as the formatting of program output. The `local.h` header provides types, macros, and functions that allow a locale to be selected and utilized. The locale in effect is called the current locale.

Attention: This header is not implemented when building with MPLAB XC8 for PIC MCUs. Only the default `"C"` locale is supported for all other devices and compilers.

5.10.1 Iconv Type

Iconv Type
A structure whose members can hold values appropriate for the formatting of numeric quantities according to the rules of the current locale.

Attention: This type is not implemented when building with MPLAB XC8 for PIC MCUs. It is implemented for all other devices and compilers.

Include

`<locale.h>`

Remarks

The members contained within this structure are tabulated below with the values relevant for the `"C"` locale.

Member	"C" Locale value
`char *decimal_point;`	`"."`
`char *thousands_sep;`	`""`
`char *grouping;`	`""`
`char *mon_decimal_point;`	`""`
`char *mon_thousands_sep;`	`""`
`char *mon_grouping;`	`""`
`char *positive_sign;`	`""`
`char *negative_sign;`	`""`
`char *currency_symbol;`	`""`
`char frac_digits;`	`CHAR_MAX`
`char p_cs_precedes;`	`CHAR_MAX`
`char n_cs_precedes;`	`CHAR_MAX`
`char p_sep_by_space;`	`CHAR_MAX`
`char n_sep_by_space;`	`CHAR_MAX`
`char p_sign_posn;`	`CHAR_MAX`
`char n_sign_posn;`	`CHAR_MAX`
`char *int_curr_symbol;`	`""`
`char int_frac_digits;`	`CHAR_MAX`
`char int_p_cs_precedes;`	`CHAR_MAX`
`char int_n_cs_precedes;`	`CHAR_MAX`
`char int_p_sep_by_space;`	`CHAR_MAX`
`char int_n_sep_by_space;`	`CHAR_MAX`
`char int_p_sign_posn;`	`CHAR_MAX`
`char int_n_sign_posn;`	`CHAR_MAX`

5.10.2 <locale.h> Types

Locale Category Macros

Macros which specify a portion of a locale, usable as the first argument to the `setlocale()` function.

Attention: These macros are not implemented when building with MPLAB XC8 for PIC MCUs. They are implemented for all other devices and compilers.

Include

`<locale.h>`

Remarks

The locale category macros defined by this header are tabulated below with that portion of a locale that they affect.

Macro	Affects
`LC_ALL`	The entire locale
`LC_COLLATE`	The behavior of the `strcoll()` and `strxfrm()` function
`LC_CTYPE`	The behavior of the character handling functions
`LC_MONETARY`	The monetary formatting information returned by the `localeconv()` function
`LC_NUMERIC`	The decimal-point character for the formatted I/O and string conversion functions; the nonmonetary formatting information returned by the `localeconv()` function
`LC_TIME`	The behavior of the `strftime()` and `wcsftime()` functions

5.10.3 NULL Macro

A constant value which represent a null pointer constant. It's value and type is implementation defined.

Include

`<locale.h>`

`<stddef.h>`

`<stdio.h>`

`<stdlib.h>`

`<string.h>`

`<time.h>`

`<wchar.h>`

Definition

```
#define NULL ((void*)0)
```

5.10.4 localeconv Function

Sets the components of an structure with locale-specific values appropriate for the formatting of numeric quantities.

Attention: This function is not implemented when building with MPLAB XC8 for PIC MCUs. It is implemented for all other devices and compilers; however, only the default `"C"` locale is supported.

Include

`<locale.h>`

Prototype

```
struct lconv * localeconv(void);
```

Return Value

The function returns a pointer to the completed object, which shall not be modified by the program, but may be overwritten by a subsequent call to this function. In addition, calls to the `setlocale()` function with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

Remarks

The function sets the components of a structure with locale-specific values appropriate for the formatting of numeric quantities. Structure members of type `char *` (with the exception of `decimal_point`) can point to `""`, to indicate that the value is not available in the current locale or is of zero length. Apart from `grouping` and `mon_grouping`, the strings shall start and end in the initial shift state. The members with type `char` are non-negative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current locale.

Those MPLAB XC compilers that implement `setlocale()` only support the `"C"`, which specifies the minimal environment for C translation.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <locale.h>

struct lconv * ll;

int main(void) {
  ll = localeconv();
  setlocale(LC_ALL, "C");
}
```

5.10.5 setlocale Function

Selects the appropriate portion of the program's locale.

Attention: This function is not implemented when building with MPLAB XC8 for PIC MCUs. It is implemented for all other devices and compilers; however, only the default `"C"` locale is supported.

Include

`<locale.h>`

Prototype

```
char * setlocale(int category, const char * locale);
```

Arguments

`category`	the portion of the locale to change, specified by macros `LC_ALL`, `LC_NUMERIC` etc.
`locale`	a string representing the environment to set

Return Value

The function returns a pointer to the string associated with the specified category for the new locale. If the selection cannot be honored, the function returns a null pointer and the program's locale is not changed. If `locale` is a null pointer, the function returns a pointer to the string associated with the category for the program's current locale.

Remarks

The function sets the environment, specified by the `locale` argument, in the portion of the program's locale, as specified by the `category` argument. If `locale` is a null pointer, the locale is not set.

Those MPLAB XC compilers that implement `setlocale()` only support the `"C"` for `locale`, which specifies the minimal environment for C translation.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <locale.h>

struct lconv * ll;

int main(void) {
  ll = localeconv();
  setlocale(LC_ALL, "C");
}
```

5.11 <math.h> Mathematical Functions

The header file `math.h` consists of a macro and various functions that calculate common mathematical operations. Error conditions may be handled with a domain error or range error (see `errno.h`).

A domain error occurs when the input argument is outside the domain over which the function is defined. The error is reported by storing the value of `EDOM` in `errno` and returning a particular value defined for each function.

A range error occurs when the result is too large or too small to be represented in the target precision. The error is reported by storing the value of `ERANGE` in `errno` and returning `HUGE_VAL`/`HUGE_VALL`/`HUGE_VALF` if the (`double`/`long double`/`float`) result overflowed (return value was too large) or a zero if the result underflowed (return value is too small).

Responses to special values, such as NaNs, zeros and infinities, may vary depending upon the function.

5.11.1 Floating-point Types

float_t
Represents the `float` type, since `FLT_EVAL_METHOD` is defined as 0.

Include

`<math.h>`

double_t
Represents the `double` type, since `FLT_EVAL_METHOD` is defined as 0.

Include

`<math.h>`

5.11.2 Math Value Macros

HUGE_VAL
This macro expands to a large positive `double` value. It is returned by math functions where a `double` result has overflowed and default rounding is in effect, or if the mathematical result is an exact infinity.

Include

`<math.h>`

Value

The value `INFINITY`.

HUGE_VALF
This macro expands to a large positive `float` value. It is returned by math functions where a `float` result has overflowed and default rounding is in effect, or if the mathematical result is an exact infinity.

Include

`<math.h>`

Value

The value `INFINITY`.

HUGE_VALL
This macro expands to a large positive `long double` value. It is returned by math functions where a `long double` result has overflowed and default rounding is in effect, or if the mathematical result is an exact infinity.

Include

`<math.h>`

Value

The value `INFINITY`.

INFINITY
This macro expands to a positive `float` constant that will overflow at compile time.

Include

`<math.h>`

NAN
This macro expands to a `float` constant that represents a NaN (not a number).

Include

`<math.h>`

5.11.3 Floating-point Classification Macros

FP_INFINITE
Floating-point classification type used by `fp_classify`, for example, indicating the value is infinity.

Include

`<math.h>`

FP_NAN
Floating-point classification type used by `fp_classify`, for example, indicating the value is not a number.

Include

`<math.h>`

FP_NORMAL
Floating-point classification type used by `fp_classify`, for example, indicating the value is normal, i.e., it is not zero, NaN, infinite, nor subnormal.

Include

`<math.h>`

FP_SUBNORMAL
Floating-point classification type used by `fp_classify`, for example, indicating the value is subnormal. Subnormal values are non-zero values, with magnitude smaller than the smallest normal value. The normalisation process shifts leading zeros out of the significand and decreases the exponent, but where this process would cause the exponent to become smaller than its smallest representable value, the leading zeros are permitted and the value becomes subnormal, or denormalised.

Include

<math.h>

FP_ZERO
Floating-point classification type used by fp_classify, for example, indicating the value is zero.

Include

<math.h>

FP_FAST_FMA
The definition of this macro indicates that the fma function executes at least as fast as the discrete multiplication and addition of double values.

Include

<math.h>

FP_FAST_FMAF
The definition of this macro indicates that the fma function executes at least as fast as the discrete multiplication and addition of float values.

Include

<math.h>

FP_FAST_FMAL
The definition of this macro indicates that the fma function executes at least as fast as the discrete multiplication and addition of long double values.

Include

<math.h>

FP_ILOGB0
This macro expands to the value returned by the ilogb function when passed an argument of zero.

Include

<math.h>

Value

Expands to the value of INT_MIN, as defined by <limits.h>

FP_ILOGBNAN
This macro expands to the value returned by the ilogb function when passed an argument of NaN.

Include

<math.h>

Value

Expands to the value of INT_MIN, as defined by <limits.h>

5.11.4 Math Error Condition Macros

MATH_ERRNO
This macro has the value 1 and is usable with math_errhandling to determine the implementation response to a domain error.

Include

`<math.h>`

Value

The value 1.

MATH_ERREXCEPT
This macro has the value 2 and is usable with `math_errhandling` to determine the implementation response to a domain error.

Include

`<math.h>`

Value

The value 2.

math_errhandling
This is a value that can be used to determine if `errno` will be set or an exception will be raised on a domain error. If the value of `math_errhandling & MATH_ERRNO` is true, then `errno` will be updated with the appropriate error number. If the value of `math_errhandling & MATH_ERREXCEPT` is true, then an invalid floating-point exception will be raised when a domain error is encountered.

Include

`<math.h>`

Value

The value 1.

5.11.5 acos Function

Calculates the trigonometric arc cosine function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double acos(double x);`

Argument

x value between -1 and 1 for which to return the arc cosine

Return Value

Returns the arc cosine of `x` in the range $[0, \pi]$ radians (inclusive) or NaN if a domain error occurs.

Remarks

If `x` is less than -1 or greater than 1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y;

  errno = 0;
  x = -2.0;
```

```
  y = acos (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n", x, y);

  errno = 0;
  x = 0.10;
  y = acos (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n", x, y);
}
```

Example Output

```
Error: Domain error
The arccosine of -2.000000 is nan
The arccosine of 0.100000 is 1.470629
```

5.11.6 acosf Function

Calculates the trigonometric arc cosine function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float acosf(float x);`

Argument

x value between -1 and 1

Return Value

Returns the arc cosine of `x` in the range $[0, \pi]$ radians (inclusive) or NaN if a domain error occurs.

Remarks

If `x` is less than -1 or greater than 1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = acosf (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0F;
  y = acosf (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n", x, y);
}
```

Example Output

```
Error: Domain error
The arccosine of 2.000000 is nan
The arccosine of 0.000000 is 1.570796
```

5.11.7 acosl Function

Calculates the trigonometric arc cosine function of a long double precision floating-point value.

Include

`<math.h>`

Prototype

```
long double acosl(long double x);
```

Argument

x value between -1 and 1 for which to return the arc cosine

Return Value

Returns the arc cosine of `x` in the range $[0, \pi]$ radians (inclusive) or NaN if a domain error occurs.

Remarks

If `x` is less than -1 or greater than 1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x,y;

  errno = 0;
  x = -2.0;
  y = acosl(x);
  if (errno)
    perror("Error");
  printf("The arccosine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 0.10;
  y = acosl(x);
  if (errno)
    perror("Error");
  printf("The arccosine of %Lf is %Lf\n\n", x, y);
}
```

Example Output

```
Error: Domain error
The arccosine of -2.000000 is nan
The arccosine of 0.100000 is 1.470629
```

5.11.8 acosh Function

Calculates the arc hyperbolic cosine function of a double precision floating-point value.

Include

`<math.h>`

Prototype

```
double acosh(double x);
```

Argument

x value for which to return the arc hyperbolic cosine

Return Value

Returns the arc hyperbolic cosine of `x` in the range $[0, \infty]$radians (inclusive) or NaN if a domain error occurs.

Remarks

If `x` is less than 1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 0.0;
  y = acosh(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %f is %f\n", x, y);

  errno = 0;
  x = 1.0;
  y = acosh(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %f is %f\n", x, y);

  errno = 0;
  x = 720.0;
  y = acosh(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %f is %f\n", x, y);
}
```

Example Output

```
Error: domain error
The arc hyperbolic cosine of 0.000000 is nan
The arc hyperbolic cosine of 1.000000 is 0.000000
The arc hyperbolic cosine of 720.000000 is 7.272398
```

5.11.9 acoshf Function

Calculates the arc hyperbolic cosine function of a single precision floating-point value.

Include

```
<math.h>
```

Prototype

```
float acoshf(float x);
```

Argument

x value for which to return the arc hyperbolic cosine

Return Value

Returns the arc hyperbolic cosine of x in the range $[0, \infty]$ radians (inclusive) or NaN if a domain error occurs.

Remarks

If x is less than 1, a domain error will occur and errno will be set to EDOM.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 0.0;
  y = acoshf(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %f is %f\n", x, y);

  errno = 0;
  x = 1.0;
  y = acoshf(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %f is %f\n", x, y);

  errno = 0;
  x = 720.0;
  y = acoshf(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %f is %f\n", x, y);
}
```

Example Output

```
Error: domain error
The arc hyperbolic cosine of 0.000000 is nan
The arc hyperbolic cosine of 1.000000 is 0.000000
The arc hyperbolic cosine of 720.000000 is 7.272398
```

5.11.10 acoshl Function

Calculates the arc hyperbolic cosine function of a long double precision floating-point value.

Include

<math.h>

Prototype

long double acoshl(long double **x**);

Argument

x value for which to return the arc hyperbolic cosine

Return Value

Returns the arc hyperbolic cosine of x in the range $[0, \infty]$ radians (inclusive) or NaN if a domain error occurs.

Remarks

If `x` is less than 1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = 0.0;
  y = acoshl(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 1.0;
  y = acoshl(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 720.0;
  y = acoshl(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic cosine of %Lf is %Lf\n", x, y);
}
```

Example Output

```
Error: domain error
The arc hyperbolic cosine of 0.000000 is nan
The arc hyperbolic cosine of 1.000000 is 0.000000
The arc hyperbolic cosine of 720.000000 is 7.272398
```

5.11.11 asin Function

Calculates the trigonometric arc sine function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double asin(double x);`

Argument

x value between -1 and 1 for which to return the arc sine

Return Value

Returns the arc sine in radians in the range $[-\pi/2, +\pi/2]$ radians (inclusive) or NaN if a domain error occurs.

Remarks

If `x` is less than -1 or greater than 1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = asin (x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = asin (x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n", x, y);
}
```

Example Output

```
Error: Domain error
The arcsine of 2.000000 is nan
The arcsine of 0.000000 is 0.000000
```

5.11.12 asinf Function

Calculates the trigonometric arc sine function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float asinf(float x);`

Argument

x value between -1 and 1 for which to return the arc sine

Return Value

Returns the arc sine in radians in the range $[-\pi/2, +\pi/2]$ radians (inclusive) or NaN if a domain error occurs.

Remarks

If `x` is less than -1 or greater than 1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = asinf(x);
```

```
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0F;
  y = asinf(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %f is %f\n", x, y);
}
```

Example Output

```
Error: Domain error
The arcsine of 2.000000 is nan
The arcsine of 0.000000 is 0.000000
```

5.11.13 asinl Function

Calculates the trigonometric arc sine function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`long double asinl(long double` **`x`**`);`

Argument

`x` value between -1 and 1 for which to return the arc sine

Return Value

Returns the arc sine in radians in the range $[-\pi/2, +\pi/2]$ radians (inclusive) or NaN if a domain error occurs.

Remarks

If `x` is less than -1 or greater than 1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = 2.0;
  y = asinl(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 0.0;
  y = asinl(x);
  if (errno)
    perror("Error");
  printf("The arcsine of %Lf is %Lf\n", x, y);
}
```

Example Output

```
Error: Domain error
The arcsine of 2.000000 is nan
The arcsine of 0.000000 is 0.000000
```

5.11.14 asinh Function

Calculates the arc hyperbolic sine function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double asinh(double x);`

Argument

x value for which to return the arc hyperbolic sine

Return Value

Returns the arc hyperbolic sine of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = asinh(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %f is %f\n", x, y);

  errno = 0;
  x = 1.0;
  y = asinh(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %f is %f\n", x, y);

  errno = 0;
  x = 720.0;
  y = asinh(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %f is %f\n", x, y);
}
```

Example Output

```
The arc hyperbolic sine of -1.000000 is -0.881374
The arc hyperbolic sine of 1.000000 is 0.881374
The arc hyperbolic sine of 720.000000 is 7.272399
```

5.11.15 asinhf Function

Calculates the arc hyperbolic sine function of a single precision floating-point value.

Include

<math.h>

Prototype

```
float asinhf(float x);
```

Argument

x value for which to return the arc hyperbolic sine

Return Value

Returns the arc hyperbolic sine of x.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = asinhf(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %f is %f\n", x, y);

  errno = 0;
  x = 1.0;
  y = asinhf(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %f is %f\n", x, y);

  errno = 0;
  x = 720.0;
  y = asinhf(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %f is %f\n", x, y);
}
```

Example Output

```
The arc hyperbolic sine of -1.000000 is -0.881374
The arc hyperbolic sine of 1.000000 is 0.881374
The arc hyperbolic sine of 720.000000 is 7.272399
```

5.11.16 asinhl Function

Calculates the arc hyperbolic sine function of a double precision floating-point value.

Include

<math.h>

Prototype

```
long double asinhl(long double x);
```

Argument

x value for which to return the arc hyperbolic sine

Return Value

Returns the arc hyperbolic sine of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = -1.0;
  y = asinhl(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 1.0;
  y = asinhl(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 720.0;
  y = asinhl(x);
  if (errno)
    perror("Error");
  printf("The arc hyperbolic sine of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The arc hyperbolic sine of -1.000000 is -0.881374
The arc hyperbolic sine of 1.000000 is 0.881374
The arc hyperbolic sine of 720.000000 is 7.272399
```

5.11.17 atan Function

Calculates the trigonometric arc tangent function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double atan(double x);`

Argument

x value for which to return the arc tangent

Return Value

Returns the arc tangent in the range of $-\pi/2$ to $+\pi/2$ radians (inclusive).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
```

```
{
  double x, y;

  x = 2.0;
  y = atan (x);
  printf("The arctangent of %f is %f\n", x, y);

  x = -1.0;
  y = atan (x);
  printf("The arctangent of %f is %f\n", x, y);
}
```

Example Output

```
The arctangent of 2.000000 is 1.107149
The arctangent of -1.000000 is -0.785398
```

5.11.18 atanf Function

Calculates the trigonometric arc tangent function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float atanf(float x);`

Argument

x value for which to return the arc tangent

Return Value

Returns the arc tangent in the range of $-\pi/2$ to $+\pi/2$ radians (inclusive).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 2.0F;
  y = atanf (x);
  printf("The arctangent of %f is %f\n", x, y);

  x = -1.0F;
  y = atanf (x);
  printf("The arctangent of %f is %f\n", x, y);
}
```

Example Output

```
The arctangent of 2.000000 is 1.107149
The arctangent of -1.000000 is -0.785398
```

5.11.19 atanl Function

Calculates the trigonometric arc tangent function of a double precision floating-point value.

Include

`<math.h>`

Prototype

```
double atanl(double x);
```

Argument

x value for which to return the arc tangent

Return Value

Returns the arc tangent in the range of $-\pi/2$ to $+\pi/2$ radians (inclusive).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = 2.0;
  y = atanl(x);
  printf("The arctangent of %Lf is %Lf\n", x, y);

  x = -1.0;
  y = atanl(x);
  printf("The arctangent of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The arctangent of 2.000000 is 1.107149
The arctangent of -1.000000 is -0.785398
```

5.11.20 atan2 Function

Calculates the trigonometric arc tangent function of y/x.

Include

```
<math.h>
```

Prototype

```
double atan2(double y, double x);
```

Arguments

y y value for which to return the arc tangent

x x value for which to return the arc tangent

Return Value

Returns the arc tangent in radians in the range $[-\pi, +\pi]$ (inclusive) with the quadrant determined by the signs of both parameters.

Remarks

A domain error occurs if both `x` and `y` are zero.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

```
int main(void)
{
  double x, y, z;

  errno = 0;
  x = 0.0;
  y = 2.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n", y, x, z);

  errno = 0;
  x = -1.0;
  y = 0.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n", y, x, z);

  errno = 0;
  x = 0.0;
  y = 0.0;
  z = atan2(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n", y, x, z);
}
```

Example Output

```
The arctangent of 2.000000/0.000000 is 1.570796
The arctangent of 0.000000/-1.000000 is 3.141593
Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

5.11.21 atan2f Function

Calculates the trigonometric arc tangent function of `y/x`.

Include

`<math.h>`

Prototype

`float atan2f(float` **`y`**`, float` **`x`**`);`

Arguments

y	y value for which to return the arc tangent
x	x value for which to return the arc tangent

Return Value

Returns the arc tangent in radians in the range $[-\pi, +\pi]$ with the quadrant determined by the signs of both parameters.

Remarks

A domain error occurs if both `x` and `y` are zero.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
```

```
{
  float x, y, z;

  errno = 0;
  x = 2.0F;
  y = 0.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n", y, x, z);

  errno = 0;
  x = 0.0F;
  y = -1.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n", y, x, z);

  errno = 0;
  x = 0.0F;
  y = 0.0F;
  z = atan2f (y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %f/%f is %f\n", y, x, z);
}
```

Example Output

```
The arctangent of 2.000000/0.000000 is 1.570796
The arctangent of 0.000000/-1.000000 is 3.141593
Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

5.11.22 atan2l Function

Calculates the trigonometric arc tangent function of `y/x`.

Include

`<math.h>`

Prototype

```
long double atan2l(long double y, long double x);
```

Arguments

y	y value for which to return the arc tangent
x	x value for which to return the arc tangent

Return Value

Returns the arc tangent in radians in the range $[-\pi, +\pi]$ (inclusive) with the quadrant determined by the signs of both parameters.

Remarks

A domain error occurs if both `x` and `y` are zero.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
```

```
  long double x, y, z;

  errno = 0;
  x = 0.0;
  y = 2.0;
  z = atan2l(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %Lf/%Lf is %Lf\n", y, x, z);

  errno = 0;
  x = -1.0;
  y = 0.0;
  z = atan2l(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %Lf/%Lf is %Lf\n", y, x, z);

  errno = 0;
  x = 0.0;
  y = 0.0;
  z = atan2l(y, x);
  if (errno)
    perror("Error");
  printf("The arctangent of %Lf/%Lf is %Lf\n", y, x, z);
}
```

Example Output

```
The arctangent of 2.000000/0.000000 is 1.570796
The arctangent of 0.000000/-1.000000 is 3.141593
Error: domain error
The arctangent of 0.000000/0.000000 is nan
```

5.11.23 atanh Function

Calculates the trigonometric arc hyperbolic tangent function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double atanh(double x);`

Argument

x value for which to return the arc hyperbolic tangent

Return Value

Returns the arc hyperbolic tangent of `x` or NaN if a domain error occurs.

Remarks

If `x` is not in the range $[-1, +1]$, a domain error occurs and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 0.5;
  y = atanh(x);
  printf("The arc hyperbolic tangent of %f is %f\n", x, y);
```

```
  x = -1.0;
  y = atanh(x);
  printf("The arc hyperbolic tangent of %f is %f\n", x, y);
}
```

Example Output

```
The arc hyperbolic tangent of 0.500000 is 0.549306
The arc hyperbolic tangent of -1.000000 is -inf
```

5.11.24 atanhf Function

Calculates the trigonometric arc hyperbolic tangent function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float atanhf(float` **`x`**`);`

Argument

`x` value for which to return the arc hyperbolic tangent

Return Value

Returns the arc hyperbolic tangent of `x` or NaN if a domain error occurs.

Remarks

If `x` is not in the range $[-1, +1]$, a domain error occurs and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 0.5;
  y = atanhf(x);
  printf("The arc hyperbolic tangent of %f is %f\n", x, y);

  x = -1.0;
  y = atanhf(x);
  printf("The arc hyperbolic tangent of %f is %f\n", x, y);
}
```

Example Output

```
The arc hyperbolic tangent of 0.500000 is 0.549306
The arc hyperbolic tangent of -1.000000 is -inf
```

5.11.25 atanhl Function

Calculates the trigonometric arc hyperbolic tangent function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`long double atanh(long double` **`x`**`);`

Argument

x value for which to return the arc hyperbolic tangent

Return Value

Returns the arc hyperbolic tangent of `x` or NaN if a domain error occurs.

Remarks

If `x` is not in the range $[-1, +1]$, a domain error occurs and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = 0.5;
  y = atanhl(x);
  printf("The arc hyperbolic tangent of %Lf is %Lf\n", x, y);

  x = -1.0;
  y = atanhl(x);
  printf("The arc hyperbolic tangent of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The arc hyperbolic tangent of 0.500000 is 0.549306
The arc hyperbolic tangent of -1.000000 is -inf
```

5.11.26 cbrt Function

Calculates the real cube root of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double cbrt(double x);`

Argument

x a non-negative floating-point value

Return Value

Returns the real cube root of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
```

```
  x = 0.0;
  y = cbrt(x);
  if (errno)
    perror("Error");
  printf("The cube root of %f is %f\n", x, y);

  errno = 0;
  x = 9.5;
  y = cbrt(x);
  if (errno)
    perror("Error");
  printf("The cube root of %f is %f\n", x, y);

  errno = 0;
  x = -25.0;
  y = cbrt(x);
  if (errno)
    perror("Error");
  printf("The cube root of %f is %f\n", x, y);
}
```

Example Output

```
The cube root of 0.000000 is 0.000000
The cube root of 9.500000 is 2.117912
The cube root of -25.000000 is -2.924018
```

5.11.27 cbrtf Function

Calculates the real cube root of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float cbrtf(float` **`x`**`);`

Argument

`x` a non-negative floating-point value

Return Value

Returns the real cube root of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 0.0;
  y = cbrtf(x);
  if (errno)
    perror("Error");
  printf("The cube root of %f is %f\n", x, y);

  errno = 0;
  x = 9.5;
  y = cbrtf(x);
  if (errno)
    perror("Error");
  printf("The cube root of %f is %f\n", x, y);
```

```
  errno = 0;
  x = -25.0;
  y = cbrtf(x);
  if (errno)
    perror("Error");
  printf("The cube root of %f is %f\n", x, y);
}
```

Example Output

```
The cube root of 0.000000 is 0.000000
The cube root of 9.500000 is 2.117912
The cube root of -25.000000 is -2.924018
```

5.11.28 cbrtl Function

Calculates the real cube root of a double precision floating-point value.

Include

`<math.h>`

Prototype

`long double cbrtl(long double` **`x`**`);`

Argument

`x` a non-negative floating-point value

Return Value

Returns the real cube root of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 0.0;
  y = cbrtl(x);
  if (errno)
    perror("Error");
  printf("The cube root of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 9.5;
  y = cbrtl(x);
  if (errno)
    perror("Error");
  printf("The cube root of %Lf is %Lf\n", x, y);

  errno = 0;
  x = -25.0;
  y = cbrtl(x);
  if (errno)
    perror("Error");
  printf("The cube root of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The cube root of 0.000000 is 0.000000
The cube root of 9.500000 is 2.117912
The cube root of -25.000000 is -2.924018
```

5.11.29 ceil Function

Calculates the ceiling of a value.

Include

`<math.h>`

Prototype

`double ceil(double x);`

Argument

x a floating-point value for which to return the ceiling

Return Value

Returns the smallest integer value greater than or equal to `x`.

Remarks

No domain or range error will occur. See `floor`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0, -1.75, -1.5, -1.25};
  double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceil (x[i]);
    printf("The ceiling for  %f is  %f\n", x[i], y);
  }
}
```

Example Output

```
The ceiling for  2.000000 is  2.000000
The ceiling for  1.750000 is  2.000000
The ceiling for  1.500000 is  2.000000
The ceiling for  1.250000 is  2.000000
The ceiling for  -2.000000 is  -2.000000
The ceiling for  -1.750000 is  -1.000000
The ceiling for  -1.500000 is  -1.000000
The ceiling for  -1.250000 is  -1.000000
```

5.11.30 ceilf Function

Calculates the ceiling of a value.

Include

`<math.h>`

Prototype

```
float ceilf(float x);
```

Argument

x a floating-point value for which to return the ceiling

Return Value

Returns the smallest integer value greater than or equal to `x`.

Remarks

No domain or range error will occur. See `floorf`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x[8] = {2.0F, 1.75F, 1.5F, 1.25F, -2.0F, -1.75F, -1.5F, -1.25F};
  float y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceilf (x[i]);
    printf("The ceiling for  %f is  %f\n", x[i], y);
  }
}
```

Example Output

```
The ceiling for  2.000000 is  2.000000
The ceiling for  1.750000 is  2.000000
The ceiling for  1.500000 is  2.000000
The ceiling for  1.250000 is  2.000000
The ceiling for  -2.000000 is  -2.000000
The ceiling for  -1.750000 is  -1.000000
The ceiling for  -1.500000 is  -1.000000
The ceiling for  -1.250000 is  -1.000000
```

5.11.31 ceill Function

Calculates the ceiling of a value.

Include

`<math.h>`

Prototype

```
long double ceill(long double x);
```

Argument

x a floating-point value for which to return the ceiling

Return Value

Returns the smallest integer value greater than or equal to `x`.

Remarks

No domain or range error will occur. See `floor`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0, -1.75, -1.5, -1.25};
  long double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = ceil (x[i]);
    printf("The ceiling for  %f is  %f\n", x[i], y);
  }
}
```

Example Output

```
The ceiling for  2.000000 is  2.000000
The ceiling for  1.750000 is  2.000000
The ceiling for  1.500000 is  2.000000
The ceiling for  1.250000 is  2.000000
The ceiling for  -2.000000 is  -2.000000
The ceiling for  -1.750000 is  -1.000000
The ceiling for  -1.500000 is  -1.000000
The ceiling for  -1.250000 is  -1.000000
```

5.11.32 copysign Function

Returns a double precision value with the magnitude of one value and the sign of another.

Include

`<math.h>`

Prototype

`double copysign(double` **`x`**`, double` **`y`**`);`

Arguments

`x`	a double precision floating-point value
`y`	value whose sign will apply to the result

Return Value

Returns the value of `x` but with the sign of `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y,z;

  x = 10.0;
  y = -3.0;
  z = copysign(x, y);
  printf("The value %f but with %f's sign is %f\n\n", x, y, z);
}
```

Example Output

```
The value 10.000000 but with -3.000000's sign is -10.000000
```

5.11.33 copysignf Function

Returns a single precision value with the magnitude of one value and the sign of another.

Include

`<math.h>`

Prototype

`float copysignf(float x, float y);`

Arguments

x a single precision floating-point value

y value whose sign will apply to the result

Return Value

Returns the value of `x` but with the sign of `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x,y,z;

  x = 10.0;
  y = -3.0;
  z = copysignf(x, y);
  printf("The value %f but with %f's sign is %f\n\n", x, y, z);
}
```

Example Output

```
The value 10.000000 but with -3.000000's sign is -10.000000
```

5.11.34 copysignl Function

Returns a long double precision value with the magnitude of one value and the sign of another.

Include

`<math.h>`

Prototype

`long double copysignl(long double x, long double y);`

Arguments

x a double precision floating-point value

y value whose sign will apply to the result

Return Value

Returns the value of `x` but with the sign of `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x,y,z;

  x = 10.0;
  y = -3.0;
  z = copysignl(x, y);
  printf("The value %Lf but with %Lf's sign is %Lf\n\n", x, y, z);
}
```

Example Output

```
The value 10.000000 but with -3.000000's sign is -10.000000
```

5.11.35 cos Function

Calculates the trigonometric cosine function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double cos(double x);`

Argument

x value for which to return the cosine

Return Value

Returns the cosine of `x` specified in radians in the range $[-1, 1]$. NaN is returned if `x` is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y;

  errno = 0;
  x = -1.0;
  y = cos (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = cos (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n", x, y);
}
```

Example Output

```
The cosine of -1.000000 is 0.540302
The cosine of 0.000000 is 1.000000
```

5.11.36 cosf Function

Calculates the trigonometric cosine function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float cosf (float` **`x`**`);`

Argument

`x` value for which to return the cosine

Return Value

Returns the cosine of `x` specified in radians in the range $[-1, 1]$. NaN is returned if `x` is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = cosf (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0F;
  y = cosf (x);
  if (errno)
    perror("Error");
  printf("The cosine of %f is %f\n", x, y);
}
```

Example Output

```
The cosine of -1.000000 is 0.540302
The cosine of 0.000000 is 1.000000
```

5.11.37 cosl Function

Calculates the trigonometric cosine function of a long double precision floating-point value.

Include

`<math.h>`

Prototype

`long double cosl(long double` **`x`**`);`

Argument

x value for which to return the cosine

Return Value

Returns the cosine of x specified in radians in the range $[-1, 1]$. NaN is returned if x is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x,y;

  errno = 0;
  x = -1.0;
  y = cosl(x);
  if (errno)
    perror("Error");
  printf("The cosine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 0.0;
  y = cosl(x);
  if (errno)
    perror("Error");
  printf("The cosine of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The cosine of -1.000000 is 0.540302
The cosine of 0.000000 is 1.000000
```

5.11.38 cosh Function

Calculates the hyperbolic cosine function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double cosh(double x);`

Argument

x value for which to return the hyperbolic cosine

Return Value

Returns the hyperbolic cosine of x. It returns infinity if x is $\pm\infty$.

Remarks

If the magnitude of x is too large, a range error occurs, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

```
int main(void)
{
  double x, y;

  errno = 0;
  x = -1.5;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n", x, y);

  errno = 0;
  x = 720.0;
  y = cosh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n", x, y);
}
```

Example Output

```
The hyperbolic cosine of -1.500000 is 2.352410
The hyperbolic cosine of 0.000000 is 1.000000
Error: range error
The hyperbolic cosine of 720.000000 is inf
```

5.11.39 coshf Function

Calculates the hyperbolic cosine function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float coshf(float` **`x`**`);`

Argument

`x` value for which to return the hyperbolic cosine

Return Value

Returns the hyperbolic cosine of `x`. It returns infinity if `x` is $\pm\infty$.

Remarks

If the magnitude of `x` is too large, a range error occurs, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = coshf (x);
```

```
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n", x, y);

  errno = 0;
  x = 720.0F;
  y = coshf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %f is %f\n", x, y);
}
```

Example Output

```
The hyperbolic cosine of -1.000000 is 1.543081
The hyperbolic cosine of 0.000000 is 1.000000
Error: range error
The hyperbolic cosine of 720.000000 is inf
```

5.11.40 coshl Function

Calculates the hyperbolic cosine function of a long double precision floating-point value.

Include

`<math.h>`

Prototype

```
long double coshl(long double x);
```

Argument

x value for which to return the hyperbolic cosine

Return Value

Returns the hyperbolic cosine of `x`. It returns infinity if `x` is ±∞.

Remarks

If the magnitude of `x` is too large, a range error occurs, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = -1.0F;
  y = coshl(x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 0.0F;
  y = coshl(x);
```

```
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 720.0F;
  y = coshl(x);
  if (errno)
    perror("Error");
  printf("The hyperbolic cosine of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The hyperbolic cosine of -1.000000 is 1.543081
The hyperbolic cosine of 0.000000 is 1.000000
Error: range error
The hyperbolic cosine of 720.000000 is inf
```

5.11.41 erf Function

Calculates the error function of the argument.

Include

`<math.h>`

Prototype

`double erf(double` **`x`**`);`

Argument

x	upper limit of summation

Return Value

Returns the error function of the argument, being the summation of the error function from zero to the argument value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 0.5;
  y = erf(x);
  printf("The summation of the error function from 0 to %f is %f\n", x, y);

  x = -0.75;
  y = erf(x);
  printf("The summation of the error function from 0 to %f is %f\n", x, y);
}
```

Example Output

```
The summation of the error function from 0 to 0.500000 is 0.520500
The summation of the error function from 0 to -0.750000 is -0.711156
```

5.11.42 erff Function

Calculates the error function of the argument.

Include

<math.h>

Prototype

```
float erff(float x);
```

Argument

x upper limit of summation

Return Value

Calculates the error function of the argument, being the summation of the error function from zero to the argument value.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 0.5;
  y = erff(x);
  printf("The summation of the error function from 0 to %f is %f\n", x, y);

  x = -0.75;
  y = erff(x);
  printf("The summation of the error function from 0 to %f is %f\n", x, y);
}
```

Example Output

```
The summation of the error function from 0 to 0.500000 is 0.520500
The summation of the error function from 0 to -0.750000 is -0.711156
```

5.11.43 erfl Function

Calculates the error function of the argument.

Include

<math.h>

Prototype

```
long double erfl(long double x);
```

Argument

x upper limit of summation

Return Value

Calculates the error function of the argument, being the summation of the error function from zero to the argument value.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
```

```
  double x, y;

  x = 0.5;
  y = erfl(x);
  printf("The summation of the error function from 0 to %f is %f\n", x, y);

  x = -0.75;
  y = erfl(x);
  printf("The summation of the error function from 0 to %f is %f\n", x, y);
}
```

Example Output

```
The summation of the error function from 0 to 0.500000 is 0.520500
The summation of the error function from 0 to -0.750000 is -0.711156
```

5.11.44 erfc Function

Calculates the complementary error function of the argument.

Include

`<math.h>`

Prototype

`double erfc(double x);`

Argument

x upper limit of summation

Return Value

Returns the complementary error function of the argument, being 1 minus the error function of the same argument.

Remarks

A range error occurs if the `x` is too large and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 0.5;
  y = erfc(x);
  printf("The complementary error function from 0 to %f is %f\n", x, y);

  x = -0.75;
  y = erfc(x);
  printf("The complementary error function from 0 to %f is %f\n", x, y);
}
```

Example Output

```
The complementary error function from 0 to 0.500000 is 0.479500
The complementary error function from 0 to -0.750000 is 1.711156
```

5.11.45 erfcf Function

Calculates the complementary error function of the argument.

Include

<math.h>

Prototype

```
float erfcf(float x);
```

Argument

x upper limit of summation

Return Value

Returns the complementary error function of the argument, being 1 minus the error function of the same argument.

Remarks

A range error occurs if the x is too large and errno will be set to ERANGE.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 0.5;
  y = erfcf(x);
  printf("The complementary error function from 0 to %f is %f\n", x, y);

  x = -0.75;
  y = erfcf(x);
  printf("The complementary error function from 0 to %f is %f\n", x, y);
}
```

Example Output

```
The complementary error function from 0 to 0.500000 is 0.479500
The complementary error function from 0 to -0.750000 is 1.711156
```

5.11.46 erfcl Function

Calculates the complementary error function of the argument.

Include

<math.h>

Prototype

```
long double erfcl(long double x);
```

Argument

x upper limit of summation

Return Value

Returns the complementary error function of the argument, being 1 minus the error function of the same argument.

Remarks

A range error occurs if the x is too large and errno will be set to ERANGE.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = 0.5;
  y = erfcl(x);
  printf("The complementary error function from 0 to %Lf is %Lf\n", x, y);

  x = -0.75;
  y = erfcl(x);
  printf("The complementary error function from 0 to %Lf is %Lf\n", x, y);
}
```

Example Output

```
The complementary error function from 0 to 0.500000 is 0.479500
The complementary error function from 0 to -0.750000 is 1.711156
```

5.11.47 exp Function

Calculates the exponential function of `x` (e^x, where `x` is a double precision floating-point value).

Include

`<math.h>`

Prototype

`double exp(double x);`

Argument

x value for which to return the exponential

Return Value

Returns the exponential of `x`. Infinity is returned on overflow; 0 is returned on underflow.

Remarks

A range error occurs if the magnitude of `x` is too large, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 1.0;
  y = exp(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = 1E3;
  y = exp(x);
  if (errno)
```

```
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = -1E3;
  y = exp(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);
}
```

Example Output

```
The exponential of 1.000000 is 2.718282
Error: range error
The exponential of 1000.000000 is inf
Error: range error
The exponential of -1000.000000 is 0.000000
```

5.11.48 expf Function

Calculates the exponential function of `x` (e^x, where `x` is a single precision floating-point value).

Include

`<math.h>`

Prototype

`float expf(float` **`x`**`);`

Argument

`x` floating-point value for which to return the exponential

Return Value

Returns the exponential of `x`. Infinity is returned on overflow; 0 is returned on underflow.

Remarks

A range error occurs if the magnitude of `x` is too large, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 1.0F;
  y = expf(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = 1.0E3F;
  y = expf(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = -1.0E3F;
  y = expf(x);
```

```
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);
}
```

Example Output

```
The exponential of 1.000000 is 2.718282
Error: range error
The exponential of 1000.000000 is inf
Error: range error
The exponential of -1000.000000 is 0.000000
```

5.11.49 expl Function

Calculates the exponential function of `x` (e^x, where `x` is a long double precision floating-point value).

Include

`<math.h>`

Prototype

`long double expl(long double` **`x`**`);`

Argument

`x` value for which to return the exponential

Return Value

Returns the exponential of `x`. Infinity is returned on overflow; 0 is returned on underflow.

Remarks

A range error occurs if the magnitude of `x` is too large, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = 1.0;
  y = expl(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = 1E3;
  y = expl(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = -1E3;
  y = expl(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);
}
```

Example Output

```
The exponential of 1.000000 is 2.718282
Error: range error
The exponential of 1000.000000 is inf
Error: range error
The exponential of -1000.000000 is 0.000000
```

5.11.50 exp2 Function

Calculates the base 2 exponential function of `x` (2^x, where `x` is a double precision floating-point value).

Include

`<math.h>`

Prototype

`double exp2(double x);`

Argument

x value for which to return the exponential

Return Value

Returns 2^x.

Remarks

A range error occurs if the magnitude of `x` is too large, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 1.0;
  y = exp2(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %f is %f\n", x, y);

  errno = 0;
  x = 10;
  y = exp2(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %f is %f\n", x, y);

  errno = 0;
  x = -10;
  y = exp2(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %f is %f\n", x, y);
}
```

Example Output

```
The base 2 exponential of 1.000000 is 2.000000
The base 2 exponential of 10.000000 is 1024.000000
The base 2 exponential of -10.000000 is 0.000977
```

5.11.51 exp2f Function

Calculates the base 2 exponential function of x (2^x, where x is a single precision floating-point value).

Include

<math.h>

Prototype

```
float exp2f(float x);
```

Argument

x value for which to return the exponential

Return Value

Returns 2^x.

Remarks

A range error occurs if the magnitude of x is too large, and errno will be set to ERANGE.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 1.0;
  y = exp2f(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %f is %f\n", x, y);

  errno = 0;
  x = 10;
  y = exp2f(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %f is %f\n", x, y);

  errno = 0;
  x = -10;
  y = exp2f(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %f is %f\n", x, y);
}
```

Example Output

```
The base 2 exponential of 1.000000 is 2.000000
The base 2 exponential of 10.000000 is 1024.000000
The base 2 exponential of -10.000000 is 0.000977
```

5.11.52 exp2l Function

Calculates the base 2 exponential function of x (2^x, where x is a long double precision floating-point value).

Include

<math.h>

Prototype

```
long double exp2l(long double x);
```

Argument

x value for which to return the exponential

Return Value

Returns 2^x.

Remarks

A range error occurs if the magnitude of `x` is too large, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = 1.0;
  y = exp2l(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 10;
  y = exp2l(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %Lf is %Lf\n", x, y);

  errno = 0;
  x = -10;
  y = exp2l(x);
  if (errno)
    perror("Error");
  printf("The base 2 exponential of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The base 2 exponential of 1.000000 is 2.000000
The base 2 exponential of 10.000000 is 1024.000000
The base 2 exponential of -10.000000 is 0.000977
```

5.11.53 expm1 Function

Calculates the exponential function of `x`, minus 1, where `x` is a double precision floating-point value).

Include

```
<math.h>
```

Prototype

```
double expm1(double x);
```

Argument

x value for which to return the exponential

Return Value

Returns the exponential of x, minus 1 (e^x- 1). On a range overflow, expm1 returns the value of the HUGE_VAL macro.

Remarks

A range error occurs if x is too large, and errno will be set to ERANGE. This function may produce more accurate results compared to the expression exp(x) - 1 when the argument has a small magnitude.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 1.0;
  y = expm1(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = 1E3;
  y = expm1(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = -1E3;
  y = expm1(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);
}
```

Example Output

```
The exponential of 1.000000 is 1.718282
Error: range error
The exponential of 1000.000000 is inf
Error: range error
The exponential of -1000.000000 is -1.000000
```

5.11.54 expm1f Function

Calculates the exponential function of x, minus 1, where x is a single precision floating-point value).

Include

<math.h>

Prototype

float expm1(float **x**);

Argument

x value for which to return the exponential

Return Value

Returns the exponential of x, minus 1 (e^x- 1). On a range overflow, expm1 returns the value of the HUGE_VALF macro.

Remarks

A range error occurs if x is too large, and errno will be set to ERANGE. This function may produce more accurate results compared to the expression exp(x) – 1 when the argument has a small magnitude.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 1.0;
  y = expm1f(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = 1E3;
  y = expm1f(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);

  errno = 0;
  x = -1E3;
  y = expm1f(x);
  if (errno)
    perror("Error");
  printf("The exponential of %f is %f\n", x, y);
}
```

Example Output

```
The exponential of 1.000000 is 1.718282
Error: range error
The exponential of 1000.000000 is inf
Error: range error
The exponential of -1000.000000 is -1.000000
```

5.11.55 expm1l Function

Calculates the exponential function of x, minus 1, where x is a long double precision floating-point value).

Include

<math.h>

Prototype

```
long double expm1(long double x);
```

Argument

x value for which to return the exponential

Return Value

Returns the exponential of x, minus 1 (e^x - 1). On a range overflow, expm1 returns the value of the HUGE_VALL macro.

Remarks

A range error occurs if x is too large, and errno will be set to ERANGE. This function may produce more accurate results compared to the expression exp(x) – 1 when the argument has a small magnitude.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = 1.0;
  y = expm1l(x);
  if (errno)
    perror("Error");
  printf("The exponential of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 1E3;
  y = expm1l(x);
  if (errno)
    perror("Error");
  printf("The exponential of %Lf is %Lf\n", x, y);

  errno = 0;
  x = -1E3;
  y = expm1l(x);
  if (errno)
    perror("Error");
  printf("The exponential of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The exponential of 1.000000 is 1.718282
Error: range error
The exponential of 1000.000000 is inf
Error: range error
The exponential of -1000.000000 is -1.000000
```

5.11.56 fabs Function

Calculates the absolute value of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double fabs(double x);`

Argument

x floating-point value for which to return the absolute value

Return Value

Returns the absolute value of `x`. A negative number is returned as positive; a positive number is unchanged.

Remarks

No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 1.75;
  y = fabs (x);
  printf("The absolute value of  %f is  %f\n", x, y);

  x = -1.5;
  y = fabs (x);
  printf("The absolute value of %f is  %f\n", x, y);
}
```

Example Output

```
The absolute value of  1.750000 is  1.750000
The absolute value of -1.500000 is  1.500000
```

5.11.57 fabsf Function

Calculates the absolute value of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float fabsf(float` **`x`**`);`

Argument

`x` floating-point value for which to return the absolute value

Return Value

Returns the absolute value of `x`. A negative number is returned as positive; a positive number is unchanged.

Remarks

No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x,y;

  x = 1.75F;
  y = fabsf (x);
  printf("The absolute value of  %f is  %f\n", x, y);

  x = -1.5F;
  y = fabsf (x);
  printf("The absolute value of %f is  %f\n", x, y);
}
```

Example Output

```
The absolute value of  1.750000 is  1.750000
The absolute value of -1.500000 is  1.500000
```

5.11.58 fabsl Function

Calculates the absolute value of a long double precision floating-point value.

Include

`<math.h>`

Prototype

```
long double fabsl(long double x);
```

Argument

x floating-point value for which to return the absolute value

Return Value

Returns the absolute value of `x`. A negative number is returned as positive; a positive number is unchanged.

Remarks

No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = 1.75;
  y = fabsl(x);
  printf("The absolute value of  %Lf is  %Lf\n", x, y);

  x = -1.5;
  y = fabsl(x);
  printf("The absolute value of %Lf is  %Lf\n", x, y);
}
```

Example Output

```
The absolute value of  1.750000 is  1.750000
The absolute value of -1.500000 is  1.500000
```

5.11.59 fdim Function

Calculates the positive difference between two double-precision floating-point arguments.

Include

`<math.h>`

Prototype

```
double fdim(double x, double y);
```

Arguments

x any double-precision floating-point number

y any double-precision floating-point number

Return Value

Returns the positive difference between the two arguments, that being `x - y` when `x` is larger than `y`, and 0 for all other values of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, z;

  errno = 0;
  x = 5.7;
  y = 2.0;
  z = fdim(x, y);
  if(errno)
    perror("Error");
  printf("The positive difference between %f and %f is %f\n", x, y, z);

  errno = 0;
  x = 3.0;
  y = 4.2;
  z = fdim(x, y);
  if(errno)
    perror("Error");
  printf("The positive difference between %f and %f is %f\n", x, y, z);
}
```

Example Output

```
The positive difference between 5.700000 and 2.000000 is 3.700000
The positive difference between 3.000000 and 4.200000 is 0.000000
```

5.11.60 fdimf Function

Calculates the positive difference between two single-precision floating-point arguments.

Include

`<math.h>`

Prototype

`float fdimf(float x, float y);`

Arguments

x any single-precision floating-point number

y any single-precision floating-point number

Return Value

Returns the positive difference between the two arguments, that being `x - y` when `x` is larger than `y`, and 0 for all other values of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y, z;

  errno = 0;
  x = 5.7;
  y = 2.0;
  z = fdimf(x, y);
  if(errno)
    perror("Error");
  printf("The positive difference between %f and %f is %f\n", x, y, z);

  errno = 0;
  x = 3.0;
  y = 4.2;
  z = fdimf(x, y);
  if(errno)
    perror("Error");
  printf("The positive difference between %f and %f is %f\n", x, y, z);
}
```

Example Output

```
The positive difference between 5.700000 and 2.000000 is 3.700000
The positive difference between 3.000000 and 4.200000 is 0.000000
```

5.11.61 fdiml Function

Calculates the positive difference between two long double-precision floating-point arguments.

Include

`<math.h>`

Prototype

`long double fdiml(long double` **`x`**`, long double` **`y`**`);`

Arguments

`x` any long double-precision floating-point number

`y` any long double-precision floating-point number

Return Value

Returns the positive difference between the two arguments, that being `x - y` when `x` is larger than `y`, and 0 for all other values of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y, z;

  errno = 0;
  x = 5.7;
  y = 2.0;
```

```
  z = fdiml(x, y);
  if(errno)
    perror("Error");
  printf("The positive difference between %Lf and %Lf is %Lf\n", x, y, z);

  errno = 0;
  x = 3.0;
  y = 4.2;
  z = fdiml(x, y);
  if(errno)
    perror("Error");
  printf("The positive difference between %Lf and %Lf is %Lf\n", x, y, z);
}
```

Example Output

```
The positive difference between 5.700000 and 2.000000 is 3.700000
The positive difference between 3.000000 and 4.200000 is 0.000000
```

5.11.62 floor Function

Calculates the floor of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double floor (double` **`x`**`);`

Argument

`x` floating-point value for which to return the floor

Return Value

Returns the largest integer value less than or equal to `x`.

Remarks

No domain or range error will occur. See `ceil`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0,
                 -1.75, -1.5, -1.25};
  double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floor (x[i]);
    printf("The ceiling for %f is %f\n", x[i], y);
  }
}
```

Example Output

```
The  floor for 2.000000 is 2.000000
The  floor for 1.750000 is 1.000000
The  floor for 1.500000 is 1.000000
The  floor for 1.250000 is 1.000000
The  floor for -2.000000 is -2.000000
The  floor for -1.750000 is -2.000000
```

```
The  floor for -1.500000 is -2.000000
The  floor for -1.250000 is -2.000000
```

5.11.63 floorf Function

Calculates the floor of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float floorf(float x);`

Argument

x floating-point value for which to return the floor

Return Value

Returns the largest integer value less than or equal to `x`.

Remarks

No domain or range error will occur. See `ceilf`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x[8] = {2.0F, 1.75F, 1.5F, 1.25F,
                -2.0F, -1.75F, -1.5F, -1.25F};
  float y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floorf (x[i]);
    printf("The floor for  %f is  %f\n", x[i], y);
  }
}
```

Example Output

```
The floor for  2.000000 is  2.000000
The floor for  1.750000 is  1.000000
The floor for  1.500000 is  1.000000
The floor for  1.250000 is  1.000000
The floor for  -2.000000 is  -2.000000
The floor for  -1.750000 is  -2.000000
The floor for  -1.500000 is  -2.000000
The floor for  -1.250000 is  -2.000000
```

5.11.64 floorl Function

Calculates the floor of a long double precision floating-point value.

Include

`<math.h>`

Prototype

`long double floor (long double x);`

Argument

x floating-point value for which to return the floor

Return Value

Returns the largest integer value less than or equal to `x`.

Remarks

No domain or range error will occur. See `ceil`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x[8] = {2.0, 1.75, 1.5, 1.25, -2.0,
                -1.75, -1.5, -1.25};
  long double y;
  int i;

  for (i=0; i<8; i++)
  {
    y = floor (x[i]);
    printf("The ceiling for %Lf is %Lf\n", x[i], y);
  }
}
```

Example Output

```
The  floor for 2.000000 is 2.000000
The  floor for 1.750000 is 1.000000
The  floor for 1.500000 is 1.000000
The  floor for 1.250000 is 1.000000
The  floor for -2.000000 is -2.000000
The  floor for -1.750000 is -2.000000
The  floor for -1.500000 is -2.000000
The  floor for -1.250000 is -2.000000
```

5.11.65 fma Function

Returns the value of `x * y + z` for double-precision floating-point values.

Include

`<math.h>`

Prototype

```
double fma(double x, double y, double z);
```

Arguments

x any double precision floating-point number

y any double precision floating-point number

z any double precision floating-point number

Return Value

Returns the value of `x * y + z`, computed as if in one operation with infinite precision and rounded to the rounding mode indicated by `FLT_ROUNDS`.

Example

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

```
int main(void)
{
  double x, y, z, m;

  x = -5.7;
  y = 2.0;
  z = 1.0;
  m = fma(x, y, z);
  printf("The multiplication of %f and %f summed with %f is %f\n\n", x, y, z, m);
}
```

Example Output

```
The multiplication of -5.700000 and 2.000000 summed with 1.000000 is -10.400000
```

5.11.66 fmaf Function

Returns the value of `x * y + z` for single-precision floating-point values.

Include

`<math.h>`

Prototype

```
float fmaf(float x, float y, float z);
```

Arguments

x any single precision floating-point number

y any single precision floating-point number

z any single precision floating-point number

Return Value

Returns the value of `x * y + z`, computed as if in one operation with infinite precision and rounded to the rounding mode indicated by `FLT_ROUNDS`.

Example

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

int main(void)
{
  float x, y, z, m;

  x = -5.7;
  y = 2.0;
  z = 1.0;
  m = fma(x, y, z);
  printf("The multiplication of %f and %f summed with %f is %f\n\n", x, y, z, m);
}
```

Example Output

```
The multiplication of -5.700000 and 2.000000 summed with 1.000000 is -10.400000
```

5.11.67 fmal Function

Returns the value of `x * y + z` for long double-precision floating-point values.

Include

`<math.h>`

Prototype

```
long double fmal(long double x, long double y, long double z);
```

Arguments

x any double precision floating-point number

y any double precision floating-point number

z any double precision floating-point number

Return Value

Returns the value of `x * y + z`, computed as if in one operation with infinite precision and rounded to the rounding mode indicated by `FLT_ROUNDS`.

Example

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

int main(void)
{
  long double x, y, z, m;

  x = -5.7;
  y = 2.0;
  z = 1.0;
  m = fmal(x, y, z);
  printf("The multiplication of %Lf and %Lf summed with %Lf is %Lf\n\n", x, y, z, m);
}
```

Example Output

```
The multiplication of -5.700000 and 2.000000 summed with 1.000000 is -10.400000
```

5.11.68 fmax Function

Returns the value of the larger double-precision argument.

Include

`<math.h>`

Prototype

`double fmax(double` **`x`**`, double` **`y`**`);`

Arguments

x any double precision floating-point number

y any double precision floating-point number

Return Value

Returns the double-precision value of the larger argument.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, z;

  x = -5.7;
  y = 2.0;
  z = fmax(x, y);
```

```
  printf("The larger of %f and %f is %f\n\n", x, y, z);
}
```

Example Output

```
The larger of -5.700000 and 2.000000 is 2.000000
```

5.11.69 fmaxf Function

Returns the value of the larger single-precision argument.

Include

`<math.h>`

Prototype

```
float fmaxf(float x, float y);
```

Arguments

x any single precision floating-point number

y any single precision floating-point number

Return Value

Returns the single-precision value of the larger argument.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y, z;

  x = -5.7;
  y = 2.0;
  z = fmaxf(x, y);
  printf("The larger of %f and %f is %f\n", x, y, z);
}
```

Example Output

```
The larger of -5.700000 and 2.000000 is 2.000000
```

5.11.70 fmaxl Function

Returns the long double-precision value of the larger argument.

Include

`<math.h>`

Prototype

```
long double fmaxl(long double x, long double y);
```

Arguments

x any long double precision floating-point number

y any long double precision floating-point number

Return Value

Returns the long double-precision value of the larger argument.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y, z;

  x = -5.7;
  y = 2.0;
  z = fmaxl(x, y);
  printf("The larger of %Lf and %Lf is %Lf\n", x, y, z);
}
```

Example Output

```
The larger of -5.700000 and 2.000000 is 2.000000
```

5.11.71 fmin Function

Returns the value of the smaller double-precision argument.

Include

`<math.h>`

Prototype

```
double fmin(double x, double y);
```

Arguments

x any double precision floating-point number

y any double precision floating-point number

Return Value

Returns the double-precision value of the smaller argument.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, z;

  x = -5.7;
  y = 2.0;
  z = fmin(x, y);
  printf("The smaller of %f and %f is %f\n", x, y, z);
}
```

Example Output

```
The smaller of -5.700000 and 2.000000 is -5.700000
```

5.11.72 fminf Function

Returns the value of the smaller single-precision argument.

Include

<math.h>

Prototype

```
float fminf(float x, float y);
```

Arguments

x any single precision floating-point number

y any single precision floating-point number

Return Value

Returns the single-precision value of the smaller argument.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y, z;

  x = -5.7;
  y = 2.0;
  z = fminf(x, y);
  printf("The smaller of %f and %f is %f\n", x, y, z);
}
```

Example Output

```
The smaller of -5.700000 and 2.000000 is -5.700000
```

5.11.73 fminl Function

Returns the value of the smaller long double-precision argument.

Include

<math.h>

Prototype

```
long double fminl(long double x, long double y);
```

Arguments

x any long double precision floating-point number

y any long double precision floating-point number

Return Value

Returns the long double-precision value of the smaller argument.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y, z;
```

```
  x = -5.7;
  y = 2.0;
  z = fminl(x, y);
  printf("The smaller of %Lf and %Lf is %Lf\n", x, y, z);
}
```

Example Output

```
The smaller of -5.700000 and 2.000000 is -5.700000
```

5.11.74 fmod Function

Calculates the remainder of the division between two double-precision floating-point values.

Include

`<math.h>`

Prototype

`double fmod(double` **`x`**`, double` **`y`**`);`

Arguments

x a double precision floating-point value

y a double precision floating-point value

Return Value

Returns the remainder of `x/y` with the same sign as `x` and magnitude less than the magnitude of `y`, or returns NaN if `y` is zero.

Remarks

If `y` is zero, a domain error occurs, and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y,z;

  errno = 0;
  x = 7.0;
  y = 3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n", x, y, z);

  errno = 0;
  x = 7.0;
  y = 7.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n", x, y, z);

  errno = 0;
  x = -5.0;
  y = 3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n", x, y, z);
```

```
  errno = 0;
  x = 5.0;
  y = -3.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n", x, y, z);

  errno = 0;
  x = -5.0;
  y = -5.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n", x, y, z);

  errno = 0;
  x = 7.0;
  y = 0.0;
  z = fmod(x, y);
  if (errno)
    perror("Error");
  printf("For fmod(%f, %f) the remainder is %f\n", x, y, z);
}
```

Example Output

```
For fmod(7.000000, 3.000000) the remainder is 1.000000
For fmod(7.000000, 7.000000) the remainder is 0.000000
For fmod(-5.000000, 3.000000) the remainder is -2.000000
For fmod(5.000000, -3.000000) the remainder is 2.000000
For fmod(-5.000000, -5.000000) the remainder is -0.000000
Error: domain error
For fmod(7.000000, 0.000000) the remainder is nan
```

5.11.75 fmodf Function

Calculates the remainder of the division between two single-precision floating-point values.

Include

`<math.h>`

Prototype

```
float fmodf(float x, float y);
```

Arguments

x a double precision floating-point value

y a double precision floating-point value

Return Value

Returns the remainder of `x/y` with the same sign as `x` and magnitude less than the magnitude of `y`, or returns NaN if `y` is zero.

Remarks

If `y` is zero, a domain error occurs, and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
```

```
{
  float x,y,z;

  errno = 0;
  x = 7.0F;
  y = 3.0F;
  z = fmodf(x, y);
  if(errno)
    perror("Error");
  printf("For fmodf(%f, %f) the remainder is"
         " %f\n\n", x, y, z);

  errno = 0;
  x = -5.0F;
  y = 3.0F;
  z = fmodf(x, y);
  if(errno)
    perror("Error");
     printf("For fmodf(%f, %f) the remainder is %f\n", x, y, z);

  errno = 0;
  x = 5.0F;
  y = -3.0F;
  z = fmodf(x, y);
  if(errno)
    perror("Error");
  printf("For fmodf(%f, %f) the remainder is %f\n", x, y, z);

  errno = 0;
  x = 5.0F;
  y = -5.0F;
  z = fmodf (x, y);
  if(errno)
    perror("Error");
  printf("For fmodf (%f, %f) the remainder is %f\n", x, y, z);

  errno = 0;
  x = 7.0F;
  y = 0.0F;
  z = fmodf(x, y);
  if(errno)
    perror("Error");
  printf("For fmodf(%f, %f) the remainder is %f\n", x, y, z);

  errno = 0;
  x = 7.0F;
  y = 7.0F;
  z = fmodf(x, y);
  if(errno)
    perror("Error");
  printf("For fmodf(%f, %f) the remainder is %f\n", x, y, z);
}
```

Example Output

```
For fmodf (7.000000, 3.000000) the remainder is 1.000000
For fmodf (-5.000000, 3.000000) the remainder is -2.000000
For fmodf (5.000000, -3.000000) the remainder is 2.000000
For fmodf (5.000000, -5.000000) the remainder is 0.000000
Error: domain error
For fmodf (7.000000, 0.000000) the remainder is nan
For fmodf (7.000000, 7.000000) the remainder is 0.000000
```

5.11.76 fmodl Function

Calculates the remainder of the division between two long double-precision floating-point values.

Include

```
<math.h>
```

Prototype

```
long double fmodl(long double x, long double y);
```

Arguments

x a long double precision floating-point value

y a long double precision floating-point value

Return Value

Returns the remainder of `x/y` with the same sign as `x` and magnitude less than the magnitude of `y`, or returns NaN if `y` is zero.

Remarks

If `y` is zero, a domain error occurs, and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x,y,z;

  errno = 0;
  x = 7.0;
  y = 3.0;
  z = fmodl(x, y);
  if (errno)
    perror("Error");
  printf("For fmodl(%Lf, %Lf) the remainder is %Lf\n", x, y, z);

  errno = 0;
  x = 7.0;
  y = 7.0;
  z = fmodl(x, y);
  if (errno)
    perror("Error");
  printf("For fmodl(%Lf, %Lf) the remainder is %Lf\n", x, y, z);

  errno = 0;
  x = -5.0;
  y = 3.0;
  z = fmodl(x, y);
  if (errno)
    perror("Error");
  printf("For fmodl(%Lf, %Lf) the remainder is %Lf\n", x, y, z);

  errno = 0;
  x = 5.0;
  y = -3.0;
  z = fmodl(x, y);
  if (errno)
    perror("Error");
  printf("For fmodl(%Lf, %Lf) the remainder is %Lf\n", x, y, z);

  errno = 0;
  x = -5.0;
  y = -5.0;
  z = fmodl(x, y);
  if (errno)
    perror("Error");
  printf("For fmodl(%Lf, %Lf) the remainder is %Lf\n", x, y, z);

  errno = 0;
  x = 7.0;
  y = 0.0;
  z = fmodl(x, y);
  if (errno)
    perror("Error");
```

```
  printf("For fmodl(%Lf, %Lf) the remainder is %Lf\n", x, y, z);
}
```

Example Output

```
For fmod(7.000000, 3.000000) the remainder is 1.000000
For fmod(7.000000, 7.000000) the remainder is 0.000000
For fmod(-5.000000, 3.000000) the remainder is -2.000000
For fmod(5.000000, -3.000000) the remainder is 2.000000
For fmod(-5.000000, -5.000000) the remainder is -0.000000
Error: domain error
For fmod(7.000000, 0.000000) the remainder is nan
```

5.11.77 fpclassify Macro

Classifies its argument as one of several floating-point categories.

Include

`<math.h>`

Prototype

`int fpclassify(`*`floating-point`* **`x`**`);`

Argument

x	any floating-point number

Return Value

Classifies its argument as one of several floating-point categories, such as NaN, infinite, normal, subnormal, zero, or as another implementation-defined category, represented by the floating-point classification macros discussed in 5.11.3 Floating-point Classification Macros.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;
  int b;
  x = 0.0;
  b = fpclassify(x);
  if(b == FP_ZERO)
    printf("The value %f has been classified as a floating-point zero\n", x);
}
```

Example Output

```
The value 0.000000 has been classified as a floating-point zero
```

5.11.78 frexp Function

Gets the fraction and the exponent of a double precision floating-point number.

Include

`<math.h>`

Prototype

`double frexp(double` **`x`**`, int *` **`exp`**`);`

Arguments

x floating-point value for which to return the fraction and exponent

exp pointer to a stored integer exponent

Return Value

Returns the fraction, `exp` points to the exponent. If `x` is 0, the function returns 0 for both the fraction and exponent.

Remarks

The absolute value of the fraction is in the range of 1/2 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y;
  int n;

  x = 50.0;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ", x, y);
  printf("  and the exponent is %d\n\n", n);

  x = -2.5;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ", x, y);
  printf("  and the exponent is %d\n\n", n);

  x = 0.0;
  y = frexp (x, &n);
  printf("For frexp of %f\n the fraction is %f\n ", x, y);
  printf("  and the exponent is %d\n\n", n);
}
```

Example Output

```
For frexp of 50.000000
  the fraction is 0.781250
   and the exponent is 6

For frexp of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexp of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

5.11.79 frexpf Function

Gets the fraction and the exponent of a single precision floating-point number.

Include

`<math.h>`

Prototype

`float frexpf(float` **`x`**`, int *` **`exp`**`);`

Arguments

x floating-point value for which to return the fraction and exponent

exp pointer to a stored integer exponent

Return Value

Returns the fraction, `exp` points to the exponent. If `x` is 0, the function returns 0 for both the fraction and exponent.

Remarks

The absolute value of the fraction is in the range of 1/2 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x,y;
  int n;

  x = 0.15F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ", x, y);
  printf("  and the exponent is %d\n\n", n);

  x = -2.5F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ", x, y);
  printf("  and the exponent is %d\n\n", n);

  x = 0.0F;
  y = frexpf (x, &n);
  printf("For frexpf of %f\n  the fraction is %f\n ", x, y);
  printf("  and the exponent is %d\n\n", n);
}
```

Example Output

```
For frexpf of 0.150000
  the fraction is 0.600000
   and the exponent is -2

For frexpf of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexpf of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

5.11.80 frexpl Function

Gets the fraction and the exponent of a long double precision floating-point number.

Include

`<math.h>`

Prototype

```
long double frexpl(long double x, int * exp);
```

Arguments

x floating-point value for which to return the fraction and exponent

exp pointer to a stored integer exponent

Return Value

Returns the fraction, `exp` points to the exponent. If `x` is 0, the function returns 0 for both the fraction and exponent.

Remarks

The absolute value of the fraction is in the range of 1/2 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x,y;
  int n;

  x = 50.0;
  y = frexpl(x, &n);
  printf("For frexpl of %Lf\n the fraction is %Lf\n ", x, y);
  printf("  and the exponent is %d\n\n", n);

  x = -2.5;
  y = frexpl(x, &n);
  printf("For frexpl of %Lf\n the fraction is %Lf\n ", x, y);
  printf("  and the exponent is %d\n\n", n);

  x = 0.0;
  y = frexpl(x, &n);
  printf("For frexpl of %Lf\n the fraction is %Lf\n ", x, y);
  printf("  and the exponent is %d\n\n", n);
}
```

Example Output

```
For frexpl of 50.000000
  the fraction is 0.781250
   and the exponent is 6

For frexpl of -2.500000
  the fraction is -0.625000
   and the exponent is 2

For frexpl of 0.000000
  the fraction is 0.000000
   and the exponent is 0
```

5.11.81 hypot Function

Calculates the square root of the sum of squared double precision floating-point values.

Include

`<math.h>`

Prototype

`double hypot(double` **`x`**`, double` **`y`**`);`

Arguments

x	first argument, or length of one side
y	second argument, or length of other side

Return Value

Returns the square root of a sum of the arguments squared, being the hypotenuse of a right-angled triangle with perpendicular sides of length `x` and `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, h;

  x = 1.75;
  y = 2.05
  h = hypot(x, y);
  printf("The hypotenuse of a right-angle triangle with other side lengths of %f and %f is
%f\n", x, y, h);
}
```

Example Output

```
The hypotenuse of a right-angle triangle with other side lengths of 1.750000 and 2.050000 is
2.695366
```

5.11.82 hypotf Function

Calculates the square root of the sum of squared single precision floating-point values.

Include

`<math.h>`

Prototype

`float hypotf(float` **`x`**`, float` **`y`**`);`

Arguments

x	first argument, or length of one side
y	second argument, or length of other side

Return Value

Returns the square root of a sum of the arguments squared, being the hypotenuse of a right-angled triangle with perpendicular sides of length `x` and `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y, h;

  x = 1.75;
  y = 2.05
  h = hypotf(x, y);
  printf("The hypotenuse of a right-angle triangle with other side lengths of %f and %f is
%f\n", x, y, h);
}
```

Example Output

```
The hypotenuse of a right-angle triangle with other side lengths of 1.750000 and 2.050000 is
2.695366
```

5.11.83 hypotl Function

Calculates the square root of the sum of squared long double precision floating-point values.

Include

`<math.h>`

Prototype

`long double hypotl(long double` **`x`**`, long double` **`y`**`);`

Arguments

`x` first argument, or length of one side

`y` second argument, or length of other side

Return Value

Returns the square root of a sum of the arguments squared, being the hypotenuse of a right-angled triangle with perpendicular sides of length `x` and `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y, h;

  x = 1.75;
  y = 2.05
  h = hypotl(x, y);
  printf("The hypotenuse of a right-angle triangle with other side lengths of %Lf and %Lf is
%Lf\n", x, y, h);
}
```

Example Output

```
The hypotenuse of a right-angle triangle with other side lengths of 1.750000 and 2.050000 is
2.695366
```

5.11.84 ilogb Function

Calculates the signed integer exponent of a double precision floating-point value.

Include

`<math.h>`

Prototype

`int ilogb(double` **`x`**`);`

Argument

`x` any positive value for which to return the exponent

Return Value

Returns the exponent of `x` as a signed integer value. If `x` is 0, it returns the value `FP_ILOGB0`; if `x` is infinite, it returns the value `INT_MAX`; if `x` is a NaN it returns the value `FP_ILOGBNAN`; otherwise, this will yield the same value as the corresponding `logb` function cast to type `int`.

Remarks

A range error might occur if `x` is 0.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;
  int y;

  errno = 0;
  x = 13.45;
  y = ilogb(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);

  errno = 0;
  x = 0.0;
  y = ilogb(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);

  errno = 0;
  x = -2.0;
  y = ilogb(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);
}
```

Example Output

```
The exponent of 13.450000 is 3
The exponent of 0.000000 is -2147483648
The exponent of -2.000000 is 1
```

5.11.85 ilogbf Function

Calculates the signed integer exponent of a single precision floating-point value.

Include

`<math.h>`

Prototype

```
int ilogbf(float x);
```

Argument

x any positive value for which to return the exponent

Return Value

Returns the exponent of `x` as a signed integer value. If `x` is 0, it returns the value `FP_ILOGB0`; if `x` is infinite, it returns the value `INT_MAX`; if `x` is a NaN it returns the value `FP_ILOGBNAN`; otherwise, this will yield the same value as the corresponding `logb` function cast to type `int`.

Remarks

A range error might occur if `x` is 0.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x;
  int y;

  errno = 0;
  x = 13.45;
  y = ilogbf(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);

  errno = 0;
  x = 0.0;
  y = ilogbf(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);

  errno = 0;
  x = -2.0;
  y = ilogbf(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);
}
```

Example Output

```
The exponent of 13.450000 is 3
The exponent of 0.000000 is -2147483648
The exponent of -2.000000 is 1
```

5.11.86 ilogbl Function

Calculates the signed integer exponent of a long double precision floating-point value.

Include

`<math.h>`

Prototype

```
int ilogbl(long double x);
```

Argument

x any positive value for which to return the exponent

Return Value

Returns the exponent of `x` as a signed integer value. If `x` is 0, it returns the value `FP_ILOGB0`; if `x` is infinite, it returns the value `INT_MAX`; if `x` is a NaN it returns the value `FP_ILOGBNAN`; otherwise, this will yield the same value as the corresponding `logb` function cast to type `int`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;
  int y;

  errno = 0;
  x = 13.45;
  y = ilogbl(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);

  errno = 0;
  x = 0.0;
  y = ilogbl(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);

  errno = 0;
  x = -2.0;
  y = ilogbl(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %d\n", x, y);
}
```

Example Output

```
The exponent of 13.450000 is 3
The exponent of 0.000000 is -2147483648
The exponent of -2.000000 is 1
```

5.11.87 isfinite Macro

Returns true if its argument is finite.

Include

`<math.h>`

Prototype

`int isfinite(`*`floating-point`* `x);`

Argument

x any floating-point number

Return Value

Returns true if `x` is finite, i.e. is not infinite or NaN.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;
  int b;
```

```
  x = 0.0;
  b = isfinite(x);
  printf("The value %f is %sconsidered finite\n", x, b ? "" : "not ");
}
```

Example Output

```
The value 0.000000 is considered finite
```

5.11.88 isgreater Macro

Determines if its first argument is larger than its second.

Include

`<math.h>`

Prototype

`int isgreater(floating-point x, floating-point y);`

Argument

x any floating-point number

y any floating-point number

Return Value

Determines if `x` is larger than `y`, as if by the expression `(x) > (y)` only without any invalid floating-point exception should the arguments be unordered (i.e. should one of them be NaN).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;
  int b;

  x = -5.7;
  y = 2.0;
  b = isgreater(x, y);
  printf("That %f is greater than %f is %s\n", x, y, b ? "true" : "false");
}
```

Example Output

```
That -5.700000 is greater than 2.000000 is false
```

5.11.89 isgreaterequal Macro

Determines if its first argument is larger than or equal to its second.

Include

`<math.h>`

Prototype

`int isgreaterequal(floating-point x, floating-point y);`

Argument

x any floating-point number

y any floating-point number

Return Value

Determines if `x` is larger than or equal to `y`, as if by the expression `(x) >= (y)` only without any invalid floating-point exception should the arguments be unordered (i.e. should one of them be NaN).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;
  int b;

  x = -5.7;
  y = 2.0;
  b = isgreaterequal(x, y);
  printf("That %f is greater than or equal %f is %s\n", x, y, b ? "true" : "false");
}
```

Example Output

```
That -5.700000 is greater than or equal 2.000000 is false
```

5.11.90 isinf Macro

Returns true if its argument is an infinity.

Include

`<math.h>`

Prototype

`int isinf(`*`floating-point`* `x);`

Argument

x any floating-point number

Return Value

Returns true if `x` is either positive or negative infinity; zero otherwise.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, z;

  x = 5.0;
  y = 0.0;
  z = x / y;
  if(isinf(z))
    printf("Infinity detected in division of %f by %f\n", x, y);
}
```

Example Output

```
Infinity detected in division of 5.000000 by 0.000000
```

5.11.91 isless Macro

Determines if its first argument is smaller than its second.

Include

<math.h>

Prototype

int isless(*floating-point* x, *floating-point* y);

Argument

x any floating-point number

y any floating-point number

Return Value

Determines if x is smaller than y, as if by the expression (x) < (y) only without any invalid floating-point exception should the arguments be unordered (i.e. should one of them be NaN).

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  double x, y;
  int b;

  x = -5.7;
  y = 2.0;
  b = isless(x, y);
  printf("That %f is less than %f is %s\n", x, y, b ? "true" : "false");
}
```

Example Output

```
That -5.700000 is less than 2.000000 is true
```

5.11.92 islessequal Macro

Determines if its first argument is smaller than or equal to its second.

Include

<math.h>

Prototype

int islessequal(*floating-point* x, *floating-point* y);

Argument

x any floating-point number

y any floating-point number

Return Value

Determines if `x` is smaller than or equal to `y`, as if by the expression `(x) <= (y)` only without any invalid floating-point exception should the arguments be unordered (i.e. should one of them be NaN).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;
  int b;

  x = -5.7;
  y = 2.0;
  b = islessequal(x, y);
  printf("That %f is less than or equal to %f is %s\n", x, y, b ? "true" : "false");
}
```

Example Output

```
That -5.700000 is less than or equal to 2.000000 is true
```

5.11.93 islessgreater Macro

Determines if its first argument is smaller than or larger than its second.

Include

`<math.h>`

Prototype

`int islessgreater(`*`floating-point`* `x, `*`floating-point`* `y);`

Arguments

x	any floating-point number
y	any floating-point number

Return Value

Determines if `x` is smaller than or greater than `y`, as if by the expression `(x) < (y) || (x) > (y)` only without any invalid floating-point exception should the arguments be unordered (i.e. should one of them be NaN).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;
  int b;

  x = -5.7;
  y = 2.0;
  b = islessgreater(x, y);
  printf("That %f is less than or greater than %f is %s\n", x, y, b ? "true" : "false");
}
```

Example Output

```
That -5.700000 is less than or greater than 2.000000 is true
```

5.11.94 isnan Macro

Returns true if its argument is NaN.

Include

`<math.h>`

Prototype

`int isnan(floating-point x);`

Argument

x any floating-point number

Return Value

Returns true if `x` is NaN (not a number); false otherwise.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, z;

  x = 0.0;
  y = 0.0;
  z = x / y;
  if(isnan(z))
    printf("NaN detected in division of %f by %f\n", x, y);
}
```

Example Output

```
NaN detected in division of 0.000000 by 0.000000
```

5.11.95 isnormal Macro

Returns true if its argument is normal.

Include

`<math.h>`

Prototype

`int isnormal(floating-point x);`

Argument

x any floating-point number

Return Value

Returns true if `x` is normal, i.e. it is not NaN, infinite, subnormal, nor zero; it returns zero otherwise.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, z;

  x = 5.0;
  y = 2.0;
  z = x / y;
  if(isnormal(z))
    printf("The division of %f by %f is normal\n", x, y);
}
```

Example Output

```
The division of 5.000000 by 2.000000 is normal
```

5.11.96 isunordered Macro

Determines if its arguments are unordered.

Include

`<math.h>`

Prototype

`int isunordered(`*`floating-point`* `x, `*`floating-point`* `y);`

Arguments

x	any floating-point number
y	any floating-point number

Return Value

Returns 1 if its arguments are unordered (i.e. one or both of them are NaN) or 0 if they are not.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;
  int b;

  x = -5.7;
  y = nan(NULL);
  b = isunordered(x, y);
  printf("The arguments %f and %f are %s\n\n", x, y, b ? "unordered" : "ordered");
}
```

Example Output

```
The arguments -5.700000 and nan are unordered
```

5.11.97 ldexp Function

Calculates the result of a double precision floating-point number multiplied by an exponent of 2.

Include

<math.h>

Prototype

double ldexp(double **x**, int **exp**);

Arguments

x	floating-point value
exp	integer exponent

Return Value

Returns x * 2^exp.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  double x,y;
  int n;

  x = -0.625;
  n = 2;
  y = ldexp(x, n);
  printf("For a number = %f and an exponent = %d\n", x, n);
  printf("  ldexp(%f, %d) = %f\n\n", x, n, y);

  x = 2.5;
  n = 3;
  y = ldexp(x, n);
  printf("For a number = %f and an exponent = %d\n", x, n);
  printf("  ldexp(%f, %d) = %f\n\n", x, n, y);

  x = 15.0;
  n = 10000;
  y = ldexp(x, n);
  printf("For a number = %f and an exponent = %d\n", x, n);
  printf("  ldexp(%f, %d) = %f\n\n", x, n, y);
}
```

Example Output

```
For a number = -0.625000 and an exponent = 2
  ldexp(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexp(2.500000, 3) = 20.000000

For a number = 15.000000 and an exponent = 10000
  ldexp(15.000000, 10000) = inf
```

5.11.98 ldexpf Function

Calculates the result of a single precision floating-point number multiplied by an exponent of 2.

Include

<math.h>

Prototype

float ldexpf(float **x**, int **exp**);

Arguments

x	floating-point value
exp	integer exponent

Return Value

Returns `x` * 2^`exp`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x,y;
  int n;

  x = -0.625F;
  n = 2;
  y = ldexpf(x, n);
  printf("For a number = %f and an exponent = %d\n", x, n);
  printf("  ldexpf(%f, %d) = %f\n\n", x, n, y);

  x = 2.5F;
  n = 3;
  y = ldexpf(x, n);
  printf("For a number = %f and an exponent = %d\n", x, n);
  printf("  ldexpf(%f, %d) = %f\n\n", x, n, y);

  x = 15.0F;
  n = 10000;
  y = ldexpf(x, n);
  printf("For a number = %f and an exponent = %d\n", x, n);
  printf("  ldexpf(%f, %d) = %f\n\n", x, n, y);
}
```

Example Output

```
For a number = -0.625000 and an exponent = 2
  ldexpf(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexpf(2.500000, 3) = 20.000000

For a number = 15.000000 and an exponent = 10000
  ldexpf(15.000000, 10000) = inf
```

5.11.99 ldexpl Function

Calculates the result of a long double precision floating-point number multiplied by an exponent of 2.

Include

`<math.h>`

Prototype

`long double ldexpf(long double` **`x`**`, int` **`exp`**`);`

Arguments

x	floating-point value
exp	integer exponent

Return Value

Returns x * 2^exp

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  long double x,y;
  int n;

  x = -0.625;
  n = 2;
  y = ldexpl(x, n);
  printf("For a number = %Lf and an exponent = %d\n", x, n);
  printf("  ldexpl(%Lf, %d) = %Lf\n\n", x, n, y);

  x = 2.5;
  n = 3;
  y = ldexpl(x, n);
  printf("For a number = %Lf and an exponent = %d\n", x, n);
  printf("  ldexpl(%Lf, %d) = %Lf\n\n", x, n, y);

  x = 15.0;
  n = 10000;
  y = ldexpl(x, n);
  printf("For a number = %Lf and an exponent = %d\n", x, n);
  printf("  ldexp(%Lf, %d) = %Lf\n\n", x, n, y);
}
```

Example Output

```
For a number = -0.625000 and an exponent = 2
  ldexp(-0.625000, 2) = -2.500000

For a number = 2.500000 and an exponent = 3
  ldexp(2.500000, 3) = 20.000000

For a number = 15.000000 and an exponent = 10000
  ldexp(15.000000, 10000) = inf
```

5.11.100 lgamma Function

Calculates the natural logarithm of the absolute value of gamma of the double-precision floating-point argument.

Include

<math.h>

Prototype

```
double lgamma(double x);
```

Argument

x value to evaluate the natural logarithm of the absolute value of gamma

Return Value

Returns the natural logarithm of the absolute value of gamma of the argument.

Remarks

A range error occurs if x is too large, and errno will be set to ERANGE.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 0.5;
  y = lgamma(x);
  if(errno)
    perror("Error");
  printf("The natural log of gamma of %f is %f\n", x, y);

  x = -0.75;
  y = lgamma(x);
  if(errno)
    perror("Error");
  printf("The natural log of gamma of %f is %f\n", x, y);
}
```

Example Output

```
The natural log of gamma of 0.500000 is 0.572365
The natural log of gamma of -0.750000 is 1.575705
```

5.11.101 lgammaf Function

Calculates the natural logarithm of the absolute value of gamma of the single-precision floating-point argument.

Include

`<math.h>`

Prototype

`float lgammaf(float` **`x`**`);`

Argument

`x` value to evaluate the natural logarithm of the absolute value of gamma

Return Value

Returns the natural logarithm of the absolute value of gamma of the argument.

Remarks

A range error occurs if `x` is too large, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 0.5;
  y = lgammaf(x);
  if(errno)
    perror("Error");
  printf("The natural log of gamma of %f is %f\n", x, y);
```

```
  x = -0.75;
  y = lgammaf(x);
  if(errno)
    perror("Error");
  printf("The natural log of gamma of %f is %f\n", x, y);
}
```

Example Output

```
The natural log of gamma of 0.500000 is 0.572365
The natural log of gamma of -0.750000 is 1.575705
```

5.11.102 lgammal Function

Calculates the natural logarithm of the absolute value of gamma of the long double-precision floating-point argument.

Include

`<math.h>`

Prototype

`long double lgammal(long double` **`x`**`);`

Argument

x value to evaluate the natural logarithm of the absolute value of gamma

Return Value

Returns the natural logarithm of the absolute value of gamma of the argument.

Remarks

A range error occurs if `x` is too large, and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = 0.5;
  y = lgammal(x);
  if(errno)
    perror("Error");
  printf("The natural log of gamma of %Lf is %Lf\n", x, y);

  x = -0.75;
  y = lgammal(x);
  if(errno)
    perror("Error");
  printf("The natural log of gamma of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The natural log of gamma of 0.500000 is 0.572365
The natural log of gamma of -0.750000 is 1.575705
```

5.11.103 llrint Function

Returns the double precision floating-point argument rounded to the nearest integer value.

Include

<math.h>

Prototype

```
long long int llrint(double x);
```

Argument

x the value to round

Return Value

Returns the value of x rounded to the nearest integer value using the current rounding direction. The rounded valued is returned as a long long integer value, but is unspecified if the rounded value is outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  double x;
  long long int y;

  x = 10.103;
  y = llrint(x);
  printf("The nearest integer value to %f is %lld\n", x, y);

  x = 10.51;
  y = llrint(x);
  printf("The nearest integer value to %f is %lld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.510000 is 11
```

5.11.104 llrintf Function

Returns the single precision floating-point argument rounded to the nearest integer value.

Include

<math.h>

Prototype

```
long long int llrintf(float x);
```

Argument

x the value to round

Return Value

Returns the value of x rounded to the nearest integer value using the current rounding direction. The rounded valued is returned as a long long integer value, but is unspecified if the rounded value is outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x;
  long long int y;

  x = 10.103;
  y = llrintf(x);
  printf("The nearest integer value to %f is %lld\n", x, y);

  x = 10.51;
  y = llrintf(x);
  printf("The nearest integer value to %f is %lld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.510000 is 11
```

5.11.105 llrintl Function

Returns the long double precision floating-point argument rounded to the nearest integer value.

Include

`<math.h>`

Prototype

`long long int llrintl(long double` **`x`**`);`

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value using the current rounding direction. The rounded valued is returned as a `long long` integer value, but is unspecified if the rounded value is outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x;
  long long int y;

  x = 10.103;
  y = llrintl(x);
  printf("The nearest integer value to %Lf is %lld\n", x, y);

  x = 10.51;
  y = llrintl(x);
  printf("The nearest integer value to %Lf is %lld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.510000 is 11
```

5.11.106 llround Function

Returns the double precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

`long long int llround(double x);`

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded value is returned as a `long long` integer value, but is unspecified should the rounded value fall outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;
  long long int y;

  x = 10.103;
  y = llround(x);
  printf("The nearest integer value to %f is %lld\n", x, y);

  x = 10.5;
  y = llround(x);
  printf("The nearest integer value to %f is %lld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.500000 is 11
```

5.11.107 llroundf Function

Returns the single precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

`long long int llroundf(float x);`

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded value is returned as a `long long` integer value, but is unspecified should the rounded value fall outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x;
  long long int y;

  x = 10.103;
  y = llroundf(x);
  printf("The nearest integer value to %f is %lld\n", x, y);

  x = 10.5;
  y = llroundf(x);
  printf("The nearest integer value to %f is %lld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.500000 is 11
```

5.11.108 llroundl Function

Returns the long double precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

`long long int llroundl(long double` **`x`**`);`

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded value is returned as a `long long` integer value, but is unspecified should the rounded value fall outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x;
  long long int y;

  x = 10.103;
  y = llroundl(x);
  printf("The nearest integer value to %Lf is %lld\n", x, y);

  x = 10.5;
```

```
  y = llroundl(x);
  printf("The nearest integer value to %Lf is %lld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.500000 is 11
```

5.11.109 log Function

Calculates the natural logarithm of a double-precision floating-point value.

Include

`<math.h>`

Prototype

`double log(double x);`

Argument

x any positive value for which to return the log

Return Value

Returns the natural (base-*e*) logarithm of `x`. If `x` is 0, infinity is returned. If `x` is a negative number, NaN is returned.

Remarks

If `x` < 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n",
         x, y);

  errno = 0;
  x = 0.0;
  y = log(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n",
         x, y);

  errno = 0;
  x = -2.0;
  y = log(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n",
         x, y);
}
```

Example Output

```
The natural logarithm of 2.000000 is 0.693147
The natural logarithm of 0.000000 is -inf
Error: domain error
The natural logarithm of -2.000000 is nan
```

5.11.110 logf Function

Calculates the natural logarithm of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float log(float x);`

Argument

x any positive value for which to return the log

Return Value

Returns the natural (base-*e*) logarithm of `x`. If `x` is 0, infinity is returned. If `x` is a negative number, NaN is returned.

Remarks

If `x` < 0, a domain error will occur and `errno` will be set to `EDOM`.

Example Output

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = logf(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n",
         x, y);

  errno = 0;
  x = 0.0F;
  y = logf(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n",
         x, y);

  errno = 0;
  x = -2.0F;
  y = logf(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %f is %f\n",
         x, y);
}
```

Example Output

```
The natural logarithm of 2.000000 is 0.693147
The natural logarithm of 0.000000 is -inf
```

```
Error: domain error
The natural logarithm of -2.000000 is nan
```

5.11.111 logl Function

Calculates the natural logarithm of a long double precision floating-point value.

Include

`<math.h>`

Prototype

```
long double logl(long double x);
```

Argument

x any positive value for which to return the log

Return Value

Returns the natural (base-*e*) logarithm of `x`. If `x` is 0, infinity is returned. If `x` is a negative number, NaN is returned .

Remarks

If `x` < 0, a domain error will occur and `errno` will be set to `EDOM`.

Example Output

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = 2.0F;
  y = logl(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %Lf is %Lf\n",
         x, y);

  errno = 0;
  x = 0.0F;
  y = logl(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %Lf is %Lf\n",
         x, y);

  errno = 0;
  x = -2.0F;
  y = logl(x);
  if (errno)
    perror("Error");
  printf("The natural logarithm of %Lf is %Lf\n",
         x, y);
}
```

Example Output

```
The natural logarithm of 2.000000 is 0.693147
The natural logarithm of 0.000000 is -inf
Error: domain error
The natural logarithm of -2.000000 is nan
```

5.11.112 log10 Function

Calculates the base-10 logarithm of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double log10(double x);`

Argument

x any double precision floating-point positive number

Return Value

Returns the base-10 logarithm of `x`. If `x` is 0, infinity is returned. If `x` is a negative number, NaN is returned .

Remarks

If `x` < 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = -2.0;
  y = log10 (x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);
}
```

Example Output

```
The base-10 logarithm of 2.000000 is 0.301030
The base-10 logarithm of 0.000000 is -inf
Error: domain error
The base-10 logarithm of -2.000000 is nan
```

5.11.113 log10f Function

Calculates the base-10 logarithm of a single-precision floating-point value.

Include

`<math.h>`

Prototype

```
float log10(float x);
```

Argument

x any single-precision floating-point positive number

Return Value

Returns the base-10 logarithm of `x`. If `x` is 0, infinity is returned. If `x` is a negative number, NaN is returned .

Remarks

If `x` < 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = 0.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = -2.0F;
  y = log10f(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);
}
```

Example Output

```
The base-10 logarithm of 2.000000 is 0.301030
Error: domain error
The base-10 logarithm of 0.000000 is -inf
Error: domain error
The base-10 logarithm of -2.000000 is nan
```

5.11.114 log10l Function

Calculates the base-10 logarithm of a long double-precision floating-point value.

Include

```
<math.h>
```

Prototype

```
long double log10(long double x);
```

Argument

x any long double-precision floating-point positive number

Return Value

Returns the base-10 logarithm of `x`. If `x` is 0, infinity is returned. If `x` is < 0, NaN is returned .

Remarks

If `x` < 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = 2.0;
  y = log10l(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = log10l(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = -2.0;
  y = log10l(x);
  if (errno)
    perror("Error");
  printf("The base-10 logarithm of %f is %f\n", x, y);
}
```

Example Output

```
The base-10 logarithm of 2.000000 is 0.301030
The base-10 logarithm of 0.000000 is -inf
Error: domain error
The base-10 logarithm of -2.000000 is nan
```

5.11.115 log1p Function

Calculates the natural logarithm of a double precision floating-point value plus 1.

Include

`<math.h>`

Prototype

```
double log1p(double x);
```

Argument

x any positive value for which to return the log

Return Value

Returns the natural (base-*e*) logarithm of `x+1` or NaN if a domain error occurs. The result of `log1p(x)` is generally expected to be more accurate than that of `log(x+1)` when the magnitude of `x` is small.

Remarks

If `x` is less than -1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log1p(x);
  if(errno)
    perror("Error");
  printf("The result of log1p(%f) is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = log1p(x);
  if(errno)
    perror("Error");
  printf("The result of log1p(%f) is %f\n", x, y);

  errno = 0;
  x = -2.0;
  y = log1p(x);
  if(errno)
    perror("Error");
  printf("The result of log1p(%f) is %f\n", x, y);
}
```

Example Output

```
The result of log1p(2.000000) is 1.098612
The result of log1p(0.000000) is 0.000000
Error: domain error
The result of log1p(-2.000000) is nan
```

5.11.116 log1pf Function

Calculates the natural logarithm of a single precision floating-point value plus 1.

Include

`<math.h>`

Prototype

`float log1pf(float x);`

Argument

x any positive value for which to return the log

Return Value

Returns the natural (base-*e*) logarithm of `x+1` or NaN if a domain error occurs. The result of `log1p(x)` is generally expected to be more accurate than that of `log(x+1)` when the magnitude of `x` is small.

Remarks

If `x` is less than -1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0;
  y = log1pf(x);
  if(errno)
    perror("Error");
  printf("The result of log1pf(%f) is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = log1pf(x);
  if(errno)
    perror("Error");
  printf("The result of log1pf(%f) is %f\n", x, y);

  errno = 0;
  x = -2.0;
  y = log1pf(x);
  if(errno)
    perror("Error");
  printf("The result of log1pf(%f) is %f\n", x, y);
}
```

Example Output

```
The result of log1pf(2.000000) is 1.098612
The result of log1pf(0.000000) is 0.000000
Error: domain error
The result of log1pf(-2.000000) is nan
```

5.11.117 log1pl Function

Calculates the natural logarithm of a long double precision floating-point value plus 1.

Include

`<math.h>`

Prototype

`long double log1pl(long double` **`x`**`);`

Argument

`x` any positive value for which to return the log

Return Value

Returns the natural (base-*e*) logarithm of `x+1` or NaN if a domain error occurs. The result of `log1p(x)` is generally expected to be more accurate than that of `log(x+1)` when the magnitude of `x` is small.

Remarks

If `x` is less than -1, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

```
#include <errno.h>

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log1pl(x);
  if(errno)
    perror("Error");
  printf("The result of log1pl(%f) is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = log1pl(x);
  if(errno)
    perror("Error");
  printf("The result of log1pl(%f) is %f\n", x, y);

  errno = 0;
  x = -2.0;
  y = log1pl(x);
  if(errno)
    perror("Error");
  printf("The result of log1pl(%f) is %f\n", x, y);
}
```

Example Output

```
The result of log1pl(2.000000) is 1.098612
The result of log1pl(0.000000) is 0.000000
Error: domain error
The result of log1pl(-2.000000) is nan
```

5.11.118 log2 Function

Calculates the base-2 logarithm of a double-precision floating-point value.

Include

`<math.h>`

Prototype

`double log2(double x);`

Argument

x any double-precision floating-point positive number

Return Value

Returns the base-2 logarithm of `x`. If `x` is +/-0, -infinity is returned. If `x` is a negative number, NaN is returned .

Remarks

If `x` is less than 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
```

```
  x = 2.0;
  y = log2(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = log2(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = -2.0;
  y = log2(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %f is %f\n", x, y);
}
```

Example Output

```
The base-2 logarithm of 2.000000 is 1.000000
The base-2 logarithm of 0.000000 is -inf
Error: domain error
The base-2 logarithm of -2.000000 is nan
```

5.11.119 log2f Function

Calculates the base-2 logarithm of a single-precision floating-point value.

Include

`<math.h>`

Prototype

`float log2f(float` **`x`**`);`

Argument

`x` any single-precision floating-point positive number

Return Value

Returns the base-2 logarithm of `x`. If `x` is +/-0, -infinity is returned. If `x` is a negative number, NaN is returned .

Remarks

If `x` is less than 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 2.0;
  y = log2f(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %f is %f\n", x, y);

  errno = 0;
```

```
  x = 0.0;
  y = log2f(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %f is %f\n", x, y);

  errno = 0;
  x = -2.0;
  y = log2f(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %f is %f\n", x, y);
}
```

Example Output

```
The base-2 logarithm of 2.000000 is 1.000000
The base-2 logarithm of 0.000000 is -inf
Error: domain error
The base-2 logarithm of -2.000000 is nan
```

5.11.120 log2l Function

Calculates the base-2 logarithm of a long double-precision floating-point value.

Include

`<math.h>`

Prototype

`long double log2l(long double x);`

Argument

x any long double-precision floating-point positive number

Return Value

Returns the base-2 logarithm of `x`. If `x` is +/-0, -infinity is returned. If `x` is < 0, NaN is returned .

Remarks

If `x` is less than 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 2.0;
  y = log2l(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 0.0;
  y = log2l(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %Lf is %Lf\n", x, y);

  errno = 0;
```

```
  x = -2.0;
  y = log2l(x);
  if(errno)
    perror("Error");
  printf("The base-2 logarithm of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The base-2 logarithm of 2.000000 is 1.000000
The base-2 logarithm of 0.000000 is -inf
Error: domain error
The base-2 logarithm of -2.000000 is nan
```

5.11.121 logb Function

Calculates the signed exponent of a double-precision floating-point value.

Include

`<math.h>`

Prototype

`double logb(double` **`x`**`);`

Argument

x any positive value for which to return the exponent

Return Value

Returns the exponent of `x` as a signed floating-point value. If `x` is 0, ∞ is returned.

Remarks

The argument is treated as being normalized if it is a subnormal floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 13.45;
  y = logb(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %f\n",
         x, y);

  errno = 0;
  x = 0.0;
  y = logb(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %f\n",
         x, y);

  errno = 0;
  x = -2.0;
  y = logb(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %f\n",
```

```
          x, y);
}
```

Example Output

```
The exponent of 13.450000 is 3.000000
The exponent of 0.000000 is -inf
The exponent of -2.000000 is 1.000000
```

5.11.122 logbf Function

Calculates the signed exponent of a single-precision floating-point value.

Include

`<math.h>`

Prototype

`float logbf(float x);`

Argument

x any positive value for which to return the exponent

Return Value

Returns the exponent of `x` as a signed floating-point value. If `x` is 0, ∞ is returned.

Remarks

The argument is treated as being normalized if it is a subnormal floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = 13.45;
  y = logbl(x);
  if (errno)
    perror("Error");
  printf("The exponent of %Lf is %Lf\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = logbl(x);
  if (errno)
    perror("Error");
  printf("The exponent of %Lf is %Lf\n",
          x, y);

  errno = 0;
  x = -2.0;
  y = logbl(x);
  if (errno)
    perror("Error");
  printf("The exponent of %Lf is %Lf\n",
          x, y);
}
```

Example Output

```
The exponent of 13.450000 is 3.000000
The exponent of 0.000000 is -inf
The exponent of -2.000000 is 1.000000
```

5.11.123 logbl Function

Calculates the signed exponent of a long double-precision floating-point value.

Include

`<math.h>`

Prototype

`long double logbl(long double` **x**`);`

Argument

x any positive value for which to return the exponent

Return Value

Returns the exponent of `x` as a signed floating-point value. If `x` is 0, ∞ is returned.

Remarks

The argument is treated as being normalized if it is a subnormal floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = 13.45;
  y = logbf(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %f\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = logbf(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %f\n",
          x, y);

  errno = 0;
  x = -2.0;
  y = logbf(x);
  if (errno)
    perror("Error");
  printf("The exponent of %f is %f\n",
          x, y);
}
```

Example Output

```
The exponent of 13.450000 is 3.000000
The exponent of 0.000000 is -inf
The exponent of -2.000000 is 1.000000
```

5.11.124 lrint Function

Returns the double precision floating-point argument rounded to the nearest integer value.

Include

`<math.h>`

Prototype

`long int lrint(double x);`

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value using the current rounding direction. The rounded integer is returned as a `long` integer value.

Remarks

The value returned is unspecified if the rounded value is outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;
  long int y;

  x = 10.103;
  y = lrint(x);
  printf("The nearest integer value to %f is %ld\n", x, y);

  x = 10.51;
  y = lrint(x);
  printf("The nearest integer value to %f is %ld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.510000 is 11
```

5.11.125 lrintf Function

Returns the single precision floating-point argument rounded to the nearest integer value.

Include

`<math.h>`

Prototype

`long int lrintf(float x);`

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value using the current rounding direction. The rounded integer is returned as a `long` integer value.

Remarks

The value returned is unspecified if the rounded value is outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x;
  long int y;

  x = 10.103;
  y = lrintf(x);
  printf("The nearest integer value to %f is %ld\n", x, y);

  x = 10.51;
  y = lrintf(x);
  printf("The nearest integer value to %f is %ld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.510000 is 11
```

5.11.126 lrintl Function

Returns the long double precision floating-point argument rounded to the nearest integer value.

Include

`<math.h>`

Prototype

`long int lrintl(long double x);`

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value using the current rounding direction. The rounded integer is returned as a `long` integer value.

Remarks

The value returned is unspecified if the rounded value is outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

```
int main(void)
{
  double x;
  long int y;

  x = 10.103;
  y = lrintl(x);
  printf("The nearest integer value to %Lf is %ld\n", x, y);

  x = 10.51;
  y = lrintl(x);
  printf("The nearest integer value to %Lf is %ld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.510000 is 11
```

5.11.127 lround Function

Returns the double precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

`long int lround(double` **`x`**`);`

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded integer is returned as a `long` integer value.

Remarks

The value returned is unspecified should the rounded value fall outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;
  long int y;

  x = 10.103;
  y = lround(x);
  printf("The nearest integer value to %f is %ld\n", x, y);

  x = 10.5;
  y = lround(x);
  printf("The nearest integer value to %f is %ld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.500000 is 11
```

5.11.128 lroundf Function

Returns the single precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

```
long int lroundf(float x);
```

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded integer is returned as a `long` integer value.

Remarks

The value returned is unspecified should the rounded value fall outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x;
  long int y;

  x = 10.103;
  y = lroundf(x);
  printf("The nearest integer value to %f is %ld\n", x, y);

  x = 10.5;
  y = lroundf(x);
  printf("The nearest integer value to %f is %ld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.500000 is 11
```

5.11.129 lroundl Function

Returns the long double precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

```
long int lroundl(long double x);
```

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded integer is returned as a `long` integer value.

Remarks

The value returned is unspecified should the rounded value fall outside the range of the return type.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x;
  long int y;

  x = 10.103;
  y = lroundl(x);
  printf("The nearest integer value to %Lf is %ld\n", x, y);

  x = 10.5;
  y = lroundl(x);
  printf("The nearest integer value to %Lf is %ld\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10
The nearest integer value to 10.500000 is 11
```

5.11.130 modf Function

Splits a double precision floating-point value into fractional and integer parts.

Include

`<math.h>`

Prototype

`double modf(double` **`x`**`, double *` **`pint`**`);`

Arguments

`x`	double precision floating-point value
`pint`	pointer to where the integer part should be stored

Return Value

Returns the signed fractional part and `pint` points to the integer part.

Remarks

The absolute value of the fractional part is in the range of 0 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, n;

  x = 0.707;
  y = modf(x, &n);
```

```
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);

  x = -15.2121;
  y = modf(x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);
}
```

Example Output

```
For 0.707000 the fraction is 0.707000
   and the integer is 0

For -15.212100 the fraction is -0.212100
   and the integer is -15
```

5.11.131 modff Function

Splits a single precision floating-point value into fractional and integer parts.

Include

`<math.h>`

Prototype

`float modff(float` **`x`**`, float * ` **`pint`**`);`

Arguments

`x`	single precision floating-point value
`pint`	pointer to where the integer part should be stored

Return Value

Returns the signed fractional part and `pint` points to the integer part.

Remarks

The absolute value of the fractional part is in the range of 0 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y, n;

  x = 0.707F;
  y = modff(x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);

  x = -15.2121F;
  y = modff(x, &n);
  printf("For %f the fraction is %f\n ", x, y);
  printf("  and the integer is %0.f\n\n", n);
}
```

Example Output

```
For 0.707000 the fraction is 0.707000
   and the integer is 0
```

```
For -15.212100 the fraction is -0.212100
   and the integer is -15
```

5.11.132 modfl Function

Splits a long double precision floating-point value into fractional and integer parts.

Include

`<math.h>`

Prototype

```
long double modfl(long double x, long double * pint);
```

Arguments

x	long double precision floating-point value
pint	pointer to where the integer part should be stored

Return Value

Returns the signed fractional part and `pint` points to the integer part.

Remarks

The absolute value of the fractional part is in the range of 0 (inclusive) to 1 (exclusive). No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y, n;

  x = 0.707;
  y = modfl(x, &n);
  printf("For %Lf the fraction is %Lf\n ", x, y);
  printf("  and the integer is %0.Lf\n\n", n);

  x = -15.2121;
  y = modfl(x, &n);
  printf("For %Lf the fraction is %Lf\n ", x, y);
  printf("  and the integer is %0.Lf\n\n", n);
}
```

Example Output

```
For 0.707000 the fraction is 0.707000
   and the integer is 0

For -15.212100 the fraction is -0.212100
   and the integer is -15
```

5.11.133 nan Function

Returns a quiet NaN double precision floating-point value.

Include

`<math.h>`

Prototype

```
double nan(const char * tagp);
```

Arguments

tagp an optional, implementation-defined string which might be used to represent extra information in the NaN's significand

Return Value

The call `nan("n-char-sequence")` is equivalent to `strtod("NAN(n-char- sequence)", (char**) NULL);` the call `nan("")` is equivalent to `strtod("NAN()", (char**) NULL)`. When `tagp` does not point to an n-char sequence or an empty string, the equivalent call to `strtod` would have a first argument of `"NAN"`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;

  z = nan(NULL);
  printf("Here is our not-a-number: %f\n", z);
}
```

Example Output

```
Here is our not-a-number: nan
```

5.11.134 nanf Function

Returns a quiet NaN single precision floating-point value.

Include

`<math.h>`

Prototype

`float nanf(const char * tagp);`

Arguments

tagp an optional, implementation-defined string which might be used to represent extra information in the NaN's significand

Return Value

The call `nan("n-char-sequence")` is equivalent to `strtof("NAN(n-char- sequence)", (char**) NULL);` the call `nan("")` is equivalent to `strtof("NAN()", (char**) NULL)`. When `tagp` does not point to an n-char sequence or an empty string, the equivalent call to `strtof` would have a first argument of `"NAN"`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x;

  z = nanf(NULL);
  printf("Here is our not-a-number: %f\n", z);
}
```

Example Output

```
Here is our not-a-number: nan
```

5.11.135 nanl Function

Returns a quiet NaN long double precision floating-point value.

Include

`<math.h>`

Prototype

```
long double nanl(const char * tagp);
```

Arguments

tagp an optional, implementation-defined string which might be used to represent extra information in the NaN's significand

Return Value

The call `nan("n-char-sequence")` is equivalent to `strtold("NAN(n-char- sequence)", (char**) NULL);` the call `nan("")` is equivalent to `strtold("NAN()", (char**) NULL)`. When `tagp` does not point to an n-char sequence or an empty string, the equivalent call to `strtold` would have a first argument of `"NAN"`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x;

  z = nanl(NULL);
  printf("Here is our not-a-number: %Lf\n", z);
}
```

Example Output

```
Here is our not-a-number: nan
```

5.11.136 nearbyint Function

Returns the double precision floating-point argument rounded to an integer value but returned in the argument's type.

Include

`<math.h>`

Prototype

```
double nearbyint(double x);
```

Argument

x the value to round

Return Value

Returns the value of `x` rounded to an integer value using the current rounding direction and without generating an exception. The rounded integer is returned as a floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 10.103;
  y = nearbyint(x);
  printf("The nearest integer value to %f is %f\n", x, y);

  x = 10.51;
  y = nearbyint(x);
  printf("The nearest integer value to %f is %f\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.510000 is 11.000000
```

5.11.137 nearbyintf Function

Returns the double precision floating-point argument rounded to an integer value but returned in the argument's type.

Include

`<math.h>`

Prototype

`float nearbyintf(float` **`x`**`);`

Argument

`x`	the value to round

Return Value

Returns the value of `x` rounded to an integer value using the current rounding direction and without generating an exception. The rounded integer is returned as a floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 10.103;
  y = nearbyintf(x);
  printf("The nearest integer value to %f is %f\n", x, y);

  x = 10.51;
  y = nearbyintf(x);
  printf("The nearest integer value to %f is %f\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.510000 is 11.000000
```

5.11.138 nearbyintl Function

Returns the double precision floating-point argument rounded to an integer value but returned in the argument's type.

Include

`<math.h>`

Prototype

```
long double nearbyintl(long double x);
```

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to an integer value using the current rounding direction and without generating an exception. The rounded integer is returned as a floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 10.103;
  y = nearbyintl(x);
  printf("The nearest integer value to %Lf is %Lf\n", x, y);

  x = 10.51;
  y = nearbyintl(x);
  printf("The nearest integer value to %Lf is %Lf\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.510000 is 11.000000
```

5.11.139 nextafter Function

Determines the next value after `x` in the direction of `y` that can be represented by a double precision floating-point value.

Include

`<math.h>`

Prototype

```
double nextafter(double x, double y);
```

Arguments

x	the original value
y	the target value

Return Value

Determines the next value after `x` in the direction of `y` that can be represented by the type of the function. The value of `y` is returned if `x` equals `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y,z;

  x = 150.0;
  y = 300.0;
  z = nextafter(x, y);
  printf("The next representable value after %f in the direction of %f is %.10f\n", x, y, z);

  x = 150.0;
  y = -300.0;
  z = nextafter(x, y);
  printf("The next representable value after %f in the direction of %f is %.10f\n", x, y, z);

}
```

Example Output

```
The next representable value after 150.00000 in the direction of 300.00000 is 150.0000152588
The next representable value after 150.00000 in the direction of -300.00000 is 149.9999847412
```

5.11.140 nextafterf Function

Determines the next value after `x` in the direction of `y` that can be represented by a single precision floating-point value.

Include

`<math.h>`

Prototype

`float nextafterf(float` **`x`**`, float` **`y`**`);`

Arguments

x	the original value
y	the target value

Return Value

Determines the next value after `x` in the direction of `y` that can be represented by the type of the function. The value of `y` is returned if `x` equals `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x,y,z;

  x = 150.0;
  y = 300.0;
  z = nextafterf(x, y);
  printf("The next representable value after %f in the direction of %f is %.10f\n", x, y, z);

  x = 150.0;
  y = -300.0;
  z = nextafterf(x, y);
  printf("The next representable value after %f in the direction of %f is %.10f\n", x, y, z);
```

```
}
```

Example Output

```
The next representable value after 150.0000000000 in the direction of 300.0000000000 is
150.0000152588
The next representable value after 150.0000000000 in the direction of -300.0000000000 is
149.9999847412
```

5.11.141 nextafterl Function

Determines the next value after `x` in the direction of `y` that can be represented by a long double precision floating-point value.

Include

`<math.h>`

Prototype

`long double nextafterl(long double` **`x`**`, long double` **`y`**`);`

Arguments

x	the original value
y	the target value

Return Value

Determines the next value after `x` in the direction of `y` that can be represented by the type of the function. The value of `y` is returned if `x` equals `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x,y,z;

  x = 150.0;
  y = 300.0;
  z = nextafterl(x, y);
  printf("The next representable value after %Lf in the direction of %Lf is %.20Lf\n", x, y,
z);

  x = 150.0;
  y = -300.0;
  z = nextafterl(x, y);
  printf("The next representable value after %Lf in the direction of %Lf is %.20Lf\n", x, y,
z);

}
```

Example Output

```
The next representable value after 150.000000 in the direction of 300.000000 is
150.00000000000002842171
The next representable value after 150.000000 in the direction of -300.000000 is
149.99999999999997157829
```

5.11.142 nexttoward Function

Determines the next value after `x` in the direction of `y` that can be represented by a double precision floating-point value.

Include

`<math.h>`

Prototype

```
double nexttoward(double x, long double y);
```

Arguments

x the original value

y the target value as a `long double`

Return Value

Determines the next value after `x` in the direction of `y` that can be represented by the `double` type. The value of `y` converted to the type of the function is returned if `x` equals `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, z;
  long double y;

  x = 150.0;
  y = 300.0;
  z = nexttoward(x, y);
  printf("The next representable value after %f in the direction of %Lf is %.10f\n", x, y, z);

  x = 150.0;
  y = -300.0;
  z = nexttoward(x, y);
  printf("The next representable value after %f in the direction of %Lf is %.10f\n", x, y, z);

}
```

Example Output

```
The next representable value after 150.000000 in the direction of 300.000000 is 150.0000000000
The next representable value after 150.000000 in the direction of -300.000000 is
150.0000000000
```

5.11.143 nexttowardf Function

Determines the next value after `x` in the direction of `y` that can be represented by a single precision floating-point value.

Include

`<math.h>`

Prototype

```
float nexttowardf(float x, long double y);
```

Arguments

x the original value

y the target value as a `long double`

Return Value

Determines the next value after `x` in the direction of `y` that can be represented by the `float` type. The value of `y` converted to the type of the function is returned if `x` equals `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, z;
  long double y;

  x = 150.0;
  y = 300.0;
  z = nexttowardf(x, y);
  printf("The next representable value after %f in the direction of %Lf is %.10f\n", x, y, z);

  x = 150.0;
  y = -300.0;
  z = nexttowardf(x, y);
  printf("The next representable value after %f in the direction of %Lf is %.10f\n", x, y, z);

}
```

Example Output

```
The next representable value after 150.000000 in the direction of 300.000000 is 150.0000000000
The next representable value after 150.000000 in the direction of -300.000000 is
150.0000000000
```

5.11.144 nexttowardl Function

Determines the next value after `x` in the direction of `y` that can be represented by a long double precision floating-point value.

Include

`<math.h>`

Prototype

```
long double nexttowardl(long double x, long double y);
```

Arguments

x the original value

y the target value as a `long double`

Return Value

Determines the next value after `x` in the direction of `y` that can be represented by the `long double` type. The value of `y` converted to the type of the function is returned if `x` equals `y`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
```

```
{
  long double x, z;
  long double y;

  x = 150.0;
  y = 300.0;
  z = nexttowardl(x, y);
  printf("The next representable value after %Lf in the direction of %Lf is %.10Lf\n", x, y,
z);

  x = 150.0;
  y = -300.0;
  z = nexttowardl(x, y);
  printf("The next representable value after %Lf in the direction of %Lf is %.10Lf\n", x, y,
z);

}
```

Example Output

```
The next representable value after 150.000000 in the direction of 300.000000 is 150.0000000000
The next representable value after 150.000000 in the direction of -300.000000 is
150.0000000000
```

5.11.145 pow Function

Calculates `x` raised to the power `y`.

Include

`<math.h>`

Prototype

`double pow(double` **`x`**`, double` **`y`**`);`

Arguments

x	the base
y	the exponent

Return Value

Returns `x` raised to the power `y` (x^y).

Remarks

If `x` is finite and negative and `y` is finite and not an integer value, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y,z;

  errno = 0;
  x = -2.0;
  y = 3.0;
  z = pow(x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n ", x, y, z);

  errno = 0;
```

```
  x = 3.0;
  y = -0.5;
  z = pow(x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n ", x, y, z);

  errno = 0;
  x = 4.0;
  y = 0.0;
  z = pow(x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n ", x, y, z);

  errno = 0;
  x = 0.0;
  y = -3.0;
  z = pow(x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n ", x, y, z);
}
```

Example Output

```
-2.000000 raised to 3.000000 is -8.000000
 3.000000 raised to -0.500000 is 0.577350
 4.000000 raised to 0.000000 is 1.000000
 Error: domain error
 0.000000 raised to -3.000000 is inf
```

5.11.146 powf Function

Calculates x raised to the power y.

Include

`<math.h>`

Prototype

`float powf(float` **`x`**`, float` **`y`**`);`

Arguments

x	the base
y	the exponent

Return Value

Returns `x` raised to the power `y` (`x^y`).

Remarks

If `x` is finite and negative and `y` is finite and not an integer value, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x,y,z;

  errno = 0;
```

```
  x = -2.0F;
  y = 3.0F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n ", x, y, z);

  errno = 0;
  x = 3.0F;
  y = -0.5F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n ", x, y, z);

  errno = 0;
  x = 0.0F;
  y = -3.0F;
  z = powf (x, y);
  if (errno)
    perror("Error");
  printf("%f raised to %f is %f\n ", x, y, z);
}
```

Example Output

```
-2.000000 raised to 3.000000 is -8.000000
 3.000000 raised to -0.500000 is 0.577350
 Error: domain error
 0.000000 raised to -3.000000 is inf
```

5.11.147 powl Function

Calculates x raised to the power y.

Include

`<math.h>`

Prototype

`long double powl(long double` **`x`**`, long double` **`y`**`);`

Arguments

x	the base
y	the exponent

Return Value

Returns `x` raised to the power `y` (x^y).

Remarks

If `x` is finite and negative and `y` is finite and not an integer value, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x,y,z;

  errno = 0;
  x = -2.0;
```

```
  y = 3.0;
  z = powl(x, y);
  if (errno)
    perror("Error");
  printf("%Lf raised to %Lf is %Lf\n ", x, y, z);

  errno = 0;
  x = 3.0;
  y = -0.5;
  z = powl(x, y);
  if (errno)
    perror("Error");
  printf("%Lf raised to %Lf is %Lf\n ", x, y, z);

  errno = 0;
  x = 4.0;
  y = 0.0;
  z = powl(x, y);
  if (errno)
    perror("Error");
  printf("%Lf raised to %Lf is %Lf\n ", x, y, z);

  errno = 0;
  x = 0.0;
  y = -3.0;
  z = pow (x, y);
  if (errno)
    perror("Error");
  printf("%Lf raised to %Lf is %Lf\n ", x, y, z);
}
```

Example Output

```
-2.000000 raised to 3.000000 is -8.000000
 3.000000 raised to -0.500000 is 0.577350
 4.000000 raised to 0.000000 is 1.000000
 Error: domain error
 0.000000 raised to -3.000000 is inf
```

5.11.148 remainder Function

Calculates `x` REM `y` as a double precision value.

Include

`<math.h>`

Prototype

`double remainder(double` **`x`**`, double` **`y`**`);`

Arguments

`x` a double precision floating-point value

`y` a double precision floating-point value

Return Value

Returns the remainder `x` REM `y`, being `x – n`y, where *n* is the nearest integer to the exact value of `x/y` when `y` is not 0. The rounding mode is ignored. If the remainder is 0, its sign shall be the same as that of `x`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y,z;
```

```
  x = 7.0;
  y = 3.0;
  z = remainder(x, y);
  printf("%f REM %f is %f\n", x, y, z);
}
```

Example Output

```
7.000000 REM 3.000000 is 1.000000
```

5.11.149 remainderf Function

Calculates x REM y as a single precision value.

Include

<math.h>

Prototype

```
float remainderf(float x, float y);
```

Arguments

x	a single precision floating-point value
y	a single precision floating-point value

Return Value

Returns the remainder x REM y, being $x - ny$, where n is the nearest integer to the exact value of x/y when y is not 0. The rounding mode is ignored. If the remainder is 0, its sign shall be the same as that of x.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x,y,z;

  x = 7.0;
  y = 3.0;
  z = remainderf(x, y);
  printf("%f REM %f is %f\n", x, y, z);
}
```

Example Output

```
7.000000 REM 3.000000 is 1.000000
```

5.11.150 remainderl Function

Calculates x REM y as a long double precision value.

Include

<math.h>

Prototype

```
long double remainderl(long double x, long double y);
```

Arguments

x a double precision floating-point value

y a double precision floating-point value

Return Value

Returns the remainder x REM y, being x – *n*y, where *n* is the nearest integer to the exact value of x/y when y is not 0. The rounding mode is ignored. If the remainder is 0, its sign shall be the same as that of x.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  long double x,y,z;

  x = 7.0;
  y = 3.0;
  z = remainderl(x, y);
  printf("%Lf REM %Lf is %Lf\n", x, y, z);
}
```

Example Output

```
7.000000 REM 3.000000 is 1.000000
```

5.11.151 remquo Function

Calculates x REM y as a double precision floating-point value.

Include

<math.h>

Prototype

```
double remquo(double x, double y, int * quo);
```

Arguments

x a double precision floating-point value

y a double precision floating-point value

quo a pointer to an int object that can hold the quotient

Return Value

Returns the same remainder as remainder function, being x – *n*y, where *n* is the nearest integer to the exact value of x/y. In the object pointed to by quo is stored a quotient with the same sign as x/y, and whose magnitude is congruent modulo 2^m to the magnitude of the integral quotient of x/y, where *m* is an implementation-defined integer greater than or equal to 3. The rounding mode is ignored.

Remarks

If the remainder is 0, its sign shall be the same as that of x.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

```
int main(void)
{
  double x, y, z;
  int q;

  x = 7.0;
  y = 3.0;
  z = remquo(x, y, &q);
  printf("%f REM %f is %f with quotient %d\n", x, y, z, *q);
}
```

Example Output

```
7.000000 REM 3.000000 is 1.000000 with quotient 2
```

5.11.152 remquof Function

Calculates x REM y as a single precision floating-point value.

Include

<math.h>

Prototype

```
float requof(float x, float y, int * quo);
```

Arguments

x	a single precision floating-point value
y	a single precision floating-point value
quo	a pointer to an int object that can hold the quotient

Return Value

Returns the same remainder as remainder function, being $x - ny$, where n is the nearest integer to the exact value of x/y. In the object pointed to by quo is stored a quotient with the same sign as x/y, and whose magnitude is congruent modulo 2^m to the magnitude of the integral quotient of x/y, where *m* is an implementation-defined integer greater than or equal to 3. The rounding mode is ignored.

Remarks

If the remainder is 0, its sign shall be the same as that of x.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y, z;
  int q;

  x = 7.0;
  y = 3.0;
  z = remquof(x, y, &q);
  printf("%f REM %f is %f with quotient %d\n", x, y, z, *q);
}
```

Example Output

```
7.000000 REM 3.000000 is 1.000000 with quotient 2
```

5.11.153 remquol Function

Calculates x REM y as a long double precision floating-point value.

Include

<math.h>

Prototype

```
long double remainder(long double x, long double y, int * quo);
```

Arguments

x	a double precision floating-point value
y	a double precision floating-point value
quo	a pointer to an int object that can hold the quotient

Return Value

Returns the same remainder as remainder function, being $x - ny$, where n is the nearest integer to the exact value of x/y. In the object pointed to by quo is stored a quotient with the same sign as x/y, and whose magnitude is congruent modulo 2^m to the magnitude of the integral quotient of x/y, where *m* is an implementation-defined integer greater than or equal to 3. The rounding mode is ignored.

Remarks

If the remainder is 0, its sign shall be the same as that of x.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  long double x, y, z;
  int q;

  x = 7.0;
  y = 3.0;
  z = remquol(x, y, &q);
  printf("%Lf REM %Lf is %Lf with quotient %d\n", x, y, z, *q);
}
```

Example Output

```
7.000000 REM 3.000000 is 1.000000 with quotient 2
```

5.11.154 rint Function

Returns the double precision floating-point argument rounded to an integer value.

Include

<math.h>

Prototype

```
double rint(double x);
```

Argument

x	the value to round

Return Value

Returns the value of x rounded to an integer value using the current rounding direction, raising the inexact floating-point exception should the result not have the same value as the argument. The rounded integer is returned as a double precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 10.103;
  y = rint(x);
  printf("The nearest integer value to %f is %f\n", x, y);

  x = 10.51;
  y = rint(x);
  printf("The nearest integer value to %f is %f\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.510000 is 11.000000
```

5.11.155 rintf Function

Returns the single precision floating-point argument rounded to an integer value.

Include

<math.h>

Prototype

```
float rintf(float x);
```

Argument

x	the value to round

Return Value

Returns the value of x rounded to an integer value using the current rounding direction, raising the inexact floating-point exception should the result not have the same value as the argument. The rounded integer is returned as a single precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 10.103;
  y = rintf(x);
  printf("The nearest integer value to %f is %f\n", x, y);

  x = 10.51;
  y = rintf(x);
```

```
  printf("The nearest integer value to %f is %f\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.510000 is 11.000000
```

5.11.156 rintl Function

Returns the long double precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

`long double rintl(long double` **`x`**`);`

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to an integer value using the current rounding direction, raising the inexact floating-point exception should the result not have the same value as the argument. The rounded integer is returned as a long double precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = 10.103;
  y = rintl(x);
  printf("The nearest integer value to %Lf is %Lf\n", x, y);

  x = 10.51;
  y = rintl(x);
  printf("The nearest integer value to %Lf is %Lf\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.510000 is 11.000000
```

5.11.157 round Function

Returns the double precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

`double round(double` **`x`**`);`

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded integer is returned as a double precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 10.103;
  y = round(x);
  printf("The nearest integer value to %f is %f\n", x, y);

  x = 10.5;
  y = round(x);
  printf("The nearest integer value to %f is %f\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.500000 is 11.000000
```

5.11.158 roundf Function

Returns the double precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

`float roundf(float x);`

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded integer is returned as a single precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 10.103;
  y = roundf(x);
  printf("The nearest integer value to %f is %f\n", x, y);

  x = 10.5;
```

```
  y = roundf(x);
  printf("The nearest integer value to %f is %f\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.500000 is 11.000000
```

5.11.159 roundl Function

Returns the double precision floating-point argument rounded to an integer value.

Include

`<math.h>`

Prototype

```
long double roundl(long double x);
```

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value, always rounding midway cases away from zero. The rounded integer is returned as a long double precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = 10.103;
  y = roundl(x);
  printf("The nearest integer value to %Lf is %Lf\n", x, y);

  x = 10.5;
  y = roundl(x);
  printf("The nearest integer value to %Lf is %Lf\n", x, y);
}
```

Example Output

```
The nearest integer value to 10.103000 is 10.000000
The nearest integer value to 10.500000 is 11.000000
```

5.11.160 scalbln Function

Calculates the signed exponent of a double precision floating-point value.

Include

`<math.h>`

Prototype

```
double scalbln(double x, long int n);
```

Arguments

x multiplier

n the power to which FLT_RADIX is raised

Return Value

Efficiently calculates and returns the value of x times FLT_RADIX^n.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  double x, y;
  long int power;

  errno = 0;
  x = 13.45;
  power = 8;
  y = scalbln(x, power);
  if (errno)
    perror("Error");
  printf("FLT_RADIX raised to the power %ld, times %f is %f\n", power, x, y);
}
```

Example Output

```
FLT_RADIX raised to the power 8, times 13.450000 is 3443.200000
```

5.11.161 scalblnf Function

Calculates the signed exponent of a single precision floating-point value.

Include

<math.h>

Prototype

```
float scalblnf(float x, long int n);
```

Arguments

x multiplier

n the power to which FLT_RADIX is raised

Return Value

Efficiently calculates and returns the value of x times FLT_RADIX^n.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y;
  long int power;

  errno = 0;
  x = 13.45;
  power = 8;
```

```
  y = scalblnf(x, power);
  if (errno)
    perror("Error");
  printf("FLT_RADIX raised to the power %ld, times %f is %f\n", power, x, y);
}
```

Example Output

```
FLT_RADIX raised to the power 8, times 13.450000 is 3443.200000
```

5.11.162 scalblnl Function

Calculates the signed exponent of a long double precision floating-point value.

Include

<math.h>

Prototype

```
long double scalblnl(long double x, long int n);
```

Arguments

x multiplier

n the power to which FLT_RADIX is raised

Return Value

Efficiently calculates and returns the value of x times FLT_RADIX^n.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  long double x, y;
  long int power;

  errno = 0;
  x = 13.45;
  power = 8;
  y = scalblnl(x, power);
  if (errno)
    perror("Error");
  printf("FLT_RADIX raised to the power %ld, times %Lf is %Lf\n", power, x, y);
}
```

Example Output

```
FLT_RADIX raised to the power 8, times 13.450000 is 3443.200000
```

5.11.163 scalbn Function

Calculates the signed exponent of a double precision floating-point value.

Include

<math.h>

Prototype

```
double scalbn(double x, int n);
```

Arguments

x multiplier

n the power to which FLT_RADIX is raised

Return Value

Efficiently calculates and returns the value of x times FLT_RADIX^n.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  double x, y;
  int power;

  errno = 0;
  x = 13.45;
  power = 8;
  y = scalbn(x, power);
  if (errno)
    perror("Error");
  printf("FLT_RADIX raised to the power %d, times %f is %f\n", power, x, y);
}
```

Example Output

```
FLT_RADIX raised to the power 8, times 13.450000 is 3443.200000
```

5.11.164 scalbnf Function

Calculates the signed exponent of a single precision floating-point value.

Include

<math.h>

Prototype

```
float scalbnf(float x, int n);
```

Arguments

x multiplier

n the power to which FLT_RADIX is raised

Return Value

Efficiently calculates and returns the value of x times FLT_RADIX^n.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  float x, y;
  int power;

  errno = 0;
```

```
  x = 13.45;
  power = 8;
  y = scalbnf(x, power);
  if (errno)
    perror("Error");
  printf("FLT_RADIX raised to the power %d, times %f is %f\n", power, x, y);
}
```

Example Output

```
FLT_RADIX raised to the power 8, times 13.450000 is 3443.200000
```

5.11.165 scalbnl Function

Calculates the signed exponent of a long double precision floating-point value.

Include

`<math.h>`

Prototype

`long double scalbnl(long double` **`x`**`, int` **`n`**`);`

Argument

`x` multiplier

`n` the power to which `FLT_RADIX` is raised

Return Value

Efficiently calculates and returns the value of `x` times $\texttt{FLT\_RADIX}^{n}$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;
  int power;

  errno = 0;
  x = 13.45;
  power = 8;
  y = scalbnl(x, power);
  if (errno)
    perror("Error");
  printf("FLT_RADIX raised to the power %d, times %Lf is %Lf\n", power, x, y);
}
```

Example Output

```
FLT_RADIX raised to the power 8, times 13.450000 is 3443.200000
```

5.11.166 signbit Macro

Returns true if its argument is negative.

Include

`<math.h>`

Prototype

`int signbit(`*`floating-point`* **`x`**`);`

Argument

x any floating-point number

Return Value

Returns true if its argument is negative; false otherwise.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y, z;
  int s;

  x = -5.0;
  y = 3.0;
  z = x / y;
  s = signbit(z);
  printf("The result of the division of %f by %f is %s\n", x, y, s ? "negative" : "positive");
}
```

Example Output

```
The result of the division of -5.000000 by 3.000000 is negative
```

5.11.167 sin Function

Calculates the trigonometric sine function of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double sin (double b);`

Argument

x value for which to return the sine

Return Value

Returns the sine of x specified in radians in the range $[-1, 1]$. NaN is returned if x is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = sin (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n", x, y);
```

```
  errno = 0;
  x = 0.0;
  y = sin (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n", x, y);
}
```

Example Output

```
The sine of -1.000000 is -0.841471
The sine of 0.000000 is 0.000000
```

5.11.168 sinf Function

Calculates the trigonometric sine function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float sin (float` **`x`**`);`

Argument

`x` value for which to return the sine

Return Value

Returns the sine of `x` specified in radians in the range $[-1, 1]$. NaN is returned if `x` is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = sinf (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0F;
  y = sinf (x);
  if (errno)
    perror("Error");
  printf("The sine of %f is %f\n", x, y);
}
```

Example Output

```
The sine of -1.000000 is -0.841471
The sine of 0.000000 is 0.000000
```

5.11.169 sinl Function

Calculates the trigonometric sine function of a long double precision floating-point value.

Include

<math.h>

Prototype

```
long double sinl(long double x);
```

Argument

x value for which to return the sine

Return Value

Returns the sine of x specified in radians in the range $[-1, 1]$. NaN is returned if x is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = -1.0;
  y = sinl(x);
  if (errno)
    perror("Error");
  printf("The sine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 0.0;
  y = sinl(x);
  if (errno)
    perror("Error");
  printf("The sine of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The sine of -1.000000 is -0.841471
The sine of 0.000000 is 0.000000
```

5.11.170 sinh Function

Calculates the hyperbolic sine function of a double precision floating-point value.

Include

<math.h>

Prototype

```
double sinh (double x);
```

Argument

x value for which to return the hyperbolic sine

Return Value

Returns the hyperbolic sine of x.

Remarks

If x is too large, a range error will occur and errno will be set to ERANGE.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.5;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n",
          x, y);

  errno = 0;
  x = 0.0;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n",
          x, y);

  errno = 0;
  x = 720.0;
  y = sinh (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n",
          x, y);
}
```

Example Output

```
The hyperbolic sine of -1.500000 is -2.129279
The hyperbolic sine of 0.000000 is 0.000000
Error: range error
The hyperbolic sine of 720.000000 is inf
```

5.11.171 sinhf Function

Calculates the hyperbolic sine function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float sinhf (float` **`x`**`);`

Argument

`x` value for which to return the hyperbolic sine

Return Value

Returns the hyperbolic sine of `x`.

Remarks

If `x` is too large, a range error will occur and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  errno = 0;
  x = -1.0F;
  y = sinhf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n", x, y);

  errno = 0;
  x = 0.0F;
  y = sinhf (x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %f is %f\n", x, y);
}
```

Example Output

```
The hyperbolic sine of -1.000000 is -1.175201
The hyperbolic sine of 0.000000 is 0.000000
```

5.11.172 sinhl Function

Calculates the hyperbolic sine function of a long double precision floating-point value.

Include

`<math.h>`

Prototype

`long double sinhl(long double` **`x`**`);`

Argument

`x` value for which to return the hyperbolic sine

Return Value

Returns the hyperbolic sine of `x`.

Remarks

If `x` is too large, a range error will occur and `errno` will be set to `ERANGE`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = -1.0F;
  y = sinhl(x);
  if (errno)
```

```
    perror("Error");
  printf("The hyperbolic sine of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 0.0F;
  y = sinhl(x);
  if (errno)
    perror("Error");
  printf("The hyperbolic sine of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The hyperbolic sine of -1.000000 is -1.175201
The hyperbolic sine of 0.000000 is 0.000000
```

5.11.173 sqrt Function

Calculates the square root of a double precision floating-point value.

Include

`<math.h>`

Prototype

`double sqrt(double` **`x`**`);`

Argument

`x` a non-negative floating-point value

Return Value

Returns the non-negative square root of `x`.

Remarks

If `x` is less than 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = 0.0;
  y = sqrt(x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f\n", x, y);

  errno = 0;
  x = 9.5;
  y = sqrt(x);
  if (errno)
    perror("Error");
  printf("The square root of %f is %f\n", x, y);

  errno = 0;
  x = -25.0;
  y = sqrt(x);
  if (errno)
    perror("Error");
```

```
  printf("The square root of %f is %f\n", x, y);
}
```

Example Output

```
The square root of 0.000000 is 0.000000
The square root of 9.500000 is 3.082207
Error: domain error
The square root of -25.000000 is nan
```

5.11.174 sqrtf Function

Calculates the square root of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float sqrtf(float x);`

Argument

x a non-negative floating-point value

Return Value

Returns the non-negative square root of `x`.

Remarks

If `x` is less than 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;

  errno = 0;
  x = sqrtf (0.0F);
  if (errno)
    perror("Error");
  printf("The square root of 0.0F is %f\n", x);

  errno = 0;
  x = sqrtf (9.5F);
  if (errno)
    perror("Error");
  printf("The square root of 9.5F is %f\n", x);

  errno = 0;
  x = sqrtf (-25.0F);
  if (errno)
    perror("Error");
  printf("The square root of -25F is %f\n", x);
}
```

Example Output

```
The square root of 0.0F is 0.000000
The square root of 9.5F is 3.082207
Error: domain error
The square root of -25F is nan
```

5.11.175 sqrtl Function

Calculates the square root of a long double precision floating-point value.

Include

`<math.h>`

Prototype

```
long double sqrtf(long double x);
```

Argument

x a non-negative floating-point value

Return Value

Returns the non-negative square root of `x`.

Remarks

If `x` is less than 0, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x;

  errno = 0;
  x = sqrtl(0.0F);
  if (errno)
    perror("Error");
  printf("The square root of 0.0F is %Lf\n", x);

  errno = 0;
  x = sqrtl(9.5F);
  if (errno)
    perror("Error");
  printf("The square root of 9.5F is %Lf\n", x);

  errno = 0;
  x = sqrtl(-25.0F);
  if (errno)
    perror("Error");
  printf("The square root of -25F is %Lf\n", x);
}
```

Example Output

```
The square root of 0.0F is 0.000000
The square root of 9.5F is 3.082207
Error: domain error
The square root of -25F is nan
```

5.11.176 tan Function

Calculates the trigonometric tangent function of a double precision floating-point value.

Include

`<math.h>`

Prototype

```
double tan(double x);
```

Argument

x value for which to return the tangent

Return Value

Returns the tangent of x specified in radians in the range $[-1, 1]$. NaN is returned if x is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  errno = 0;
  x = -1.0;
  y = tan (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n", x, y);

  errno = 0;
  x = 0.0;
  y = tan (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n", x, y);
}
```

Example Output

```
The tangent of -1.000000 is -1.557408
The tangent of 0.000000 is 0.000000
```

5.11.177 tanf Function

Calculates the trigonometric tangent function of a single precision floating-point value.

Include

`<math.h>`

Prototype

`float tanf(float x);`

Argument

x value for which to return the tangent

Return Value

Returns the tangent of x specified in radians in the range $[-1, 1]$. NaN is returned if x is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
```

```
  float x, y;

  errno = 0;
  x = -1.0F;
  y = tanf (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n", x, y);

  errno = 0;
  x = 0.0F;
  y = tanf (x);
  if (errno)
    perror("Error");
  printf("The tangent of %f is %f\n", x, y);
}
```

Example Output

```
The tangent of -1.000000 is -1.557408
The tangent of 0.000000 is 0.000000
```

5.11.178 tanl Function

Calculates the trigonometric tangent function of a long double precision floating-point value.

Include

`<math.h>`

Prototype

`long double tanl(long double` **`x`**`);`

Argument

x value for which to return the tangent

Return Value

Returns the tangent of `x` specified in radians in the range $[-1, 1]$. NaN is returned if `x` is $\pm\infty$.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  errno = 0;
  x = -1.0;
  y = tanl(x);
  if (errno)
    perror("Error");
  printf("The tangent of %Lf is %Lf\n", x, y);

  errno = 0;
  x = 0.0;
  y = tanl(x);
  if (errno)
    perror("Error");
  printf("The tangent of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The tangent of -1.000000 is -1.557408
The tangent of 0.000000 is 0.000000
```

5.11.179 tanh Function

Calculates the hyperbolic tangent function of a double precision floating-point value.

Include

`<math.h>`

Prototype

```
double tanh(double x);
```

Argument

x value for which to return the hyperbolic tangent

Return Value

Returns the hyperbolic tangent of `x` in the ranges of -1 to 1 inclusive.

Remarks

No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = -1.0;
  y = tanh(x);
  printf("The hyperbolic tangent of %f is %f\n", x, y);

  x = 2.0;
  y = tanh(x);
  printf("The hyperbolic tangent of %f is %f\n", x, y);
}
```

Example Output

```
The hyperbolic tangent of -1.000000 is -0.761594
The hyperbolic tangent of 2.000000 is 0.964028
```

5.11.180 tanhf Function

Calculates the hyperbolic tangent function of a single precision floating-point value.

Include

`<math.h>`

Prototype

```
float tanhf(float x);
```

Argument

x value for which to return the hyperbolic tangent

Return Value

Returns the hyperbolic tangent of `x` in the ranges of -1 to 1 inclusive.

Remarks

No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  x = -1.0F;
  y = tanhf(x);
  printf("The hyperbolic tangent of %f is %f\n", x, y);

  x = 0.0F;
  y = tanhf(x);
  printf("The hyperbolic tangent of %f is %f\n", x, y);
}
```

Example Output

```
The hyperbolic tangent of -1.000000 is -0.761594
The hyperbolic tangent of 0.000000 is 0.000000
```

5.11.181 tanhl Function

Calculates the hyperbolic tangent function of a long double precision floating-point value.

Include

`<math.h>`

Prototype

`long double tanhl(long double` **`x`**`);`

Argument

`x` value for which to return the hyperbolic tangent

Return Value

Returns the hyperbolic tangent of `x` in the ranges of -1 to 1 inclusive..

Remarks

No domain or range error will occur.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = -1.0F;
  y = tanhl(x);
  printf("The hyperbolic tangent of %Lf is %Lf\n", x, y);

  x = 0.0F;
```

```
  y = tanhl(x);
  printf("The hyperbolic tangent of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The hyperbolic tangent of -1.000000 is -0.761594
The hyperbolic tangent of 0.000000 is 0.000000
```

5.11.182 tgamma Function

Calculates the gamma function of a double precision floating-point argument.

Attention: This function is not implemented by MPLAB XC8 C compilers.

Include

`<math.h>`

Prototype

`double tgamma(double x);`

Argument

x value for which to evaluate the gamma function

Return Value

Calculates the gamma function of the argument.

Remarks

If `x` is negative or if the result cannot be represented with `x` is zero, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = 0.5;
  y = tgamma(x);
  if(errno)
    perror("Error");
  printf("The gamma function of %f is %f\n", x, y);

  x = -0.75;
  y = tgamma(x);
  if(errno)
    perror("Error");
  printf("The gamma function of %f is %f\n", x, y);
}
```

Example Output

```
The gamma function of 0.500000 is 1.772454
The gamma function of -0.750000 is -4.834147
```

5.11.183 tgammaf Function

Calculates the gamma function of a single precision floating-point argument.

Attention: This function is not implemented by MPLAB XC8 C compilers.

Include

`<math.h>`

Prototype

```
float tgammaf(float x);
```

Argument

x value for which to evaluate the gamma function

Return Value

Calculates the gamma function of the argument.

Remarks

If `x` is negative or if the result cannot be represented with `x` is zero, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  x = 0.5;
  y = tgammaf(x);
  if(errno)
    perror("Error");
  printf("The gamma function of %f is %f\n", x, y);

  x = -0.75;
  y = tgammaf(x);
  if(errno)
    perror("Error");
  printf("The gamma function of %f is %f\n", x, y);
}
```

Example Output

```
The gamma function of 0.500000 is 1.772454
The gamma function of -0.750000 is -4.834147
```

5.11.184 tgammal Function

Calculates the gamma function of a long double precision floating-point argument.

Attention: This function is not implemented by MPLAB XC8 C compilers.

Include

`<math.h>`

Prototype

```
long double tgammal(long double x);
```

Argument

x value for which to evaluate the gamma function

Return Value

Calculates the gamma function of the argument.

Remarks

If `x` is negative or if the result cannot be represented with `x` is zero, a domain error will occur and `errno` will be set to `EDOM`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = 0.5;
  y = tgammal(x);
  if(errno)
    perror("Error");
  printf("The gamma function of %Lf is %Lf\n", x, y);

  x = -0.75;
  y = tgammal(x);
  if(errno)
    perror("Error");
  printf("The gamma function of %Lf is %Lf\n", x, y);
}
```

Example Output

```
The gamma function of 0.500000 is 1.772454
The gamma function of -0.750000 is -4.834147
```

5.11.185 trunc Function

Rounds the argument rounded to an integer value no large than the argument value.

Include

`<math.h>`

Prototype

```
double trunc(double x);
```

Argument

x the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value that is no larger in magnitude that the original argument. The rounded integer is returned as a double precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x, y;

  x = -10.103;
  y = trunc(x);
  printf("The nearest integer value to %f is %f\n", x, y);

  x = 10.9;
  y = trunc(x);
  printf("The nearest integer value to %f is %f\n", x, y);
}
```

Example Output

```
The nearest integer value to -10.103000 is -10.000000
The nearest integer value to 10.900000 is 10.000000
```

5.11.186 truncf Function

Rounds the argument rounded to an integer value no large than the argument value.

Include

`<math.h>`

Prototype

`float truncf(float x);`

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value that is no larger in magnitude that the original argument. The rounded integer is returned as a single precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  float x, y;

  x = -10.103;
  y = truncf(x);
  printf("The nearest integer value to %f is %f\n", x, y);

  x = 10.9;
  y = truncf(x);
  printf("The nearest integer value to %f is %f\n", x, y);
}
```

Example Output

```
The nearest integer value to -10.103000 is -10.000000
The nearest integer value to 10.900000 is 10.000000
```

5.11.187 truncl Function

Rounds the argument rounded to an integer value no large than the argument value.

Include

`<math.h>`

Prototype

`long double truncl(long double x);`

Argument

x	the value to round

Return Value

Returns the value of `x` rounded to the nearest integer value that is no larger in magnitude that the original argument. The rounded integer is returned as a long double precision floating-point value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long double x, y;

  x = -10.103;
  y = truncl(x);
  printf("The nearest integer value to %Lf is %Lf\n", x, y);

  x = 10.9;
  y = truncl(x);
  printf("The nearest integer value to %Lf is %Lf\n", x, y);
}
```

Example Output

```
The nearest integer value to -10.103000 is -10.000000
The nearest integer value to 10.900000 is 10.000000
```

5.12 <setjmp.h> Non-Local Jumps

The header file `setjmp.h` consists of a type, macro. and function that allow control transfers that bypass the normal function call and return process.

Attention: The MPLAB XC8 C Compiler does not implement this header for Baseline or non-enhanced Mid-range devices.

5.12.1 <setjmp.h> Types

jmp_buf Type
A type that is an integer array used by `setjmp` and `longjmp` to save and restore the program environment.

Attention: The MPLAB XC8 C Compiler does not implement this header for Baseline or non-enhanced Mid-range devices.

Include

`<setjmp.h>`

5.12.2 longjmp Function

A function that restores the environment saved by `setjmp`.

Attention: The MPLAB XC8 C Compiler does not implement this header for Baseline or non-enhanced Mid-range devices.

Include

`<setjmp.h>`

Prototype

```
void longjmp(jmp_buf env, int val);
```

Arguments

env	variable where environment is stored
val	value to be returned to `setjmp` call

Remarks

The value parameter, `val`, should be non-zero. If `longjmp` is invoked from a nested signal handler (that is, invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

jmp_buf jb;

void inner (void)
{
    longjmp(jb, 5);
}
int main (void)
{
  int i;
  if((i = setjmp(jb))) {
    printf("setjmp returned %d\n" i);
    exit(0);
  }
  printf("setjmp returned 0 - good\n");
  printf("calling inner...\n");
  inner();
```

```
  printf("inner returned - bad!\n");
}
```

Example Output

```
setjmp returned 0 - good
calling inner...
setjmp returned 5
```

5.12.3 setjmp Function

A macro that saves the current state of the program for later use by the `longjmp` function.

Attention: The MPLAB XC8 C Compiler does not implement this header for Baseline or non-enhanced Mid-range devices.

Include

`<setjmp.h>`

Prototype

`int setjmp(jmp_buf env)`

Argument

env	the array where the environment will be stored

Return Value

If the return is from a direct call, `setjmp` returns zero. If the return is from a call to `longjmp`, `setjmp` returns a non-zero value, even if `longjmp` was called with its `val` argument set to 0, in which case `setjmp` will return 1.

Remarks

The `setjmp` function can return as a result of a direct call, when the environment is stored, or from a subsequent call to `longjmp`, which uses the stored environment to continue execution from the saved instance of the previously called `setjmp` function.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

jmp_buf jb;

void inner (void)
{
    longjmp(jb, 5);
}

int main (void)
{
  int i;
  if((i = setjmp(jb))) {
    printf("setjmp returned %d\n", i);
    exit(0);
  }
  printf("setjmp returned 0 - good\n");
  printf("calling inner...\n");
  inner();
  printf("inner returned - bad!\n");
}
```

Example Output

```
setjmp returned 0 - good
calling inner...
setjmp returned 5
```

5.13 <signal.h> Signal Handling

The header file `signal.h` consists of a type, several macros and two functions that specify how the program handles signals while it is executing. A signal is a condition that may be reported during the program execution. Signals are synchronous, occurring under software control via the `raise` function.

Attention: This header is not implemented by MPLAB XC8.

5.13.1 sig_atomic_t Type

Integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

Attention: This type is not implemented by MPLAB XC8.

Include

`<signal.h>`

Remarks

Objects of this type (a `volatile signed char` for MPLAB XC8 for AVR MCUs and an `int` for other compilers that support signals) can be accessed atomically by a signal handler.

5.13.2 Signal Function Macros

Macros representing pointers to signal functions that can be used with the `signal()` function.

Attention: These macros are not implemented by MPLAB XC8.

Include

`<signal.h>`

Remarks

These macros are constant expressions with distinct values that have a type compatible with the second argument to and the return value of the `signal()` function.

Macro	Indicates
`SIG_DEF`	When passed to as the second argument to `signal()`, a default signal handler should be used on the signal.
`SIG_ERR`	When returned from `signal()`, the requested signal handling could not be established.

.....continued	
Macro	**Indicates**
SIG_IGN	When passed to as the second argument to signal(), the signal should be ignored.

5.13.3 SIGABRT Macro

Name for the abnormal termination signal.

Attention: This macro is not implemented by MPLAB XC8.

Include

<signal.h>

Remarks

The SIGABRT macro expands to a positive integer constant expression with type int that represents an abnormal termination signal and is used in conjunction with the raise() or signal() functions. The default behavior of the raise() function (action identified by SIG_DFL) is to output to the standard error stream: abort - terminating.

See the example accompanying signal() to see general usage of signal names and signal handling.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

```
#include <signal.h>
#include <stdio.h>
int main(void)
{
  raise(SIGABRT);
  printf("Program never reaches here.");
}
```

Example Output

```
ABRT
```

where ABRT stands for “abort.”

5.13.4 SIGFPE Macro

Signals floating-point errors such as for division by zero or result out of range.

Attention: This macro is not implemented by MPLAB XC8.

Include

<signal.h>

Remarks

SIGFPE is used as an argument for raise and/or signal. When used, the default behavior is to print an arithmetic error message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See signal for an example of a user-defined function.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  raise(SIGFPE);
  printf("Program never reaches here");
}
```

Example Output

```
FPE
```

where `FPE` stands for “floating-point error.”

5.13.5 SIGILL Macro

Signals illegal instruction.

Attention: This macro is not implemented by MPLAB XC8.

Include

`<signal.h>`

Remarks

`SIGILL` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print an invalid executable code message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user-defined function.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  raise(SIGILL);
  printf("Program never reaches here");
}
```

Example Output

```
ILL
```

where `ILL` stands for “illegal instruction.”

5.13.6 SIGINT Macro

Interrupt signal.

Attention: This macro is not implemented by MPLAB XC8.

Include

`<signal.h>`

Remarks

`SIGINT` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print an interruption message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user-defined function.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  raise(SIGINT);
  printf("Program never reaches here.");
}
```

Example Output

```
INT
```

where `INT` stands for “interruption.”

5.13.7 SIGSEGV Macro

Signals invalid access to storage.

Attention: This macro is not implemented by MPLAB XC8.

Include

`<signal.h>`

Remarks

`SIGSEGV` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print an invalid storage request message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user-defined function.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  raise(SIGSEGV);
  printf("Program never reaches here.");
}
```

Example Output

```
SEGV
```

where `SEGV` stands for “invalid storage access.”

5.13.8 SIGTERM Macro

Signals a termination request.

Attention: This macro is not implemented by MPLAB XC8.

Include

`<signal.h>`

Remarks

`SIGTERM` is used as an argument for `raise` and/or `signal`. When used, the default behavior is to print a termination request message and terminate the calling program. This may be overridden by a user function that defines the signal handler actions. See `signal` for an example of a user-defined function.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  raise(SIGTERM);
  printf("Program never reaches here.");
}
```

Example Output

```
TERM
```

where `TERM` stands for “termination request.”

5.13.9 raise Function

Reports a synchronous signal.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. Limited support of signals is provided by MPLAB XC8 for AVR MCUs.

Include

`<signal.h>`

Prototype

```
int raise(int sig);
```

Argument

`sig`	signal name

Return Value

Returns a 0 if successful; otherwise, returns a non-zero value.

Remarks

`raise` sends the signal identified by `sig` to the executing program.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void __attribute__((interrupt(auto_psv))) _MathError(void)
{
  raise(SIGILL);
  INTCON1bits.MATHERR = 0;
}

void illegalinsn(int idsig)
{
  printf("Illegal instruction executed\n");
  exit(1);
}
int main(void)
{
  int x, y;
  div_t z;

  signal(SIGILL, illegalinsn);
  x = 7;
  y = 0;
  z = div(x, y);
  printf("Program never reaches here");
}
```

Example Output

```
Illegal instruction executed
```

Example Explanation

This example requires the linker script, `p30f6014.gld`. There are three parts to this example.

First, an interrupt handler is written for the interrupt vector, `_MathError`, to handle a math error by sending an illegal instruction, `signal (SIGILL)`, to the executing program. The last statement in the interrupt handler clears the exception flag.

Second, the function `illegalinsn` will print an error message and call `exit`.

Third, in `main`, `signal (SIGILL, illegalinsn)` sets the handler for `SIGILL` to the function `illegalinsn`.

When a math error occurs due to a divide by zero, the `_MathError` interrupt vector is called, which in turn will raise a signal that will call the handler function for `SIGILL`: the function `illegalinsn`. Thus, error messages are printed and the program is terminated.

5.13.10 signal Function

Controls interrupt signal handling.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. Limited support of signals is provided by MPLAB XC8 for AVR MCUs.

Include

`<signal.h>`

Prototype

```
void (*signal(int sig, void(*func)(int)))(int);
```

Arguments

sig	signal name
func	function to be executed

Return Value

Returns the previous value of `func`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

/* Signal handler function */
void mysigint(int id)
{
  printf("SIGINT received\n");
}

int main(void)
{
  /* Override default with user defined function */
  signal(SIGINT, mysigint);
  raise(SIGINT);

  /* Ignore signal handler */
  signal(SIGILL, SIG_IGN);
  raise(SIGILL);
  printf("SIGILL was ignored\n");

  /* Use default signal handler */
  raise(SIGFPE);
  printf("Program never reaches here.");
}
```

Example Output

```
SIGINT received
SIGILL was ignored
FPE
```

Example Explanation

The function `mysigint` is the user-defined signal handler for `SIGINT`. Inside the main program, the function signal is called to set up the signal handler (`mysigint`) for the signal `SIGINT` that will override the default actions. The function raise is called to report the signal `SIGINT`. This causes the signal handler for `SIGINT` to use the user-defined function (`mysigint`) as the signal handler so it prints the “SIGINT received” message.

Next, the function signal is called to set up the signal handler `SIG_IGN` for the signal `SIGILL`. The constant `SIG_IGN` is used to indicate the signal should be ignored. The function raise is called to report the signal `SIGILL` that is ignored.

The function raise is called again to report the signal `SIGFPE`. Since there is no user-defined function for `SIGFPE`, the default signal handler is used so the message “FPE” is printed (which stands for “arithmetic error - terminating”). Then, the calling program is terminated. The `printf` statement is never reached.

5.14 <stdarg.h> Variable Argument Lists

The header file `stdarg.h` supports functions with variable argument lists.

Variable argument lists allow functions to have an arbitrary number of arguments without corresponding parameter declarations, thus, the number and type of these arguments can differ on each call to the function.

A function with a variable argument list must have at least one named argument. The arguments in the argument list are represented by an ellipsis, `...`. To read the argument list, an object of type `va_list` must be declared inside the function to hold information about the argument list. The `va_start` macro is used to initialize the `va_list` object so that the argument values can be obtained, the `va_arg` macro will access the next argument in the list, and the `va_end` macro ensures that the function is in a state whereby it can successfully return after all the required arguments have been read.

5.14.1 va_list Type

The type `va_list()` declares an object that holds information about the argument list needed by the `va_start()` and `va_arg()` macros.

Include

`<stdarg.h>`

Example

See example at `va_arg`.

5.14.2 va_arg Macro

Gets the next argument in a variable argument list represented by a `va_list` object.

Include

`<stdarg.h>`

Prototype

type va_arg(va_list **ap**, *type*)

Arguments

ap	pointer to list of arguments
type	type of argument to be retrieved

Return Value

Returns the next argument in the argument list represented by `ap` with the type specified as the second argument.

Remarks

The `ap` object must have been initialized by calls to either `va_start` or `va_copy` before `va_arg` can be used. If there is no next argument available or it is not of the type requested, the behavior is undefined.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void tprint(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  while (*fmt)
  {
    switch (*fmt)
    {
      case '%':
            fmt++;
            if (*fmt == 'd')
            {
              int d = va_arg(ap, int);
              printf("<%d> is an integer\n",d);
            }
            else if (*fmt == 's')
```

```
            {
              char *s = va_arg(ap, char*);
              printf("<%s> is a string\n", s);
            }
            else
            {
              printf("%%%c is an unknown format\n",
                   *fmt);
            }
            fmt++;
            break;
      default:
            printf("%c is unknown\n", *fmt);
            fmt++;
            break;
    }
  }
  va_end(ap);
}
int main(void)
{
  tprint("%d%s.%c", 83, "This is text.", 'a');
}
```

Example Output

```
<83> is an integer
<This is text.> is a string
. is unknown
%c is an unknown format
```

5.14.3 va_copy Macro

Copies a `va_list` object.

Include

`<stdarg.h>`

Prototype

`void va_copy(va_list` **`dest`**`, va_list` **`src`**`);`

Arguments

`dest`	the `va_list` object to be initialized
`src`	the `va_list` object to copy

Remarks

The function copies the `src` object into `dest`, such that the state of `dest` will reflect any call made to `va_start` to intialize `src` as well as the same sequence of `va_arg` macro calls that modified `src`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <stdio.h>
#include <stdarg.h>
#include <limits.h>

/* print values as a percentage of the maximum value
 * list ends with 0 value
 */

void PrintAsPercent(int first, ...)
{
  char * buffer;
  int val = first;
  int valMax = INT_MIN;
  va_list vl_parse, vl_print;
```

```
  va_start(vl_parse, first);
  /* copy, after having called va_start */
  va_copy(vl_print, vl_parse);

  /* find the largest value */
  while(val != 0) {
    val = va_arg(vl_parse, int);
    if(val > valMax)
      valMax = val;
  }
  va_end(vl_parse);

  /* va_start not necessary with va_print */
  val = first;
  while(val != 0) {
    printf("%d%% ", val*100/valMax);
    val = va_arg(vl_print, int);
  }
  va_end(vl_print);
  printf("\n");
}

int main(void)
{
  PrintAsPercent(10, 23, 156, 42, 88, 0);
}
```

Example Output

```
6% 14% 100% 26% 56%
```

5.14.4 va_end Macro

Ensure a function with a variable argument list can return successfully.

Include

`<stdarg.h>`

Prototype

`void va_end(va_list` **`ap`**`)`

Argument

`ap` the `va_list` object from which arguments are being obtained

Remarks

The `va_end` macro ensures that the function whose argument list is being read is in a state whereby it can successfully return after all the required arguments have been read from the list. Further calls to `va_arg` should not be made unless `ap` is reinitialized by a call to `va_start`.

Example

See example at `va_arg`.

5.14.5 va_start Macro

Initializes a `va_list` object.

Include

`<stdarg.h>`

Prototype

`void va_start(va_list` **`ap`**`,` ***`last_arg`***`)`

Arguments

ap	pointer to list of arguments
last_arg	last named parameter before the optional arguments

Remarks

Initializes the `ap` object so that it is usable with `va_arg` and `va_end` macros. The last name parameter should not be declared with the `register` storage class, be a function or array type, or be a type that is not compatible with the type that results after application of the default argument promotions.

Example

See example at `va_arg`.

5.15 <stdbool.h> Boolean Types and Values

The header file `stdbool.h` consists of macros that are usable when working with boolean types.

5.15.1 stdbool.h Types and Values

bool Macro
Alternate type name to `_Bool`.

Include

`<stdbool.h>`

Remarks

The `bool` macro allows the use of an alternate type name to `_Bool`.

true Macro
Symbolic form of the true state.

Include

`<stdbool.h>`

Remarks

The `true` macro provides a symbolic form of the true state that can be used with objects of type `_Bool` and is defined as the value 1.

false Macro
Symbolic form of the false state.

Include

`<stdbool.h>`

Remarks

The `false` macro provides a symbolic form of the false state that can be used with objects of type `_Bool` and is defined as the value 0.

_ _bool_true_false_are_defined Macro
Flag to indicate that the boolean macros are defined and are usable.

Include

`<stdbool.h>`

Remarks

The `__bool_true_false_are_defined` macro is set if the `bool`, `true` and `false` macros are defined and are usable. It is assigned the value 1 in the header, but this macro along with the `bool`, `true` and `false` macros may be undefined and potentially redefined by the program.

5.16 <stddef.h> Common Definitions

The header file `stddef.h` consists of several types and macros that are of general use in programs.

5.16.1 stddef.h Types and Macros

ptrdiff_t Type
Type used to represent the difference in two pointer values.

Include

`<stddef.h>`

Remarks

The `ptrdiff_t` type is a signed integer type that is used to represent the difference between two pointer values.

size_t Type
Type used to represent the result of the `sizeof` operator.

Include

`<stddef.h>`

Remarks

The `size_t` type is an unsigned integer type that is used to represent the size of an object, as returned by the `sizeof` operator.

wchar_t Type
Type used to represent the values of the largest extended character set.

Include

`<stddef.h>`

Remarks

The `wchar_t` type is an integer type that is used to represent the values of the largest extended character set.

5.16.2 size_t Type

An unsigned integer type used by the result of the `sizeof` operator

Include

`<stddef.h>`

`<stdio.h>`

`<stdlib.h>`

`<string.h>`

`<time.h>`

`<wchar.h>`

Definition

```
typedef unsigned size_t;
```

5.16.3 wchar_t Type

An integer type capable of holding values that can represent distinct codes for all members of the largest extended character set specified among the supported locales.

Include

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

5.16.4 NULL Macro

A constant value which represent a null pointer constant. It's value and type is implementation defined.

Include

```
<locale.h>
<stddef.h>
<stdio.h>
<stdlib.h>
<string.h>
<time.h>
<wchar.h>
```

Definition

```
#define NULL ((void*)0)
```

5.16.5 offsetof Macro

Gives the offset of a structure member from the beginning of the structure.

Include

```
<stddef.h>
```

Prototype

size_t offsetof(*type* **T,** *member-designator* **mbr**)

Arguments

T	the type of the structure
mbr	the name of the member in the structure T

Return Value

Returns the offset in bytes of the specified member (mbr) from the beginning of the structure with type size_t.

Remarks

The macro offsetof is undefined for bit-fields. An error message will occur if bit-fields are used.

When building for PIC devices with MPLAB XC8, the expression represented by this macro is evaluated at runtime, hence it does not represent a constant expression and cannot be used where a constant expression is required. For other devices, this macro represents a constant expression.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

```
struct info {
  char item1[5];
  int item2;
  char item3;
  float item4;
};

int main(void)
{
  printf("Offset of item1 = %d\n", offsetof(struct info,item1));
  printf("Offset of item2 = %d\n", offsetof(struct info,item2));
  printf("Offset of item3 = %d\n", offsetof(struct info,item3));
  printf("Offset of item4 = %d\n", offsetof(struct info,item4));
}
```

Example Output

```
Offset of item1 = 0
Offset of item2 = 6
Offset of item3 = 8
Offset of item4 = 10
```

Example Explanation

This program shows the offset in bytes of each structure member from the start of the structure. Such address information is particularly useful when the compiler pads structure members to ensure they are aligned in memory. This does not take place for structures in MPLAB XC8, but the above example shows how this would be reflected in the return values for other compilers.

In the above example, although item1 is only 5 bytes (`char item1[5]`), padding is added so the address of `item2` falls on an even boundary. The same occurs with `item3`; it is 1 byte (`char item3`) with 1 byte of padding.

5.17 <stdint.h> Integer Types

The header file `stdint.h` consists of types and macros that can be used to define integer types whose size meets certain parameters.

5.17.1 Fixed-Width Integer Types

Typedef names that designate signed and unsigned integer types with an exact width.

Attention: The types for 24-bit objects are only supported when using PIC devices and MPLAB XC8. The types for 64-bit objects are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<stdint.h>`

Type	Description
int8_t	Signed integer of exactly 8 bits width.
int16_t	Signed integer of exactly 16 bits width.
int24_t	Signed integer of exactly 24 bits width (see Attention note).
int32_t	Signed integer of exactly 32 bits width.
int64_t	Signed integer of exactly 64 bits width (see Attention note).
uint8_t	Unsigned integer of exactly 8 bits width.
uint16_t	Unsigned integer of exactly 16 bits width.
uint24_t	Unsigned integer of exactly 24 bits width (see Attention note).

.....continued	
Type	**Description**
`uint32_t`	Unsigned integer of exactly 32 bits width.
`uint64_t`	Unsigned integer of exactly 64 bits width (see Attention note).

5.17.2 Minimum Width Integer Types

Typedef names that designate signed and unsigned integer types with at least the specified width.

Attention: The types for 24-bit objects are only supported when using PIC devices and MPLAB XC8. The types for 64-bit objects are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<stdint.h>`

Type	Description
`int_least8_t`	Signed integer of at least 8 bits width.
`int_least16_t`	Signed integer of at least 16 bits width.
`int_least24_t`	Signed integer of at least 24 bits width.
`int_least32_t`	Signed integer of at least 32 bits width.
`int_least64_t`	Signed integer of at least 64 bits width (see Attention note).
`uint_least8_t`	Unsigned integer of at least 8 bits width.
`uint_least16_t`	Unsigned integer of at least 16 bits width.
`uint_least24_t`	Unsigned integer of at least 24 bits width.
`uint_least32_t`	Unsigned integer of at least 32 bits width.
`uint_least64_t`	Unsigned integer of at least 64 bits width (see Attention note).

5.17.3 Fastest Minimum-Width Integer Types

Typedef names that designate signed and unsigned integer types with at least the specified width and that are usually the fastest to work with.

Attention: The types for 24-bit objects are only supported when using PIC devices and MPLAB XC8. The types for 64-bit objects are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<stdint.h>`

Type	Description
`int_fast8_t`	The fastest signed integer of at least 8 bits width.
`int_fast16_t`	The fastest signed integer of at least 16 bits width.
`int_fast24_t`	The fastest signed integer of at least 24 bits width.
`int_fast32_t`	The fastest signed integer of at least 32 bits width.

.....continued	
Type	**Description**
int_fast64_t	The fastest signed integer of at least 64 bits width (see Attention note).
uint_fast8_t	The fastest unsigned integer of at least 8 bits width.
uint_fast16_t	The fastest unsigned integer of at least 16 bits width.
uint_fast24_t	The fastest unsigned integer of at least 24 bits width.
uint_fast32_t	The fastest unsigned integer of at least 32 bits width.
uint_fast64_t	The fastest unsigned integer of at least 64 bits width (see Attention note).

5.17.4 Greatest Width Integer Types

Typedef names that designate the largest width signed and unsigned integer types.

Include

`<stdint.h>`

Type	Description
intmax_t	Signed integer type large enough to hold any signed integer type.
uintmax_t	Unsigned integer type large enough to hold any unsigned integer type.

5.17.5 Integer Types For Pointer Objects

Typedef names that designate signed and unsigned integer types with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer

Include

`<stdint.h>`

Type	Description
intptr_t	Signed integer type capable of holding a pointer to `void` with no loss of information.
uintptr_t	Unsigned integer type capable of holding a pointer to `void` with no loss of information.

5.17.6 WCHAR_MAX Macro

A constant value which represent the maximum size of the `wchar_t` type.

Attention: This macro is implemented only by MPLAB XC32 C compilers.

Include

`<stdint.h>`

`<wchar.h>`

5.17.7 WCHAR_MIN Macro

A constant value which represent the minimum size of the `wchar_t` type.

Attention: This macro is implemented only by MPLAB XC32 C compilers.

Include

`<stdint.h>`

`<wchar.h>`

5.17.8 Limits of Fixed-Width Integer Types

Macros that specify the minimum and maximum limits of the types with fixed width declared in `<stdint.h>`.

Attention: The macros for 24-bit objects are only supported when using PIC devices and MPLAB XC8. The macros for 64-bit objects are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<stdint.h>`

Remarks

Each macro name corresponds to the `<stdint.h>` type with a similar name. The macro will expand to a constant expression suitable for use in `#if` preprocessing directives, and this expression shall have the same type as an object of the corresponding type converted according to the integer promotions.

Type	Description
INT8_MIN	Minimum value of signed integer with 8 bits width.
INT8_MAX	Maximum value of signed integer with 8 bits width.
UINT8_MAX	Maximum value of unsigned integer with 8 bits width.
INT16_MIN	Minimum value of signed integer with 16 bits width.
INT16_MAX	Maximum value of signed integer with 16 bits width.
UINT16_MAX	Maximum value of unsigned integer with 16 bits width.
INT24_MIN	Minimum value of signed integer with 24 bits width (see Attention note).
INT24_MAX	Maximum value of signed integer with 24 bits width (see Attention note).
UINT24_MAX	Maximum value of unsigned integer with 24 bits width (see Attention note).
INT32_MIN	Minimum value of signed integer with 32 bits width.
INT32_MAX	Maximum value of signed integer with 32 bits width.
UINT32_MAX	Maximum value of unsigned integer with 32 bits width.
INT64_MIN	Minimum value of signed integer with 64 bits width (see Attention note).
INT64_MAX	Maximum value of signed integer with 64 bits width (see Attention note).
UINT64_MAX	Maximum value of unsigned integer with 64 bits width (see Attention note).

5.17.9 Limits of Minimum-Width Integer Types

Macros that specify the minimum and maximum limits of the types with at least the specified width declared in `<stdint.h>`.

Attention: The macros for 24-bit objects are only supported when using PIC devices and MPLAB XC8. The macros for 64-bit objects are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<stdint.h>`

Remarks

Each macro name corresponds to the `<stdint.h>` type with a similar name. The macro will expand to a constant expression suitable for use in `#if` preprocessing directives, and this expression shall have the same type as an object of the corresponding type converted according to the integer promotions.

Type	Description
INT_LEAST8_MIN	Minimum value of signed integer with at least at least 8 bits width.
INT_LEAST8_MAX	Maximum value of signed integer with at least 8 bits width.
UINT_LEAST8_MAX	Maximum value of unsigned integer with at least 8 bits width.
INT_LEAST16_MIN	Minimum value of signed integer with at least 16 bits width.
INT_LEAST16_MAX	Maximum value of signed integer with at least 16 bits width.
UINT_LEAST16_MAX	Maximum value of unsigned integer with at least 16 bits width.
INT_LEAST24_MIN	Minimum value of signed integer with at least 24 bits width (see Attention note).
INT_LEAST24_MAX	Maximum value of signed integer with at least 24 bits width (see Attention note).
UINT_LEAST24_MAX	Maximum value of unsigned integer with at least 24 bits width (see Attention note).
INT_LEAST32_MIN	Minimum value of signed integer with at least 32 bits width.
INT_LEAST32_MAX	Maximum value of signed integer with at least 32 bits width.
UINT_LEAST32_MAX	Maximum value of unsigned integer with at least 32 bits width.
INT_LEAST64_MIN	Minimum value of signed integer with at least 64 bits width (see Attention note).
INT_LEAST64_MAX	Maximum value of signed integer with at least 64 bits width (see Attention note).
UINT_LEAST64_MAX	Maximum value of unsigned integer with at least 64 bits width (see Attention note).

5.17.10 Limits of Fastest Minimum-Width Integer Types

Macros that specify the minimum and maximum limits of the fastest types with specified width declared in `<stdint.h>`.

Attention: The macros for 24-bit objects are only supported when using PIC devices and MPLAB XC8. The macros for 64-bit objects are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<stdint.h>`

Remarks

Each macro name corresponds to the <stdint.h> type with a similar name. The macro will expand to a constant expression suitable for use in #if preprocessing directives, and this expression shall have the same type as an object of the corresponding type converted according to the integer promotions.

Type	Description
INT_FAST8_MIN	Minimum value of fastest signed integer with at least 8 bits width.
INT_FAST8_MAX	Maximum value of fastest signed integer with at least 8 bits width.
UINT_FAST8_MAX	Maximum value of fastest unsigned integer with at least 8 bits width.
INT_FAST16_MIN	Minimum value of fastest signed integer with at least 16 bits width.
INT_FAST16_MAX	Maximum value of fastest signed integer with at least 16 bits width.
UINT_FAST16_MAX	Maximum value of fastest unsigned integer with at least 16 bits width.
INT_FAST24_MIN	Minimum value of fastest signed integer with at least 24 bits width (see Attention note).
INT_FAST24_MAX	Maximum value of fastest signed integer with at least 24 bits width (see Attention note).
UINT_FAST24_MAX	Maximum value of fastest unsigned integer with at least 24 bits width (see Attention note).
INT_FAST32_MIN	Minimum value of fastest signed integer with at least 32 bits width.
INT_FAST32_MAX	Maximum value of fastest signed integer with at least 32 bits width.
UINT_FAST32_MAX	Maximum value of fastest unsigned integer with at least 32 bits width.
INT_FAST64_MIN	Minimum value of fastest signed integer with at least 64 bits width (see Attention note).
INT_FAST64_MAX	Maximum value of fastest signed integer with at least 64 bits width (see Attention note).
UINT_FAST64_MAX	Maximum value of fastest unsigned integer with at least 64 bits width (see Attention note).

5.17.11 Limits for Greatest Width Integer Types

Macros that specify the minimum and maximum limits of the types with largest width declared in <stdint.h>.

Include

<stdint.h>

Remarks

Each macro name corresponds to the <stdint.h> type with a similar name. The macro will expand to a constant expression suitable for use in #if preprocessing directives, and this expression shall have the same type as an object of the corresponding type converted according to the integer promotions.

Type	Description
INTMAX_MIN	Minimum value of largest width signed integer type.
INTMAX_MAX	Maximum value of largest width signed integer type.
UINTMAX_MAX	Maximum value of largest width unsigned integer type.

5.17.12 Limits of Integer Types for Pointer Objects

Macros that specify the minimum and maximum limits of the pointer-related types declared in <stdint.h>.

Include

`<stdint.h>`

Remarks

Each macro name corresponds to the `<stdint.h>` type with a similar name. The macro will expand to a constant expression suitable for use in `#if` preprocessing directives, and this expression shall have the same type as an object of the corresponding type converted according to the integer promotions.

Type	Description
INTPTR_MIN	The minimum value of a signed integer type capable of holding a pointer.
INTPTR_MAX	The maximum value of a signed integer type capable of holding a pointer.
UINTPTR_MAX	The maximum value of an unsigned integer type capable of holding a pointer.

5.17.13 Limits of Other Integer Types

Macros that specify the minimum and maximum limits of the miscellaneous types declared in `<stdint.h>`.

Attention: The macros relating to wide characters are implemented only by MPLAB XC32 compilers. The macros relating to signals are not implemented by MPLAB XC8 for PIC MCUs. Limited support of signals is provided by MPLAB XC8 for AVR MCUs.

Include

`<stdint.h>`

Remarks

Each macro name corresponds to the indicated type. The macro will expand to a constant expression suitable for use in `#if` preprocessing directives, and this expression shall have the same type as an object of the corresponding type converted according to the integer promotions.

Type	Description
PTRDIFF_MIN	Minimum value of the `ptrdiff_t` type.
PTRDIFF_MAX	Maximum value of the `ptrdiff_t` type.
SIG_ATOMIC_MIN	Minimum value of the `sig_atomic_t` type (see Attention note).
SIG_ATOMIC_MAX	Maximum value of the `sig_atomic_t` type (see Attention note).
SIZE_MAX	Maximum value of the `size_t` type.
WCHAR_MIN	Minimum value of the `wchar_t` type (see Attention note).
WCHAR_MAX	Maximum value of the `wchar_t` type (see Attention note).
WINT_MIN	Minimum value of the `wint_t` type (see Attention note).
WINT_MAX	Maximum value of the `wint_t` type (see Attention note).

5.17.14 Minimum-width Integer Constant Macros

Macros that expand to integer constants suitable for initializing objects that have integer types corresponding to types defined in `<stdint.h>`.

Attention: The macros for 24-bit objects are only supported when using PIC devices and MPLAB XC8. The macros for 64-bit objects are not supported when using 8-bit AVR, Baseline, or non-enhanced Mid-range PIC devices with MPLAB XC8.

Include

`<stdint.h>`

Remarks

Each macro name corresponds to the `<stdint.h>` type with a similar name. The argument to these macros shall be a decimal, octal, or hexadecimal constant with a value that does not exceed the limits for the corresponding type.

Macro	Expands to
INT8_C	A signed integer constant with the specified argument value and type `int_least8_t`.
INT16_C	A signed integer constant with the specified argument value and type `int_least16_t`.
INT24_C	A signed integer constant with the specified argument value and type `int_least24_t`, (see Attention note).
INT32_C	A signed integer constant with the specified argument value and type `int_least32_t`
INT64_C	A signed integer constant with the specified argument value and type `int_least64_t`, (see Attention note).
UINT8_C	An unsigned integer constant with the specified argument value and type `uint_least8_t`.
UINT16_C	An unsigned integer constant with the specified argument value and type `uint_least16_t`.
UINT24_C	An unsigned integer constant with the specified argument value and type `uint_least24_t`, (see Attention note).
UINT32_C	An unsigned integer constant with the specified argument value and type `uint_least32_t`.
UINT64_C	An unsigned integer constant with the specified argument value and type `uint_least64_t`, (see Attention note).

Example

```
#include <stdint.h>
#include <stdio.h>
#include <inttypes.h>

int
main(void)
{
        uint_least64_t ref;

        ref = 1 << (sizeof(int)*8);
        printf("ref is %" PRIxLEAST64 " (oops)\n", ref);
        ref = UINT64_C(1) << (sizeof(int)*8);
        printf("ref is %" PRIxLEAST64 " (better)\n", ref);
}
```

Example Output

For builds where the `int` type (being the type assigned to the constant `1`) is 16 bits wide.

```
ref is 0x0 (oops)
ref is 0x10000 (better)
```

For builds where the `int` type is 32 bits wide.

```
ref is 0x0 (oops)
ref is 0x100000000 (better)
```

5.17.15 Greatest-width Integer Constant Macros

Macros that expand to integer constants suitable for initializing objects that have the largest integer types defined in `<stdint.h>`.

Include

`<stdint.h>`

Remarks

Each macro name corresponds to the `<stdint.h>` type with a similar name. The argument to these macros shall be a decimal, octal, or hexadecimal constant with a value that does not exceed the limits for the corresponding type.

Macro	Expands to
INTMAX_C	A signed integer constant with the specified value and type `intmax_t`
UINTMAX_C	A signed integer constant with the specified value and type `uintmax_t`

Example

```
#include <stdint.h>
#include <stdio.h>
#include <inttypes.h>

int main(void)
{
  intmax_t ref;

  ref = 1 << (sizeof(int)*8);
  printf("ref is 0x%" PRIxMAX " (oops)\n", ref);
  ref = INTMAX_C(1) << 32;(sizeof(int)*8);
  printf("ref is 0x%" PRIxMAX " (better)\n", ref);
}
```

Example Output

For builds where the `int` type (being the type assigned to the constant `1`) is 16 bits wide.

```
ref is 0x0 (oops)
ref is 0x10000 (better)
```

For builds where the `int` type is 32 bits wide.

```
ref is 0x0 (oops)
ref is 0x100000000 (better)
```

5.18 <stdio.h> Input and Output

The header file `stdio.h` consists of types, macros, and functions that perform input and output operations on streams. The MPLAB XC16/32 compilers also implement functions that permit operations on files, which can be opened as a stream using the `fopen` function and the `FILE` type.

A stream is a pipeline for the flow of data, hiding much of the complexity and non-uniformity of IO operations that is present when data is transferred to and from different devices and on different systems. At start-up, three streams are

automatically opened: `stdin`, `stdout` and `stderr`, representing the standard input, standard output, and standard error locations. The particular peripheral or device feature associated with these streams can be established by customizing helper functions, described in 5.18.1 Customizing IO Functions.

The `stdio.h` header also contains functions that read input (such as `scanf`) and write output (such as `printf`).

5.18.1 Customizing IO Functions

On hosted systems, the `stdin`, `stdout`, and `stderr` streams are typically mapped to peripheral devices such as displays and keyboards, but embedded hardware may not incorporate such components, and the choice and configuration of device peripherals that would drive these components will vary greatly.

Those IO functions which read from or write to any of the standard streams use support functions to define the source or destination of the standard streams, allowing functions like `printf()` to write to any device peripheral.

The following sections indicate the support functions for each compiler and how these should be configured.

5.18.1.1 Customizing Helper Functions for 8-bit PIC Devices

When using the MPLAB XC8 Compiler for PIC MCUs, the `putch()` and `getch()` functions need to be completed to define the `stdout` and `stdin` streams used by `printf()` and `scanf()` functions, respectively.

Printing using `printf()` will not work as expected until the `putch()` stub (found in the `pic/sources/c99/common/putch.c` of your compiler distribution) is completed. The `putch` function is called repetitively by `printf()` to send each byte of the formatted output. The `putch()` function should send the byte passed in as an argument to the required destination, for example a UART. See 4.1 Example code for 8-bit PIC MCUs for information on defining `putch()` so that printed text appears in an MPLAB X IDE simulator window.

Reading input using `scanf()` will not work as expected until the `getch()` stub (found in the `pic/sources/c99/common/getch.c` of your compiler distribution) is completed. The `getch()` function is called repetitively to obtain each bye of the input. The `getch()` function should return one byte read from required source, for example a UART. See 4.1 Example code for 8-bit PIC MCUs for information on defining `getch()` so that it can read text from a file when you are using the MPLAB X IDE simulator.

Once completed, the source files for `getch()` and `putch()` should be built with your other source files in the usual way.

5.18.1.2 Customizing Helper Functions for XC8 AVR

When using the MPLAB XC8 Compiler for AVR MCUs, helper functions that can read and/or write one byte of data need to be written to define the source and destination respectively of the `stdin` and `stdout` streams used by IO functions like `scanf()` and `printf()`. In addition, objects of type `FILE` need to be defined and associated with each streams.

The `FDEV_SETUP_STREAM()` macro can assist with stream configuration. It expands to an initializer that can be used with the definition of a user-supplied buffer of type `FILE`, setting up that buffer so that it can be used with a stream that is valid for stdio operations. Streams can be read-only, write-only, or read-write, based on one of this macro's flag values `_FDEV_SETUP_READ`, `_FDEV_SETUP_WRITE`, or `_FDEV_SETUP_RW`.

The following example is applicable to the AVR128DA48 Curiosity Nano board and has writable `FILE` buffer assigned to `stdout`, which maps to USART1 via the `uart_putchar()` function.

```
#include <xc.h>

#define F_CPU                           (4000000UL)     /* using default clock 4MHz*/
#define USART1_BAUD_RATE(BAUD_RATE)     ((float)(64 * 4000000 / (16 * (float)BAUD_RATE)) +
0.5)

#include <util/delay.h>
#include <stdio.h>

void USART1_init(void)
{
    PORTC.DIRSET = PIN0_bm;                             /* set pin 0 of PORT C (TXd) as
output*/
    PORTC.DIRCLR = PIN1_bm;                             /* set pin 1 of PORT C (RXd) as
input*/
```

```
    USART1.BAUD = (uint16_t)(USART1_BAUD_RATE(9600));   /* set the baud rate*/

    USART1.CTRLC = USART_CHSIZE0_bm| USART_CHSIZE1_bm;  /* set the data format to 8-bit*/

    USART1.CTRLB |= USART_TXEN_bm;                      /* enable transmitter*/
}

static int uart_putchar(char c, FILE * stream);
static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL, _FDEV_SETUP_WRITE);

static int
uart_putchar(char c, FILE *stream)
{
    if (c == '\n')
        uart_putchar('\r', stream);
    while(!(USART1.STATUS & USART_DREIF_bm))
    {
        ;
    }

    USART1.TXDATAL = c;
    return 0;
}

int main()
{
    USART1_init();
    stdout = &mystdout;

    int i = 0;
    while (1) {
        printf ("Hello world %d\n", i++);
    }
    _delay_ms(1000);
}
```

5.18.1.3 Customizing Helper Functions for XC16

The standard I/O relies on helper functions described in 6. Syscall Interface. These functions include `read()`, `write()`, `open()`, and `close()` which are called to read, write, open or close handles that are associated with standard I/O `FILE` pointers. The sources for these libraries are provided for you to customize as you wish.

It is recommended that these customized functions be allocated in a named section that begins with `.libc`. This will cause the linker to allocate them near to the rest of the library functions, which is where they ought to be.

The simplest way to redirect standard I/O to the peripheral of your choice is to select one of the default handles already in use. Also, you could open files with a specific name via `fopen()` by rewriting `open()` to return a new handle to be recognized by `read()` or `write()` as appropriate.

If only a specific peripheral is required, then you could associate handle `1 == stdout` or `2 == stderr` to another peripheral by writing the correct code to talk to the interested peripheral.

A complete generic solution might be:

```
/* should be in a header file */
enum my_handles {
    handle_stdin,
    handle_stdout,
    handle_stderr,
    handle_can1,
    handle_can2,
    handle_spi1,
    handle_spi2,
};

int _ ___attribute_ ___((_ ___weak_ ___, _ ___section_ ___(".libc")))
  open(const char *name, int access, int mode) {
     switch (name[0]) {
       case 'i' : return handle_stdin;
       case 'o' : return handle_stdout;
       case 'e' : return handle_stderr;
       case 'c' : return handle_can1;
       case 'C' : return handle_can2;
```

```
        case 's' : return handle_spi1;
        case 'S' : return handle_spi2;
        default: return handle_stderr;
    }
}
```

Single letters were used in this example because they are faster to check and use less memory. However, if memory is not an issue, you could use `strcmp` to compare full names.

In `write()`, you would write:

```
int __attribute__((__section__(".libc.write")))
  write(int handle, void *buffer, unsigned int len) {
     int i;
     volatile UxMODEBITS *umode = &U1MODEbits;
     volatile UxSTABITS *ustatus = &U1STAbits;
     volatile unsigned int *txreg = &U1TXREG;
     volatile unsigned int *brg = &U1BRG;

     switch (handle)
     {
     default:
     case 0:
     case 1:
     case 2:
        if ((_ ___C30_UART != 1) && (&U2BRG)) {
           umode = &U2MODEbits;
           ustatus = &U2STAbits;
           txreg = &U2TXREG;
           brg = &U2BRG;
        }
        if ((umode->UARTEN) == 0)
        {
           *brg = 0;
           umode->UARTEN = 1;
        }
        if ((ustatus->UTXEN) == 0)
        {
           ustatus->UTXEN = 1;
        }
        for (i = len; i; --i)
        {
           while ((ustatus->TRMT) ==0);
           *txreg = *(char*)buffer++;
        }
        break;
     case handle_can1:  /* code to support can1 */
        break;
     case handle_can2:  /* code to support can2 */
        break;
     case handle_spi1:  /* code to support spi1 */
        break;
     case handle_spi2:  /* code to support spi2 */
        break;
     }
     return(len);
}
```

where you would fill in the appropriate code as specified in the comments.

Now you can use the generic C stdio I/O features to write to another port:

```
FILE *can1 = fopen("c","w");
     fprintf(can1,"This will be output through the can\n");
```

5.18.1.4 Customizing Helper Functions for XC32

The standard I/O relies on helper functions described in 6. Syscall Interface. These functions include `read()`, `write()`, `open()`, and `close()` which are called to read, write, open or close handles that are associated with standard I/O `FILE` pointers. The sources for these libraries are provided for you to customize as you wish.

The simplest way to redirect standard I/O to the peripheral of your choice is to select one of the default handles already in use. Also, you could open files with a specific name via `fopen()` by rewriting `open()` to return a new handle to be recognized by `read()` or `write()` as appropriate.

If only a specific peripheral is required, then you could associate handle `1 == stdout` or `2 == stderr` to another peripheral by writing the correct code to talk to the interested peripheral.

A complete generic solution might be:

```
/* should be in a header file */
enum my_handles {
    handle_stdin,
    handle_stdout,
    handle_stderr,
    handle_peripheral1,
    handle_peripheral2,
};

int
open(const char *name, int access, int mode) {
    switch (name[0]) {
        case 'i': return handle_stdin;
        case 'o': return handle_stdout;
        case 'e': return handle_stderr;
        case 'c': return handle_peripheral1;
        case 'C': return handle_peripheral2;
        default: return handle_stderr;
    }
}
```

Single letters were used in this example because they are faster to check and use less memory. However, if memory is not an issue, you could use `strcmp` to compare full names.

In `write()`, you would write:

```
int write(int handle, void *buffer, unsigned int len) {
    int i;

    /* Do not try to output an empty string */
    if (!buffer || (len == 0))
        return 0;

    switch (handle) {
        case 0:
        case 1:
        case 2:
            for (i = len; i; --i) {
                /* place code here to write the next byte of buffer to the desired peripheral
*/
                PERIPH1 = *(char*) buffer;
                buffer++;
            }
            break;
        case handle_peripheral1:
            for (i = len; i; --i) {
                /* place code here to write the next byte of buffer to the desired peripheral
*/
                PERIPH2 = *(char*) buffer;
                buffer++;
            }
            break;
        case handle_peripheral2:
            /* place code here to write the buffer to the desired peripheral */
            break;

        default:
        {
            break;
        }
    }
    return (len);
}
```

where you would fill in the appropriate code as specified in the comments.

Now you can use the generic C standard I/O features to write to another port:

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

int main(void) {
    char my_string[] = "count";
    int count = 0;

    for (count = 0; count < 5; count++) {
        printf("%s: %d\n", my_string, count);
    }

    FILE * peripheral = fopen("c", "w");
    fprintf(peripheral, "This will be output through the peripheral\n");

    __builtin_software_breakpoint();
    while (1);
}
```

5.18.2 size_t Type

An unsigned integer type used by the result of the `sizeof` operator

Include

`<stddef.h>`

`<stdio.h>`

`<stdlib.h>`

`<string.h>`

`<time.h>`

`<wchar.h>`

Definition

```
typedef unsigned size_t;
```

5.18.3 stdio Types

5.18.3.1 FILE

An object type capable of recording information necessary to control a stream, including its file position indicator, a buffer pointer, a read/write error indicator, and an end-of-file indicator.

Attention: This type is not implemented by MPLAB XC8 for PIC.

Include

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

5.18.3.2 fpos_t

An object type that can hold the necessary information to specify uniquely every position within a file.

Attention: This type is not implemented by MPLAB XC8 C compilers.

Include

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

5.18.4 Standard Stream File Pointers

These pointers specify I/O in the standard stream.

5.18.4.1 stderr

File pointer to the standard error stream.

Include

`<stdio.h>`

5.18.4.2 stdin

File pointer to the standard input stream.

Include

`<stdio.h>`

5.18.4.3 stdout

File pointer to the standard output stream.

Include

`<stdio.h>`

5.18.5 stdio Macros

5.18.5.1 BUFSIZ

An integer constant expression that is the size of the buffer used by the `setbuf()` function

Attention: This macro is implemented by all compilers, but with MPLAB XC8, its value has no significance apropos its intended purpose.

Include

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

5.18.5.2 EOF

An integer constant expression, with type `int` and a negative value, that is returned by several file-access functions to indicate end-of-file, that is, no more input from a stream.

Include

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

5.18.5.3 FILENAME_MAX

An integer constant expression that when used as the size of a `char` array will ensure that that array can hold the longest file name string that can be opened.

Attention: This macro is implemented by all compilers, but with MPLAB XC8, its value has no significance apropos its intended purpose.

Include

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

5.18.5.4 FOPEN_MAX

An integer constant expression that is the minimum number of files that can be open simultaneously.

Attention: This macro is implemented by all compilers, but with MPLAB XC8, its value has no significance apropos its intended purpose.

Include

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

5.18.6 setvbuf/setbuf Macros

These macros are used by the function `setvbuf`.

5.18.6.1 _IOFBF

Indicates full buffering.

Include

```
<stdio.h>
```

5.18.6.2 _IOLBF

Indicates line buffering.

Include

```
<stdio.h>
```

5.18.6.3 _IONBF

Indicates no buffering.

Include

```
<stdio.h>
```

5.18.6.4 BUFSIZ

An integer constant expression defining the size of the buffer used by `setbuf`.

Include

```
<stdio.h>
```

5.18.7 tmpnam Macros

These macros are used by the function `tmpnam`.

5.18.7.1 L_tmpnam

Defines the number of characters for the longest temporary file name created by the function `tmpnam`.

Include

```
<stdio.h>
```

Value

20

Remarks

`L_tmpnam` is used to define the size of the array used by `tmpnam`.

5.18.7.2 TMP_MAX

The maximum number of unique file names the function `tmpnam` can generate.

Include

```
<stdio.h>
```

Value

10000

5.18.8 fseek Macros

These macros are used by the function `fseek`.

5.18.8.1 SEEK_CUR

Indicates that `fseek` should seek from the current position of the file pointer.

Include

`<stdio.h>`

Example

See example for `fseek`.

5.18.8.2 SEEK_END

Indicates that `fseek` should seek from the end of the file.

Include

`<stdio.h>`

Example

See example for `fseek`.

5.18.8.3 SEEK_SET

Indicates that `fseek` should seek from the beginning of the file.

Include

`<stdio.h>`

Example

See example for `fseek`.

5.18.9 NULL Macro

A constant value which represent a null pointer constant. It's value and type is implementation defined.

Include

`<locale.h>`

`<stddef.h>`

`<stdio.h>`

`<stdlib.h>`

`<string.h>`

`<time.h>`

`<wchar.h>`

Definition

```
#define NULL ((void*)0)
```

5.18.10 clearerr Function

Resets the error indicator for the stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

<stdio.h>

Prototype

```
void clearerr(FILE * stream);
```

Argument

stream	stream to reset error indicators

Remarks

The function clears the end-of-file and error indicators for the given stream (i.e., feof and ferror will return false after the function clearerr is called).

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

```
/* This program tries to write to a file that is */
/* readonly. This causes the error indicator to  */
/* be set.  The function ferror is used to check */
/* the error indicator. The function clearerr is */
/* used to reset the error indicator so the next */
/* time ferror is called it will not report an   */
/* error.                                        */
#include <stdio.h>

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("sampclearerr.c", "r")) == NULL)
    printf("Cannot open file\n");
  else
  {
    fprintf(myfile, "Write this line to the file.\n");
    if (ferror(myfile))
      printf("Error\n");
    else
      printf("No error\n");
    clearerr(myfile);
    if (ferror(myfile))
      printf("Still has Error\n");
    else
      printf("Error indicator reset\n");

    fclose(myfile);
  }
}
```

Example Output

```
Error
Error indicator reset
```

5.18.11 fclose Function

Close a stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

<stdio.h>

Prototype

```
int fclose(FILE * stream);
```

Argument

stream	pointer to the stream to close

Return Value

Returns 0 if successful; otherwise, returns `EOF` if any errors were detected.

Remarks

The `fclose` function causes the stream indicated by the argument to be flushed and the associated file to be closed. This function requires a heap.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <stdio.h>

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");

    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }
}
```

Example Output

```
afile1 was opened
afile1 was closed
```

5.18.12 feof Function

Tests for end-of-file.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

```
<stdio.h>
```

Prototype

```
int feof(FILE * stream);
```

Argument

stream	stream to check for end-of-file

Return Value

Returns non-zero if stream end-of-file indicator is set; otherwise, returns zero.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <stdio.h>

int main(void)
{
  FILE * myfile;
  int y = 0;

  if( (myfile = fopen( "afile.txt", "rb" )) == NULL )
    printf( "Cannot open file\n" );
  else
  {
    for (;;)
    {
      y = fgetc(myfile);
      if (feof(myfile))
        break;
      fputc(y, stdout);
    }
    fclose( myfile );
  }
}
```

Example Input

Contents of `afile.txt` (used as input):

```
This is a sentence.
```

Example Output

```
This is a sentence.
```

5.18.13 ferror Function

Tests if error indicator is set.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int ferror(FILE * stream);
```

Argument

`stream`	pointer to `FILE` structure

Return Value

Returns a non-zero value if error indicator is set; otherwise, returns a zero.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
/* This program tries to write to a file that is */
/* readonly. This causes the error indicator to  */
/* be set.  The function ferror is used to check */
/* the error indicator and find the error. The   */
/* function clearerr is used to reset the error  */
/* indicator so the next time ferror is called   */
/* it will not report an error.                  */

#include <stdio.h>

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("sampclearerr.c", "r")) == NULL)
    printf("Cannot open file\n");
  else
  {
    fprintf(myfile, "Write this line to the file.\n");
    if (ferror(myfile))
      printf("Error\n");
    else
      printf("No error\n");
    clearerr(myfile);
    if (ferror(myfile))
      printf("Still has Error\n");
    else
      printf("Error indicator reset\n");

    fclose(myfile);
  }
}
```

Example Output

```
Error
Error indicator reset
```

5.18.14 fflush Function

Flushes the buffer in the specified stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int fflush(FILE * stream);
```

Argument

`stream`	pointer to the stream to flush

Return Value

Returns `EOF` if a write error occurs; otherwise, returns zero for success.

Remarks

If stream is a null pointer, all output buffers are written to files. The `fflush` function has no effect on an unbuffered stream.

5.18.15 fgetc Function

Get a character from a stream

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int fgetc(FILE * stream);
```

Argument

stream	pointer to the open stream

Return Value

Returns the character read or `EOF` if a read error occurs or end-of-file is reached.

Remarks

The function reads the next character from the input stream, advances the file-position indicator and returns the character as an `unsigned char` converted to an `int`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <stdio.h>

int main(void)
{
  FILE *buf;
  char y;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = fgetc(buf);
    while (y != EOF)
    {
      printf("%c|", y);
      y = fgetc(buf);
    }
    fclose(buf);
  }
}
```

Example Input

Contents of `afile.txt` (used as input):

```
Short
Longer string
```

Example Output

```
S|h|o|r|t|
|L|o|n|g|e|r| |s|t|r|i|n|g|
|
```

5.18.16 fgetpos Function

Gets the stream's file position.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
int fgetpos(FILE * stream, fpos_t * pos);
```

Arguments

stream	target stream
pos	position-indicator storage

Return Value

Returns 0 if successful; otherwise, returns a non-zero value.

Remarks

The function stores the file-position indicator for the given stream into the object pointed to by `pos` if successful; otherwise, `fgetpos` sets `errno`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
/* This program opens a file and reads bytes at */
/* several different locations. The fgetpos     */
/* function notes the 8th byte. 21 bytes are    */
/* read then 18 bytes are read. Next the        */
/* fsetpos function is set based on the         */
/* fgetpos position and the previous 21 bytes   */
/* are reread.                                  */

#include <stdio.h>

int main(void)
{
  FILE   *myfile;
  fpos_t pos;
  char   buf[25];

  if ((myfile = fopen("sampfgetpos.c", "rb")) == NULL)
    printf("Cannot open file\n");
  else
  {
   fread(buf, sizeof(char), 8, myfile);
    if (fgetpos(myfile, &pos) != 0)
      perror("fgetpos error");
    else
    {
      fread(buf, sizeof(char), 21, myfile);
      printf("Bytes read: %.21s\n", buf);
      fread(buf, sizeof(char), 18, myfile);
      printf("Bytes read: %.18s\n", buf);
    }

  if (fsetpos(myfile, &pos) != 0)
    perror("fsetpos error");

  fread(buf, sizeof(char), 21, myfile);
```

```
  printf("Bytes read: %.21s\n", buf);
  fclose(myfile);
  }
}
```

Example Output

```
Bytes read: program opens a file
Bytes read: and reads bytes at
Bytes read: program opens a file
```

5.18.17 fgets Function

Get a string from a stream.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
char * fgets(char * s, int n, FILE * stream);
```

Arguments

s	pointer to the storage string
n	maximum number of characters to read
stream	pointer to the open stream

Return Value

Returns a pointer to the string `s` if successful; otherwise, returns a null pointer.

Remarks

The function reads characters from the input stream and stores them into the string pointed to by `s` until it has read `n`-1 characters, stores a newline character or sets the end-of-file or error indicators. If any characters were stored, a null character is stored immediately after the last read character in the next element of the array. If `fgets` sets the error indicator, the array contents are indeterminate.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <stdio.h>

#define MAX 50

int main(void)
{
  FILE *buf;
  char s[MAX];

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    while (fgets(s, MAX, buf) != NULL)
      printf("%s|", s);
    fclose(buf);
```

```
  }
}
```

Example Input

Contents of `afile.txt` (used as input):

```
Short
Longer string
```

Example Output

```
Short
|Longer string
|
```

5.18.18 fopen Function

Opens a file.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
FILE * fopen(const char * filename, const char * mode);
```

Arguments

filename	name of the file
mode	type of access permitted

Return Value

Returns a pointer to the open stream. If the function fails, a null pointer is returned.

Remarks

The following are types of file access:

r	Opens an existing text file for reading.
w	Opens an empty text file for writing (an existing file is overwritten).
a	Opens a text file for appending (a file is created if it doesn't exist).
rb	Opens an existing binary file for reading.
wb	Opens an empty binary file for writing (an existing file is overwritten).
ab	Opens a binary file for appending (a file is created if it doesn't exist).
r+	Opens an existing text file for reading and writing.
w+	Opens an empty text file for reading and writing (an existing file is overwritten).
a+	Opens a text file for reading and appending (a file is created if it doesn't exist).
r+b or rb+	Opens an existing binary file for reading and writing.
w+b or wb+	Opens an empty binary file for reading and writing (an existing file is overwritten.)

a+b or ab+	Opens a binary file for reading and appending (a file is created if it doesn't exist).

This function requires a heap.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

```
#include <stdio.h>

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;
  if ((myfile1 = fopen("afile1", "r")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");
    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }
  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Second try, cannot open afile1\n");
  else
  {
    printf("Second try, afile1 was opened\n");
    y = fclose(myfile1);
    if (y == EOF)
      printf("afile1 was not closed\n");
    else
      printf("afile1 was closed\n");
  }
  if ((myfile2 = fopen("afile2", "a+")) == NULL)
    printf("Cannot open afile2\n");
  else
  {
    printf("afile2 was opened\n");
    y = fclose(myfile2);
    if (y == EOF)
      printf("afile2 was not closed\n");
    else
      printf("afile2 was closed\n");
  }
}
```

Example Output

```
Cannot open afile1
Second try, afile1 was opened
afile1 was closed
afile2 was opened
afile2 was closed
```

Example Explanation

afile1 must exist before it can be opened for reading (r) or the fopen function will fail.

If the fopen function opens a file for writing (w+) it does not have to exist, it will be created and then opened. If the file does exist, it cannot be overwritten and will be appended.

5.18.19 fprintf Function

Prints formatted text to a stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

```
<stdio.h>
```

Prototype

```
int fprintf(FILE * stream, const char * format, ...);
```

Arguments

stream	pointer to the stream in which to output data
format	format control string
...	optional arguments; see “Remarks”

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

This function can print to any stream.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[`*`flags`*`][`*`width`*`][.`*`precision`*`][`*`length`*`]`*`specifier`*

The *`flags`* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first character of a conversion result does not include a sign (+ or -) or if a signed conversion results in no characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag - has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
hh	When used with the n conversion specifier, indicates that the pointer argument points to a signed char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.

.....continued

Modifier	Meaning
L	When used with the a, A, e, E, f, F, g, G conversion specifiers, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a char type and printed as a single character. When the l modifier is present, the wint_t argument is converted to multibyte characters then printed.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
f, F	double	Converted to decimal notation using the general form [−]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of characters written so far is written to the object pointed to by the pointer. No characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (not necessarily an address) held by the pointer.
s	char array, or wchar_t array	A string, whose characters are written up to but not including the terminating null character. When the l modifier is present, wide characters from the array are converted to multibyte characters up to and including a terminating null wide character, which are then written up to but not including the terminating null character.

.....continued		
Specifier	**Argument value type**	**Printing notes**
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % character is printed.

Example

```
#include <stdio.h>

int main(void)
{
  FILE *myfile;
  int y;
  char s[]="Print this string";
  int x = 1;
  char a = '\n';
  if ((myfile = fopen("afile", "w")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    y = fprintf(myfile, "%s %d time%c", s, x, a);

    printf("Number of characters printed to file = %d",y);

    fclose(myfile);
  }
}
```

Example Output

```
Number of characters printed to file = 25
```

Contents of afile:

```
Print this string 1 time
```

Related Links

5.18.33 printf Function

5.18.20 fputc Function

Puts a character to the stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

```
<stdio.h>
```

Prototype

```
int fputc(int c, FILE * stream);
```

Arguments

c	character to be written
stream	pointer to the open stream

Return Value

Returns the character written or `EOF` if a write error occurs.

Remarks

The function writes the character to the output stream, advances the file-position indicator and returns the character as an `unsigned char` converted to an `int`.

Example

```
#include <stdio.h>

int main(void)
{
  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;

  for (y = buf; (x != EOF) && (*y != '\0'); y++)
  {
    x = fputc(*y, stdout);
    fputc('|', stdout);
  }
}
```

Example Output

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

5.18.21 fputs Function

Puts a string to the stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int fputs(const char * restrict s, FILE * restrict stream);
```

Arguments

s	string to be written
stream	pointer to the open stream

Return Value

Returns a non-negative value if successful; otherwise, returns `EOF`.

Remarks

The function writes characters to the output stream up to but not including the null character.

Example

```
#include <stdio.h>

int main(void)
{
  char buf[] = "This is text\n";

  fputs(buf,stdout);
  fputs("|",stdout);
}
```

Example Output

```
This is text
|
```

5.18.22 fread Function

Reads data from the stream.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
size_t fread(void * restrict ptr, size_t size, size_t nelem, FILE * restrict stream);
```

Arguments

ptr	pointer to the storage buffer
size	size of item
nelem	maximum number of items to be read
stream	pointer to the stream

Return Value

Returns the number of complete elements read up to `nelem` whose size is specified by `size`.

Remarks

The function reads characters from a given stream into the buffer pointed to by `ptr` until the function stores size * `nelem` characters or sets the end-of-file or error indicator. `fread` returns n/size where n is the number of characters it read. If n is not a multiple of size, the value of the last element is indeterminate. If the function sets the error indicator, the file-position indicator is indeterminate.

Example

```
#include <stdio.h>

int main(void) {
  FILE *buf;
  int x, numwrote, numread;
  double nums[10], readnums[10];

  if ((buf = fopen("afile.out", "w+")) != NULL) {
    for (x = 0; x < 10; x++) {
      nums[x] = 10.0/(x + 1);
      printf("10.0/%d = %f\n", x+1, nums[x]);
    }
    numwrote = fwrite(nums, sizeof(double), 10, buf);
```

```
    printf("Wrote %d numbers\n\n", numwrote);
    fclose(buf);
    }
  else printf("Cannot open afile.out\n");

  if ((buf = fopen("afile.out", "r+")) != NULL) {
    numread = fread(readnums, sizeof(double), 10, buf);
    printf("Read %d numbers\n", numread);
    for (x = 0; x < 10; x++) {
      printf("%d * %f = %f\n", x+1, readnums[x], (x + 1) * readnums[x]);
    }
    fclose(buf);
  }
  else printf("Cannot open afile.out\n");
}
```

Example Output

```
10.0/1 = 10.000000
10.0/2 = 5.000000
10.0/3 = 3.333333
10.0/4 = 2.500000
10.0/5 = 2.000000
10.0/6 = 1.666667
10.0/7 = 1.428571
10.0/8 = 1.250000
10.0/9 = 1.111111
10.0/10 = 1.000000
Wrote 10 numbers

Read 10 numbers
1 * 10.000000 = 10.000000
2 * 5.000000 = 10.000000
3 * 3.333333 = 10.000000
4 * 2.500000 = 10.000000
5 * 2.000000 = 10.000000
6 * 1.666667 = 10.000000
7 * 1.428571 = 10.000000
8 * 1.250000 = 10.000000
9 * 1.111111 = 10.000000
10 * 1.000000 = 10.000000
```

Example Explanation

This program uses `fwrite` to save 10 numbers to a file in binary form. This allows the numbers to be saved in the same pattern of bits as the program is using, which provides more accuracy and consistency. Using `fprintf` would save the numbers as text strings, which could cause the numbers to be truncated. Each number is divided into 10 to produce a variety of numbers. Retrieving the numbers with `fread` to a new array and multiplying them by the original number shows the numbers were not truncated in the save process.

5.18.23 freopen Function

Reassigns an existing stream to a new file.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
FILE *freopen(const char * restrict filename, const char * restrict mode, FILE *
restrict stream);
```

Arguments

filename	name of the new file
mode	type of access permitted
stream	pointer to the currently open stream

Return Value

Returns a pointer to the new open file. If the function fails a null pointer is returned.

Remarks

The function closes the file associated with the stream as though `fclose` was called. Then it opens the new file as though `fopen` was called. `freopen` will fail if the specified stream is not open. See `fopen` for the possible types of file access.

This function requires a heap.

Example

```
#include <stdio.h>

int main(void)
{
  FILE *myfile1, *myfile2;
  int y;

  if ((myfile1 = fopen("afile1", "w+")) == NULL)
    printf("Cannot open afile1\n");
  else
  {
    printf("afile1 was opened\n");

    if ((myfile2 = freopen("afile2", "w+",
         myfile1)) == NULL)
    {
      printf("Cannot open afile2\n");
      fclose(myfile1);
    }
    else
    {
      printf("afile2 was opened\n");
      fclose(myfile2);
    }
  }
}
```

Example Output

```
afile1 was opened
afile2 was opened
```

Example Explanation

This program uses `myfile2` to point to the stream when `freopen` is called, so if an error occurs, `myfile1` will still point to the stream and can be closed properly. If the `freopen` call is successful, `myfile2` can be used to close the stream properly.

5.18.24 fscanf Function

Scans formatted text from a stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int fscanf(FILE * restrict stream, const char * restrict format, ...);
```

Arguments

stream	pointer to the open stream from which to read data
format	format control string
...	optional arguments

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. `EOF` is returned if end-of-file is encountered before the first conversion or if an error occurs.

Remarks

This function can read input from any stream.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string can be composed of white space characters, which reads input up to the first non-white-space character in the input; or ordinary multibyte characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `short int` or `unsigned short int` object referenced by the pointer argument.
hh	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `signed char` or `unsigned char` object referenced by the pointer argument.
l	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `long int` or `unsigned long int` object referenced by the pointer argument.
l	When used the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` indicates that this conversion specifier assigns to a `double` object referenced by the pointer argument.
l	When used the `c`, `s`, or `[` indicates that this conversion specifier assigns to a `wchar_t` object referenced by the pointer argument.
ll	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `long long int` or `unsigned long long int` object referenced by the pointer argument.
L	When used the `a`, `A`, `e`, `E`, `f`, `F`, `g`, `G` conversion specifier, indicates the argument value is a `long double`.

.....continued	
Modifier	**Meaning**
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single character or the number of characters specified by the field width if present. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character as if by a call to the mbrtowc function. No null character or wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 0 for the base argument.
n	int	No input is consumed but the number of characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.

.....continued

Specifier	Receiving object type	Matches
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space characters, which is written to the array argument and appended with a null terminating character. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character and terminated with a null wide character.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 16 for the base argument.
%		Matches a single % character. No assignment takes place.
[	char array, or wchar_t array	Matches all the characters in the input that have been specified between the [and trailing] character, unless the character after the opening bracket is a circumflex, ^, in which case a match is only made with characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket character will match the input and the next following right bracket character is the matching right bracket that ends the specification. A null character will be appended to the characters read. If an l length modifier is present, the input shall be a sequence of multibyte characters and a null wide character will be appended to the matched wide characters.

Example

```
#include <stdio.h>
int main(void)
{
  FILE *myfile;
  char s[30];
  int x;
  char a;
  if ((myfile = fopen("afile", "w+")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    fprintf(myfile, "%s %d times%c", "Print this string", 100, '\n');

    fseek(myfile, 0L, SEEK_SET);

    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%d", &x);
    printf("%d\n", x);
    fscanf(myfile, "%s", s);
    printf("%s\n", s);
    fscanf(myfile, "%c", a);
    printf("%c\n", a);

    fclose(myfile);
```

```
  }
}
```

Example Input

Contents of `afile`:

```
Print this string 100 times
```

Example Output

```
Print
this
string
100
times
```

5.18.25 fseek Function

Moves file pointer to a specific location.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
int fseek(FILE * restrict stream, long offset, int mode);
```

Arguments

stream	stream in which to move the file pointer
offset	value to add to the current position
mode	type of seek to perform

Return Value

Returns 0 if successful; otherwise, returns a non-zero value and set `errno`.

Remarks

`mode` can be one of the following tabulated values.

Mode	Description
SEEK_SET	seeks from the beginning of the file
SEEK_CUR	seeks from the current position of the file pointer
SEEK_END	seeks from the end of the file

Example

```
#include <stdio.h>

int main(void)
{
  FILE *myfile;
  char s[70];
  int  y;
  myfile = fopen("afile.out", "w+");
  if (myfile == NULL)
```

```
    printf("Cannot open afile.out\n");
  else
  {
    fprintf(myfile, "This is the beginning, this is the middle and this is the end.");

    y = fseek(myfile, 0L, SEEK_SET);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 22, myfile);
      printf("\"%s\"\n\n", s);
    }

    y = fseek(myfile, 2L, SEEK_CUR);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 70, myfile);
      printf("\"%s\"\n\n", s);
    }

    y = fseek(myfile, -16L, SEEK_END);
    if (y)
      perror("Fseek failed");
    else
    {
      fgets(s, 70, myfile);
      printf("\"%s\"\n", s);
    }
    fclose(myfile);
  }
}
```

Example Output

```
"This is the beginning"

"this is the middle and this is the end."

"this is the end."
```

Example Explanation

The file `afile.out` is created with the text, “This is the beginning, this is the middle and this is the end.”

The function `fseek` uses an offset of zero and `SEEK_SET` to set the file pointer to the beginning of the file. `fgets` then reads 22 characters which are “This is the beginning,” and adds a null character to the string.

Next, `fseek` uses an offset of two and `SEEK_CURRENT` to set the file pointer to the current position plus two (skipping the comma and space). `fgets` then reads up to the next 70 characters. The first 39 characters are “this is the middle and this is the end.” It stops when it reads `EOF` and adds a null character to the string.

Finally, `fseek` uses an offset of negative 16 characters and `SEEK_END` to set the file pointer to 16 characters from the end of the file. `fgets` then reads up to 70 characters. It stops at the `EOF` after reading 16 characters “this is the end,” and adds a null character to the string.

5.18.26 fsetpos Function

Sets the stream’s file position.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
int fsetpos(FILE * stream, const fpos_t * pos);
```

Arguments

stream	target stream
pos	position-indicator storage as returned by an earlier call to `fgetpos`

Return Value

Returns 0 if successful; otherwise, returns a non-zero value.

Remarks

The function sets the file-position indicator for the given stream in *`pos` if successful; otherwise, `fsetpos` sets `errno`.

Example

```
/* This program opens a file and reads bytes at */
/* several different locations. The fgetpos     */
/* function notes the 8th byte. 21 bytes are   */
/* read then 18 bytes are read. Next, the        */
/* fsetpos function is set based on the         */
/* fgetpos position and the previous 21 bytes   */
/* are reread.                                  */

#include <stdio.h>

int main(void)
{
  FILE   * myfile;
  fpos_t pos;
  char   buf[25];
  if ((myfile = fopen("sampfgetpos.c", "rb")) == NULL)
    printf("Cannot open file\n");
  else
  {
    fread(buf, sizeof(char), 8, myfile);
    if (fgetpos(myfile, &pos) != 0)
      perror("fgetpos error");
    else
    {
      fread(buf, sizeof(char), 21, myfile);
      printf("Bytes read: %.21s\n", buf);
      fread(buf, sizeof(char), 18, myfile);
      printf("Bytes read: %.18s\n", buf);
    }
  if (fsetpos(myfile, &pos) != 0)
    perror("fsetpos error");

  fread(buf, sizeof(char), 21, myfile);
  printf("Bytes read: %.21s\n", buf);
  fclose(myfile);
  }
}
```

Example Output

```
Bytes read: program opens a file
Bytes read: and reads bytes at
Bytes read: program opens a file
```

5.18.27 ftell Function

Gets the current position of a file pointer.

Attention: This function is not implemented by MPLAB XC8.

Include

<stdio.h>

Prototype

```
long ftell(FILE * stream);
```

Argument

stream	stream in which to get the current file position

Return Value

Returns the position of the file pointer if successful; otherwise, returns -1.

Example

```
#include <stdio.h>

int main(void)
{
  FILE *myfile;
  char s[75];
  long y;
  myfile = fopen("afile.out", "w+");
  if (myfile == NULL)
    printf("Cannot open afile.out\n");
  else
  {
    fprintf(myfile,"This is a very long sentence "
                   "for input into the file named afile.out for testing.");

    fclose(myfile);
    if ((myfile = fopen("afile.out", "rb")) != NULL)
    {
      printf("Read some characters:\n");
      fread(s, sizeof(char), 29, myfile);
      printf("\t\"%s\"\n", s);

      y = ftell(myfile);
      printf("The current position of the file pointer is %ld\n", y);
      fclose(myfile);
    }
  }
}
```

Example Output

```
Read some characters:
        "This is a very long sentence "
The current position of the file pointer is 29
```

5.18.28 fwrite Function

Writes data to the stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

<stdio.h>

Prototype

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nelem, FILE * restrict
stream);
```

Arguments

ptr	pointer to the storage buffer
size	size of item
nelem	maximum number of items to be read
stream	pointer to the open stream

Return Value

Returns the number of complete elements successfully written, which will be less than `nelem` only if a write error is encountered.

Remarks

The function writes characters to a given stream from a buffer pointed to by `ptr` up to `nelem` elements whose size is specified by size. The file position indicator is advanced by the number of characters successfully written. If the function sets the error indicator, the file-position indicator is indeterminate.

Example

```
#include <stdio.h>

int main(void)
{
  FILE *buf;
  int x, numwrote, numread;
  double nums[10], readnums[10];
  if ((buf = fopen("afile.out", "w+")) != NULL)
  {
    for (x = 0; x < 10; x++)
    {
      nums[x] = 10.0/(x + 1);
      printf("10.0/%d = %f\n", x+1, nums[x]);
    }

    numwrote = fwrite(nums, sizeof(double), 10, buf);
    printf("Wrote %d numbers\n\n", numwrote);
    fclose(buf);

  }
  else
    printf("Cannot open afile.out\n");
  if ((buf = fopen("afile.out", "r+")) != NULL)
  {
    numread = fread(readnums, sizeof(double), 10, buf);
    printf("Read %d numbers\n", numread);
    for (x = 0; x < 10; x++)
    {
      printf("%d * %f = %f\n", x+1, readnums[x],
            (x + 1) * readnums[x]);
    }
    fclose(buf);
  }
  else
    printf("Cannot open afile.out\n");
}
```

Example Output

```
10.0/1 = 10.000000
10.0/2 = 5.000000
10.0/3 = 3.333333
10.0/4 = 2.500000
```

```
10.0/5 = 2.000000
10.0/6 = 1.666667
10.0/7 = 1.428571
10.0/8 = 1.250000
10.0/9 = 1.111111
10.0/10 = 1.000000
Wrote 10 numbers

Read 10 numbers
1 * 10.000000 = 10.000000
2 * 5.000000 = 10.000000
3 * 3.333333 = 10.000000
4 * 2.500000 = 10.000000
5 * 2.000000 = 10.000000
6 * 1.666667 = 10.000000
7 * 1.428571 = 10.000000
8 * 1.250000 = 10.000000
9 * 1.111111 = 10.000000
10 * 1.000000 = 10.000000
```

Example Explanation

This program uses `fwrite` to save 10 numbers to a file in binary form. This allows the numbers to be saved in the same pattern of bits as the program is using which provides more accuracy and consistency. Using `fprintf` would save the numbers as text strings, which could cause the numbers to be truncated. Each number is divided into 10 to produce a variety of numbers. Retrieving the numbers with `fread` to a new array and multiplying them by the original number shows the numbers were not truncated in the save process.

5.18.29 getc Function

Get a character from the stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int getc(FILE * stream);
```

Argument

stream	pointer to the open stream

Return Value

Returns the character read or `EOF` if a read error occurs or end-of-file is reached.

Remarks

The `getc` function performs the same task as the function `fgetc`.

Example

```
#include <stdio.h>

int main(void)
{
  FILE *buf;
  char y;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = getc(buf);
```

```
    while (y != EOF)
    {
      printf("%c|", y);
      y = getc(buf);
    }
    fclose(buf);
  }
}
```

Example Input

Contents of `afile.txt` (used as input):

```
Short
Longer string
```

Example Output

```
S|h|o|r|t|
|L|o|n|g|e|r| |s|t|r|i|n|g|
|
```

5.18.30 getchar Function

Get a character from `stdin`.

Include

`<stdio.h>`

Prototype

```
int getchar(void);
```

Return Value

Returns the character read or `EOF` if a read error occurs or end-of-file is reached.

Remarks

This function is equivalent to `fgetc` with the argument `stdin`.

When building with the MPLAB XC8 compiler for PIC MCUs, the `getchar()` function relies on the `getch()` function being properly defined to obtain input from the required peripheral or location. Reading will not work as expected until the `getch()` stub (found in the `pic/sources/c99/common/getch.c` of your compiler distribution) is completed. See 4.1 Example code for 8-bit PIC MCUs for information on defining `getch()` so that text can be read from a file in the MPLAB X IDE simulator.

Example

```
#include <stdio.h>

int main(void)
{
  char y;

    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
    y = getchar();
    printf("%c|", y);
}
```

Example Input

Contents of `UartIn.txt` (used as `stdin` input for simulator):

```
"Short
Longer string"
```

Example Output

```
S|h|o|r|t|
```

5.18.31 gets Function

Get a string from `stdin`.

Include

`<stdio.h>`

Prototype

```
char * gets(char * s);
```

Argument

s	pointer to the storage string

Return Value

Returns a pointer to the string s if successful; otherwise, returns a null pointer.

Remarks

The `gets()` function reads characters from the stream `stdin` and stores them into the string pointed to by `s` until it reads a newline character (which is not stored) or sets the end-of-file or error indicators. If any characters were read, a nul character is stored immediately after the last read character in the next element of the array. If `gets()` sets the error indicator, the array contents are indeterminate.

When building with the MPLAB XC8 compiler for PIC MCUs, the `gets()` function relies on the `getch()` function being properly defined to obtain input from the required peripheral or location. Reading will not work as expected until the `getch()` stub (found in the `pic/sources/c99/common/getch.c` of your compiler distribution) is completed. See 4.1 Example code for 8-bit PIC MCUs for information on defining `getch()` so that text can be read from a file in the MPLAB X IDE simulator.

Example

```
#include <stdio.h>

int main(void)
{
  char y[50];

  gets(y) ;
  printf("Text: %s\n", y);
}
```

Example Input

Contents of `UartIn.txt` (used as `stdin` input for simulator):

```
"Short
Longer string"
```

Example Output

```
Text: Short
```

5.18.32 perror Function

Prints an error message to `stderr`.

Include

`<stdio.h>`

Prototype

```
void perror(const char * s);
```

Argument

s	string to print

Return Value

None.

Remarks

The string `s` is printed followed by a colon and a space. Then, an error message based on `errno` is printed followed by an newline.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  double x,y;

  errno = 0;
  x = -2.0;
  y = acos (x);
  if (errno)
    perror("Error");
  printf("The arccosine of %f is %f\n", x, y);
}
```

Example Output

```
Error: Domain error
The arccosine of -2.000000 is nan
```

5.18.33 printf Function

Prints formatted text to `stdout`.

Include

`<stdio.h>`

Prototype

```
int printf(const char * format, ...);
```

Arguments

format	format control string
...	optional arguments; see “Remarks”

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

When building with the MPLAB XC8 compiler for PIC MCUs, the `printf()` function relies on the `putch()` function being properly defined to direct output to the required peripheral or location. Printing will not work as expected until the `putch()` stub (found in the `pic/sources/c99/common/putch.c` of your compiler distribution) is completed. See 4.1 Example code for 8-bit PIC MCUs for information on defining `putch()` so that printed text appears in an MPLAB X IDE simulator window.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[`*`flags`*`][`*`width`*`][.`*`precision`*`][`*`length`*`]`*`specifier`*

The *`flags`* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
`-`	Left justify the conversion result within a given field width.
`0`	Use 0 for the pad character instead of space (which is the default) for `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, except when converting an infinity or NaN. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and`X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.
`+`	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first character of a conversion result does not include a sign (`+` or `-`) or if a signed conversion results in no characters. If the *space* and `+` flags both appear, the *space* flag is ignored.
`#`	Convert the result to an alternative form. Specifically, for `o` conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For `x` (or `X`) conversions, `0x` (or `0X`) is prefixed to nonzero results. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. For `g` and `G` conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *`width`* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag `-` has been used.

If the asterisk, `*`, is used instead of a decimal number, the `int` argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *`precision`* field indicates:

- the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions
- the number of digits to appear after the decimal-point character for `a`, `A`, `e`, `E`, `f`, and `F` conversions
- the maximum number of significant digits for the `g` and `G` conversions
- the maximum number of bytes to be written for `s` conversions

The precision takes the form of a period, `.`, followed either by an asterisk, `*` or by an optional decimal integer. If neither the `*` or integer is specified, the precision is assumed to be zero. If the asterisk, `*`, is used instead of a decimal number, the `int` argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
hh	When used with the n conversion specifier, indicates that the pointer argument points to a signed char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used with the a, A, e, E, f, F, g, G conversion specifiers, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [-]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a char type and printed as a single character. When the l modifier is present, the wint_t argument is converted to multibyte characters then printed.
d	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [-]*d.ddde±dd*, where there is one digit before the decimal-point character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
f, F	double	Converted to decimal notation using the general form [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of characters written so far is written to the object pointed to by the pointer. No characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (not necessarily an address) held by the pointer.
s	char array, or wchar_t array	A string, whose characters are written up to but not including the terminating null character. When the l modifier is present, wide characters from the array are converted to multibyte characters up to and including a terminating null wide character, which are then written up to but not including the terminating null character.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % character is printed.

When used with MPAB XC16, use of this function requires a heap.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <stdio.h>

int main(void)
{
  /* print a character right justified in a 3  */
  /* character space.                          */
  printf("%3c\n", 'a');

  /* print an integer, left justified (as      */
  /* specified by the minus sign in the format */
  /* string) in a 4 character space. Print a   */
  /* second integer that is right justified in */
  /* a 4 character space using the pipe (|) as */
  /* a separator between the integers.         */
  printf("%-4d|%4d\n", -4, 4);

  /* print a number converted to octal in 4    */
  /* digits.                                   */
  printf("%.4o\n", 10);

  /* print a number converted to hexadecimal   */
  /* format with a 0x prefix.                  */
  printf("%#x\n", 28);

  /* print a float in scientific notation      */
  printf("%E\n", 1.1e20);

  /* print a float with 2 fraction digits      */
  printf("%.2f\n", -3.346);

  /* print a long float with %E, %e, or %f     */
  /* whichever is the shortest version         */
  printf("%Lg\n", .02L);
}
```

Example Output

```
  a
-4  |   4
0012
0x1c
1.100000E+20
-3.35
0.02
```

5.18.34 putc Function

Puts a character to the stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int putc(int c, FILE * stream);
```

Arguments

c	character to be written
stream	pointer to `FILE` structure

Return Value

Returns the character or `EOF` if an error occurs or end-of-file is reached.

Remarks

The `putc` function performs the same task as the function `fputc`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <stdio.h>

int main(void)
{
  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;

  for (y = buf; (x != EOF) && (*y != '\0'); y++)
  {
    x = putc(*y, stdout);
    putc('|', stdout);
  }
}
```

Example Output

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

5.18.35 putchar Function

Put a character to `stdout`.

Include

`<stdio.h>`

Prototype

```
int putchar(int c);
```

Argument

c	character to be written

Return Value

Returns the character or `EOF` if an error occurs or end-of-file is reached.

Remarks

Same effect as `fputc` with `stdout` as an argument.

When building with the MPLAB XC8 compiler for PIC MCUs, the `putchar()` function relies on the `putch()` function being properly defined to direct output to the required peripheral or location. Printing will not work as expected until the `putch()` stub (found in the `pic/sources/c99/common/putch.c` of your compiler distribution) is completed. See 4.1 Example code for 8-bit PIC MCUs for information on defining `putch()` so that printed text appears in an MPLAB X IDE simulator window.

Example

```
#include <stdio.h>

int main(void)
{
```

```
  char *y;
  char buf[] = "This is text\n";
  int x;

  x = 0;
  for (y = buf; (x != EOF) && (*y != '\0'); y++)
    x = putchar(*y);
}
```

Example Output

```
This is text
```

5.18.36 puts Function

Put a string to `stdout`.

Include

`<stdio.h>`

Prototype

```
int puts(const char * s);
```

Argument

s	string to be written

Return Value

Returns a non-negative value if successful; otherwise, returns `EOF`.

Remarks

The function writes characters to the stream `stdout`. A newline character is appended. The terminating null character is not written to the stream.

When building with the MPLAB XC8 compiler for PIC MCUs, the `puts()` function relies on the `putch()` function being properly defined to direct output to the required peripheral or location. Printing will not work as expected until the `putch()` stub (found in the `pic/sources/c99/common/putch.c` of your compiler distribution) is completed. See 4.1 Example code for 8-bit PIC MCUs for information on defining `putch()` so that printed text appears in an MPLAB X IDE simulator window.

Example

```
#include <stdio.h>

int main(void)
{
  char buf[] = "This is text\n";

  puts(buf);
  puts("|");
}
```

Example Output

```
This is text

|
```

5.18.37 remove Function

Deletes the specified file.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
int remove(const char * filename);
```

Argument

filename	name of file to be deleted

Return Value

Returns 0 if successful; otherwise, returns -1.

Remarks

If the file represented by *filename* does not exist or is open, remove will fail.

Example

```
#include <stdio.h>

int main(void)
{
  if (remove("myfile.txt") != 0)
    printf("Cannot remove file");
  else
    printf("File removed");
}
```

Example Output

```
File removed
```

5.18.38 rename Function

Renames the specified file.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
int rename(const char * old, const char * new);
```

Arguments

old	pointer to the old name
new	pointer to the new name

Return Value

Return 0 if successful; otherwise, returns a non-zero value.

Remarks

The old name *must* exist in the current working directory. The new name *must not* already exist in the current working directory.

Example

```
#include <stdio.h>

int main(void)
{
  if (rename("myfile.txt","newfile.txt") != 0)
    printf("Cannot rename file");
  else
    printf("File renamed");
}
```

Example Output

```
File renamed
```

5.18.39 rewind Function

Resets the file pointer to the beginning of the file.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

`void rewind(FILE * stream);`

Argument

stream	stream to reset the file pointer

Remarks

The function calls `fseek(stream, 0L, SEEK_SET)` and then clears the error indicator for the given stream.

Example

```
#include <stdio.h>

int main(void)
{
  FILE *myfile;
  char s[] = "cookies";
  int x = 10;

  if ((myfile = fopen("afile", "w+")) == NULL)
    printf("Cannot open afile\n");
  else
  {
    fprintf(myfile, "%d %s", x, s);
    printf("I have %d %s.\n", x, s);

    /* set pointer to beginning of file */
    rewind(myfile);
    fscanf(myfile, "%d %s", &x, &s);
    printf("I ate %d %s.\n", x, s);

    fclose(myfile);
  }
}
```

Example Output

```
I have 10 cookies.
I ate 10 cookies.
```

5.18.40 scanf Function

Scans formatted text from `stdin`.

Include

`<stdio.h>`

Prototype

```
int scanf(const char * restrict format, ...);
```

Arguments

`format`	format control string
`...`	optional arguments

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. `EOF` is returned if an input failure is encountered before the first conversion.

Remarks

When building with the MPLAB XC8 compiler for PIC MCUs, the `scanf()` function relies on the `getch()` function being properly defined to read input from the required peripheral or location. Scanning will not work as expected until the `getch()` stub (found in the `pic/sources/c99/common/getch.c` of your compiler distribution) is completed. See 4.1 Example code for 8-bit PIC MCUs for information on defining `getch()` so that it can read text from a file when you are using the MPLAB X IDE simulator.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string can be composed of white space characters, which reads input up to the first non-white-space character in the input; or ordinary multibyte characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `short int` or `unsigned short int` object referenced by the pointer argument.
`hh`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `signed char` or `unsigned char` object referenced by the pointer argument.

.....continued

Modifier	Meaning
l	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long int or unsigned long int object referenced by the pointer argument.
l	When used the a, A, e, E, f, F, g, or G indicates that this conversion specifier assigns to a double object referenced by the pointer argument.
l	When used the c, s, or [indicates that this conversion specifier assigns to a wchar_t object referenced by the pointer argument.
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single character or the number of characters specified by the field width if present. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character as if by a call to the mbrtowc function. No null character or wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 0 for the base argument.

.....continued

Specifier	Receiving object type	Matches
n	int	No input is consumed but the number of characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space characters, which is written to the array argument and appended with a null terminating character. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character and terminated with a null wide character.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 16 for the base argument.
%		Matches a single % character. No assignment takes place.
[	char array, or wchar_t array	Matches all the characters in the input that have been specified between the [and trailing] character, unless the character after the opening bracket is a circumflex, ^, in which case a match is only made with characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket character will match the input and the next following right bracket character is the matching right bracket that ends the specification. A null character will be appended to the characters read. If an l length modifier is present, the input shall be a sequence of multibyte characters and a null wide character will be appended to the matched wide characters.

Example

```
#include <stdio.h>

int main(void)
{
  int number, items;
  char letter;
  char color[30], string[30];
  float salary;

  printf("Enter your favorite number, favorite letter, ");
  printf("favorite color desired salary and SSN:\n");
  items = scanf("%d %c %[A-Za-z] %f %s", &number, &letter, &color, &salary, &string);

  printf("Number of items scanned = %d\n", items);
  printf("Favorite number = %d, ", number);
  printf("Favorite letter = %c\n", letter);
  printf("Favorite color = %s, ", color);
  printf("Desired salary = $%.2f\n", salary);
```

```
  printf("Social Security Number = %s, ", string);
}
```

Example Input

Contents of `UartIn.txt` (used as `stdin` input for simulator):

```
"5 T Green 300000 123-45-6789"
```

Example Output

```
Enter your favorite number, favorite letter,
favorite color, desired salary and SSN:
Number of items scanned = 5
Favorite number = 5, Favorite letter = T
Favorite color = Green, Desired salary = $300000.00
Social Security Number = 123-45-6789
```

5.18.41 setbuf Function

Defines how a stream is buffered.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
void setbuf(FILE * restrict stream, char * restrict buf);
```

Arguments

stream	pointer to the open stream
buf	user allocated buffer

Remarks

The `setbuf` function must be called after `fopen` but before any other function calls that operate on the stream. If `buf` is a null pointer, `setbuf` calls the function `setvbuf(stream, 0, _IONBF, BUFSIZ)` for no buffering; otherwise `setbuf` calls `setvbuf(stream, buf, _IOFBF, BUFSIZ)` for full buffering with a buffer of size `BUFSIZ`. See `setvbuf`.

When building for MPLAB XC16, this function requires a heap.

Example

```
#include <stdio.h>

int main(void)
{
  FILE *myfile1, *myfile2;
  char buf[BUFSIZ];

  if ((myfile1 = fopen("afile1", "w+")) != NULL)
  {
    setbuf(myfile1, NULL);
    printf("myfile1 has no buffering\n");
    fclose(myfile1);
  }

  if ((myfile2 = fopen("afile2", "w+")) != NULL)
  {
    setbuf(myfile2, buf);
```

```
    printf("myfile2 has full buffering");
    fclose(myfile2);
  }

}
```

Example Output

```
myfile1 has no buffering
myfile2 has full buffering
```

This macro is used by the function `setbuf`.

5.18.42 setvbuf Function

Defines the stream to be buffered and the buffer size.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);
```

Arguments

stream	pointer to the open stream
buf	user allocated buffer
mode	type of buffering
size	size of buffer

Return Value

Returns 0 if successful

Remarks

`setvbuf` must be called after `fopen` but before any other function calls that operate on the stream. For the `mode` argument, use one of the following tabulated modes.

Mode	Meaning
_IOFBF	for full buffering
_IOLBF	for line buffering
_IONBF	for no buffering

When using MPLAB XC16, this function requires a heap.

Example

```
#include <stdio.h>

int main(void)
{
  FILE *myfile1, *myfile2;
  char buf[256];
  if ((myfile1 = fopen("afile1", "w+")) != NULL)
  {
```

```
    if (setvbuf(myfile1, NULL, _IONBF, 0) == 0)
      printf("myfile1 has no buffering\n");
    else
      printf("Unable to define buffer stream and/or size\n");
  }
  fclose(myfile1);
  if ((myfile2 = fopen("afile2", "w+")) != NULL)
  {
    if (setvbuf(myfile2, buf, _IOFBF, sizeof(buf)) == 0)
      printf("myfile2 has a buffer of %d characters\n", sizeof(buf));
    else
      printf("Unable to define buffer stream and/or size\n");
  }
  fclose(myfile2);
}
```

Example Output

```
myfile1 has no buffering
myfile2 has a buffer of 256 characters
```

5.18.43 snprintf Function

Prints formatted text to a string.

Include

<stdio.h>

Prototype

```
int snprintf(char * restrict s, , size_t n, const char * restrict format, ...);
```

Arguments

s	storage string for output
n	maximum number of characters to write to the array
format	format control string
...	optional arguments

Return Value

Returns the number of characters stored in s, excluding the terminating null character.

Remarks

The function writes at most n characters of formatted output to the specified array, rather than a stream.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The format string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the % character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

%[*flags*][*width*][.*precision*][*length*]*specifier*

The *flags* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.

.....continued

Flag	Meaning
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and − flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first character of a conversion result does not include a sign (+ or −) or if a signed conversion results in no characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag − has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
hh	When used with the n conversion specifier, indicates that the pointer argument points to a signed char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.

.....continued

Modifier	Meaning
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used with the a, A, e, E, f, F, g, G conversion specifiers, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a char type and printed as a single character. When the l modifier is present, the wint_t argument is converted to multibyte characters then printed.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.

.....continued

Specifier	Argument value type	Printing notes
f, F	double	Converted to decimal notation using the general form [−]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of characters written so far is written to the object pointed to by the pointer. No characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (not necessarily an address) held by the pointer.
s	char array, or wchar_t array	A string, whose characters are written up to but not including the terminating null character. When the l modifier is present, wide characters from the array are converted to multibyte characters up to and including a terminating null wide character, which are then written up to but not including the terminating null character.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % character is printed.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  char sbuf[100], s[]="Print this string";
  int x = 1, y;
  char a = '\n';

  y = snprintf(sbuf, 8, "%s %d time%c", s, x, a);

  printf("Number of characters formatted = %d\n", y);
  printf("Length of string printed (excluding nul) = %lu\n", strlen(sbuf));
}
```

Example Output

```
Number of characters formatted = 25
Length of string printed (excluding nul) = 7
```

5.18.44 sprintf Function

Prints formatted text to a string.

Include

```
<stdio.h>
```

Prototype

```
int sprintf(char * restrict s, const char * format, ...);
```

Arguments

s	storage string for output
format	format control string
...	optional arguments

Return Value

Returns the number of characters stored in `s`, excluding the terminating null character.

Remarks

The function writes formatted output to the specified array, rather than a stream.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[`*`flags`*`][`*`width`*`][.`*`precision`*`][`*`length`*`]`*`specifier`*

The *`flags`* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.
0	Use 0 for the pad character instead of space (which is the default) for `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, except when converting an infinity or NaN. If the `0` and - flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and`X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first character of a conversion result does not include a sign (+ or -) or if a signed conversion results in no characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for `o` conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For `x` (or `X`) conversions, `0x` (or `0X`) is prefixed to nonzero results. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. For `g` and `G` conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *`width`* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag - has been used.

If the asterisk, `*`, is used instead of a decimal number, the `int` argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *`precision`* field indicates:

- the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions
- the number of digits to appear after the decimal-point character for `a`, `A`, `e`, `E`, `f`, and `F` conversions
- the maximum number of significant digits for the `g` and `G` conversions
- the maximum number of bytes to be written for `s` conversions

The precision takes the form of a period, `.`, followed either by an asterisk, `*` or by an optional decimal integer. If neither the `*` or integer is specified, the precision is assumed to be zero. If the asterisk, `*`, is used instead of a decimal number, the `int` argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *`length`* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, converts the argument value to a `short int` or `unsigned short int`.
`h`	When used with the `n` conversion specifier, indicates that the pointer argument points to a `short int`.
`hh`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, converts the argument value to a `signed char` or `unsigned char`.
`hh`	When used with the `n` conversion specifier, indicates that the pointer argument points to a `signed char`.
`j`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, indicates that the argument value is a`intmax_t` or `uintmax_t`.
`j`	When used with the `n` conversion specifier, indicates that the pointer points to a `intmax_t`.
`l`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, converts the argument value to a `long int` or `unsigned long int`.
`l`	When used with the `n` conversion specifier, indicates that the pointer points to a `long int`.
`l`	When used with the `c` conversion specifier, indicates that the argument value is a wide character (`wint_t` type).
`l`	When used with the `s` conversion specifier, indicates that the argument value is a wide string (`wchar_t` type).
`l`	When used the `e`, `E`, `f`, `F`, `g`, `G`, has no effect.
`ll`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, indicates that the argument value is a `long long int` or `unsigned long long int`.
`ll`	When used with the `n` conversion specifier, indicates that the pointer points to a `long long int`.
`ll`	MPLAB XC16: When used with the `s` conversion specifier, indicates that the string pointer is an `__eds__` pointer. Other compilers: The modifier is silently ignored.
`L`	When used with the `a`, `A`, `e`, `E`, `f`, `F`, `g`, `G` conversion specifiers, indicates the argument value is a `long double`.

.....continued

Modifier	Meaning
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a char type and printed as a single character. When the l modifier is present, the wint_t argument is converted to multibyte characters then printed.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
f, F	double	Converted to decimal notation using the general form [−]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of characters written so far is written to the object pointed to by the pointer. No characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (not necessarily an address) held by the pointer.
s	char array, or wchar_t array	A string, whose characters are written up to but not including the terminating null character. When the l modifier is present, wide characters from the array are converted to multibyte characters up to and including a terminating null wide character, which are then written up to but not including the terminating null character.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear

.....continued		
Specifier	**Argument value type**	**Printing notes**
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % character is printed.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

```
#include <stdio.h>

int main(void)
{
  char sbuf[100], s[]="Print this string";
  int x = 1, y;
  char a = '\n';

  y = sprintf(sbuf, "%s %d time%c", s, x, a);

  printf("Number of characters printed to "
         "string buffer = %d\n", y);
  printf("String = %s\n", sbuf);
}
```

Example Output

```
Number of characters printed to string buffer = 25
String = Print this string 1 time
```

Related Links

5.18.33 printf Function

5.18.45 sscanf Function

Scans formatted text from a string.

Include

<stdio.h>

Prototype

```
int sscanf(const char * restrict s, const char * restrict format, ...);
```

Arguments

s	storage string for input
format	format control string
...	optional arguments

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. EOF is returned if an input error is encountered before the first conversion.

Remarks

This function can read from any string.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string can be composed of white space characters, which reads input up to the first non-white-space character in the input; or ordinary multibyte characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `short int` or `unsigned short int` object referenced by the pointer argument.
`hh`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `signed char` or `unsigned char` object referenced by the pointer argument.
`l`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `long int` or `unsigned long int` object referenced by the pointer argument.
`l`	When used the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` indicates that this conversion specifier assigns to a `double` object referenced by the pointer argument.
`l`	When used the `c`, `s`, or `[` indicates that this conversion specifier assigns to a `wchar_t` object referenced by the pointer argument.
`ll`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `long long int` or `unsigned long long int` object referenced by the pointer argument.
`L`	When used the `a`, `A`, `e`, `E`, `f`, `F`, `g`, `G` conversion specifier, indicates the argument value is a `long double`.
`j`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `intmax_t` or `uintmax_t` object referenced by the pointer argument.
`t`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `ptrdiff_t` object referenced by the pointer argument.
`z`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `size_t` object referenced by the pointer argument.

The conversion *`specifiers`* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
`a`	`double`	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the `strtod` function for the first argument.

.....continued

Specifier	Receiving object type	Matches
c	char array, or wchar_t array	Matches a single character or the number of characters specified by the field width if present. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character as if by a call to the mbrtowc function. No null character or wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 0 for the base argument.
n	int	No input is consumed but the number of characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space characters, which is written to the array argument and appended with a null terminating character. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character and terminated with a null wide character.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 16 for the base argument.
%		Matches a single % character. No assignment takes place.

.....continued

Specifier	Receiving object type	Matches
[	char array, or wchar_t array	Matches all the characters in the input that have been specified between the [and trailing] character, unless the character after the opening bracket is a circumflex, ^, in which case a match is only made with characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket character will match the input and the next following right bracket character is the matching right bracket that ends the specification. A null character will be appended to the characters read. If an l length modifier is present, the input shall be a sequence of multibyte characters and a null wide character will be appended to the matched wide characters.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

```
#include <stdio.h>

int main(void)
{
  char s[] = "5 T green 3000000.00";
  int number, items;
  char letter;
  char color[10];
  float salary;

  items = sscanf(s, "%d %c %s %f", &number, &letter,
              &color, &salary);

  printf("Number of items scanned = %d\n", items);
  printf("Favorite number = %d\n", number);
  printf("Favorite letter = %c\n", letter);
  printf("Favorite color = %s\n", color);
  printf("Desired salary = $%.2f\n", salary);

}
```

Example Output

```
Number of items scanned = 4
Favorite number = 5
Favorite letter = T
Favorite color = green
Desired salary = $3000000.00
```

5.18.46 tmpfile Function

Creates a temporary file.

Attention: This function is not implemented by MPLAB XC8.

Include

<stdio.h>

Prototype

```
FILE * tmpfile(void)
```

Return Value

Returns a stream pointer if successful; otherwise, returns a `NULL` pointer.

Remarks

`tmpfile` creates a file with a unique filename. The temporary file is opened in `w+b` (binary read/write) mode. It will automatically be removed when exit is called; otherwise the file will remain in the directory. This function requires a heap.

Example

```
#include <stdio.h>
int main(void)
{
  FILE *mytempfile;

  if ((mytempfile = tmpfile()) == NULL)
    printf("Cannot create temporary file");
  else
    printf("Temporary file was created");
}
```

Example Output

```
Temporary file was created
```

5.18.47 tmpnam Function

Creates a unique temporary filename.

Attention: This function is not implemented by MPLAB XC8.

Include

`<stdio.h>`

Prototype

```
char *tmpnam(char * s);
```

Argument

s	pointer to the temporary name

Return Value

Returns a pointer to the filename generated and stores the filename in `s`. If it can not generate a filename, the `NULL` pointer is returned.

Remarks

The created filename will not conflict with an existing file name. Use `L_tmpnam` to define the size of array the argument of `tmpnam` points to.

Example

```
#include <stdio.h>

int main(void)
{
  char *myfilename;
  char mybuf[L_tmpnam];
  char *myptr = (char *) &mybuf;

  if ((myfilename = tmpnam(myptr)) == NULL)
    printf("Cannot create temporary file name");
  else
```

```
    printf("Temporary file %s was created", myfilename);
}
```

Example Output

```
Temporary file ctm00001.tmp was created
```

5.18.48 ungetc Function

Pushes character back onto stream.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int ungetc(int c, FILE * stream);
```

Arguments

c	character to be pushed back
stream	pointer to the open stream

Return Value

Returns the pushed character if successful; otherwise, returns `EOF`.

Remarks

The pushed back character will be returned by a subsequent read on the stream. If more than one character is pushed back, they will be returned in the reverse order of their pushing. A successful call to a file positioning function (`fseek`, `fsetpos` or `rewind`) cancels any pushed back characters. Only one character of push back is guaranteed. Multiple calls to `ungetc` without an intervening read or file positioning operation may cause a failure.

Example

```
#include <stdio.h>

int main(void)
{
  FILE * buf;
  char y, c;

  if ((buf = fopen("afile.txt", "r")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    y = fgetc(buf);
    while (y != EOF)
    {
      if (y == 'r')
      {
        c = ungetc(y, buf);
        if (c != EOF)
        {
          printf("2");
          y = fgetc(buf);
        }
      }
      printf("%c", y);
      y = fgetc(buf);
    }
    fclose(buf);
```

```
  }
}
```

Example Input

Contents of `afile.txt` (used as input):

```
Short
Longer string
```

Example Output

```
Sho2rt
Longe2r st2ring
```

5.18.49 vfprintf Function

Prints formatted data to a stream using a variable length argument list.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

`<stdarg.h>`

Prototype

```
int vfprintf(FILE * restrict stream, const char * restrict format, va_list ap);
```

Arguments

stream	pointer to the open stream
format	format control string
ap	pointer to a list of arguments

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

To access the arguments, the `ap` object must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vfprintf` function is called. Invoke `va_end` after the function returns. For more details, see `stdarg.h`.

When using MPLAB XC16, this function requires a heap.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[`*`flags`*`][`*`width`*`][.`*`precision`*`][`*`length`*`]`*`specifier`*

The *flags* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first character of a conversion result does not include a sign (+ or -) or if a signed conversion results in no characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag - has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
hh	When used with the n conversion specifier, indicates that the pointer argument points to a signed char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.

.....continued

Modifier	Meaning
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used with the a, A, e, E, f, F, g, G conversion specifiers, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a char type and printed as a single character. When the l modifier is present, the wint_t argument is converted to multibyte characters then printed.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.

.....continued

Specifier	Argument value type	Printing notes
f, F	double	Converted to decimal notation using the general form [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of characters written so far is written to the object pointed to by the pointer. No characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (not necessarily an address) held by the pointer.
s	char array, or wchar_t array	A string, whose characters are written up to but not including the terminating null character. When the l modifier is present, wide characters from the array are converted to multibyte characters up to and including a terminating null wide character, which are then written up to but not including the terminating null character.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % character is printed.

Example

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

FILE *myfile;

void errmsg(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  vfprintf(myfile, fmt, ap);
  va_end(ap);
}
int main(void)
{
  int num = 3;

  if ((myfile = fopen("afile.txt", "w")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    errmsg("Error: The letter '%c' is not %s\n", 'a', "an integer value.");
    errmsg("Error: Requires %d%s%c", num, " or more characters.", '\n');
  }
```

```
  fclose(myfile);
}
```

Example Output

Contents of `afile.txt`:

```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

Related Links

5.18.33 printf Function

5.18.50 vfscanf Function

Scans formatted data from a stream using a variable length argument list.

Attention: This function is not implemented by MPLAB XC8 for PIC MCUs. It is implemented by all other compilers, but MPLAB XC8 for AVR MCUs has limited support of data streams.

Include

`<stdio.h>`

Prototype

```
int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);
```

Arguments

stream	pointer to the open stream
format	format control string
arg	pointer to a list of arguments

Return Value

If an input failure occurs before any conversion. the function returns the value of the macro `EOF`; otherwise, it returns the number of input items assigned.

Remarks

To access the variable length argument list, the `arg` variable must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vfscanf` function is called. Invoke `va_end` after the function returns.

When using MPLAB XC16, this function requires a heap.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string can be composed of white space characters, which reads input up to the first non-white-space character in the input; or ordinary multibyte characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent character followed by optional fields and a required type as shown here:

`%[*][width][length]specifier`

The presence of `*` immediately after the percent character indicates that the read and converted value should not be assigned to any object.

The *width* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *length* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a short int or unsigned short int object referenced by the pointer argument.
hh	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a signed char or unsigned char object referenced by the pointer argument.
l	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long int or unsigned long int object referenced by the pointer argument.
l	When used the a, A, e, E, f, F, g, or G indicates that this conversion specifier assigns to a double object referenced by the pointer argument.
l	When used the c, s, or [indicates that this conversion specifier assigns to a wchar_t object referenced by the pointer argument.
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single character or the number of characters specified by the field width if present. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character as if by a call to the mbrtowc function. No null character or wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.

.....continued

Specifier	Receiving object type	Matches
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 0 for the base argument.
n	int	No input is consumed but the number of characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space characters, which is written to the array argument and appended with a null terminating character. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character and terminated with a null wide character.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 16 for the base argument.
%		Matches a single % character. No assignment takes place.
[	char array, or wchar_t array	Matches all the characters in the input that have been specified between the [and trailing] character, unless the character after the opening bracket is a circumflex, ^, in which case a match is only made with characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket character will match the input and the next following right bracket character is the matching right bracket that ends the specification. A null character will be appended to the characters read. If an l length modifier is present, the input shall be a sequence of multibyte characters and a null wide character will be appended to the matched wide characters.

Example

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

void readArgs(FILE * stream, const char * format, ...)
{
  va_list args;
  va_start(args, format);
  vfscanf(stream, format, args);
  va_end(args);
```

```
}

int main(void)
{
  FILE * myfile;
  int val;
  char str[20];

  myfile = fopen("afile", "r");

  if(myfile!=NULL) {
    readArgs( myfile, "%s %d", str, &val);
    printf("String and value read: %s and %d\n", str, val);
    fclose(myfile);
  }
}
```

Example Input

Contents of `afile.txt`

```
Start 23
Stop 0
```

Example Output

```
String and value read: Start and 23
```

5.18.51 vprintf Function

Prints formatted text to `stdout` using a variable length argument list.

Include

`<stdio.h>`

`<stdarg.h>`

Prototype

```
int vprintf(const char * restrict format, va_list ap);
```

Arguments

format	format control string
ap	pointer to a list of arguments

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

To access the arguments, the `ap` object must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vfprintf` function is called. Invoke `va_end` after the function returns. For more details, see `stdarg.h`.

When using MPLAB XC16, this function requires a heap.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[flags][width][.precision][length]specifier`

The *flags* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first character of a conversion result does not include a sign (+ or -) or if a signed conversion results in no characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag - has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.

.....continued

Modifier	Meaning
hh	When used with the n conversion specifier, indicates that the pointer argument points to a signed char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used with the a, A, e, E, f, F, g, G conversion specifiers, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [-]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a char type and printed as a single character. When the l modifier is present, the wint_t argument is converted to multibyte characters then printed.
d	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.

.....continued

Specifier	Argument value type	Printing notes
e, E	double	Converted to the general form [−]*d*.*ddd*e±*dd*, where there is one digit before the decimal-point character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
f, F	double	Converted to decimal notation using the general form [−]*ddd*.*ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of characters written so far is written to the object pointed to by the pointer. No characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (not necessarily an address) held by the pointer.
s	char array, or wchar_t array	A string, whose characters are written up to but not including the terminating null character. When the l modifier is present, wide characters from the array are converted to multibyte characters up to and including a terminating null wide character, which are then written up to but not including the terminating null character.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % character is printed.

Example

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

void errmsg(const char *fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  printf("Error: ");
  vprintf(fmt, ap);
  va_end(ap);
}
int main(void)
{
  int num = 3;

  errmsg("The letter '%c' is not %s\n", 'a', "an integer value.");
```

```
  errmsg("Requires %d%s\n", num, " or more characters.\n");
}
```

Example Output

```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

Related Links

5.18.33 printf Function

5.18.52 vscanf Function

Scans formatted text from `stdin` using a variable-length argument list object.

Include

`<stdio.h>`

Prototype

```
int vscanf(const char * restrict format, va_list arg);
```

Arguments

`format`	format control string
`arg`	pointer to a list of arguments

Return Value

If an input failure occurs before any conversion. the function returns the value of the macro `EOF`; otherwise, it returns the number of input items assigned.

Remarks

To access the variable-length argument list, the `arg` variable must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vscanf` function is called. Invoke `va_end` after the function returns.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string can be composed of white space characters, which reads input up to the first non-white-space character in the input; or ordinary multibyte characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `short int` or `unsigned short int` object referenced by the pointer argument.

.....continued

Modifier	Meaning
hh	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a signed char or unsigned char object referenced by the pointer argument.
l	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long int or unsigned long int object referenced by the pointer argument.
l	When used the a, A, e, E, f, F, g, or G indicates that this conversion specifier assigns to a double object referenced by the pointer argument.
l	When used the c, s, or [indicates that this conversion specifier assigns to a wchar_t object referenced by the pointer argument.
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single character or the number of characters specified by the field width if present. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character as if by a call to the mbrtowc function. No null character or wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 0 for the base argument.

.....continued

Specifier	Receiving object type	Matches
n	int	No input is consumed but the number of characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space characters, which is written to the array argument and appended with a null terminating character. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character and terminated with a null wide character.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 16 for the base argument.
%		Matches a single % character. No assignment takes place.
[	char array, or wchar_t array	Matches all the characters in the input that have been specified between the [and trailing] character, unless the character after the opening bracket is a circumflex, ^, in which case a match is only made with characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket character will match the input and the next following right bracket character is the matching right bracket that ends the specification. A null character will be appended to the characters read. If an l length modifier is present, the input shall be a sequence of multibyte characters and a null wide character will be appended to the matched wide characters.

Example

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

void readArgs(const char * format, ...)
{
  va_list args;
  va_start(args, format);
  vscanf(format, args);
  va_end(args);
}

int main(void)
{
  int val;
  char str[20];

  printf("Enter a string an value\n");
  readArgs("%s %d", str, &val);
```

```
  printf("String and value read: %s and %d\n", str, val);
}
```

Example Output

```
Enter a string an value
hello 456
String and value read: hello and 456
```

5.18.53 vsnprintf Function

Prints formatted text to a string using a variable-length argument list object.

Include

`<stdio.h>`

Prototype

```
int vsprintf(char * restrict s, size_t n, const char * restrict format, va_list arg);
```

Arguments

s	storage string for output
n	maximum number of characters written
format	format control string
arg	pointer to a list of arguments

Return Value

The function returns a negative value if an encoding error occurred; otherwise, it returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character.

Remarks

The function writes at most `n` characters of formatted output to `stdout`.

To access the arguments, the `ap` object must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vfprintf` function is called. Invoke `va_end` after the function returns. For more details, see `stdarg.h`.

When using MPLAB XC16, this function requires a heap.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[`*`flags`*`][`*`width`*`][.`*`precision`*`][`*`length`*`]`*`specifier`*

The *`flags`* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.

.....continued	
Flag	**Meaning**
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and − flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first character of a conversion result does not include a sign (+ or −) or if a signed conversion results in no characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag − has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
hh	When used with the n conversion specifier, indicates that the pointer argument points to a signed char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.

.....continued	
Modifier	**Meaning**
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used with the a, A, e, E, f, F, g, G conversion specifiers, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a char type and printed as a single character. When the l modifier is present, the wint_t argument is converted to multibyte characters then printed.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.

.....continued

Specifier	Argument value type	Printing notes
f, F	double	Converted to decimal notation using the general form [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of characters written so far is written to the object pointed to by the pointer. No characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (not necessarily an address) held by the pointer.
s	char array, or wchar_t array	A string, whose characters are written up to but not including the terminating null character. When the l modifier is present, wide characters from the array are converted to multibyte characters up to and including a terminating null wide character, which are then written up to but not including the terminating null character.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % character is printed.

Example

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

char sbuf[20];

int printStr(const char * format, ... )
{
  va_list args;
  int p;

  va_start(args, format);
  p = vsnprintf(sbuf, 20, format, args);
  va_end(args);

  return p;
}

int main(void)
{
  char s[]="Print this string";
  int x = 1, y;
  char a = '\n';

  y = printStr("%s %d time%c", s, x, a);
```

```
  printf("Number of characters formatted = %d\n", y);
  printf("Length of string printed (excluding nul) = %lu\n", strlen(sbuf));
}
```

Example Output

```
Number of characters formatted = 25
Length of string printed (excluding nul) = 19
```

5.18.54 vsprintf Function

Prints formatted text to a string using a variable length argument list.

Include

<stdio.h>

<stdarg.h>

Prototype

```
int vsprintf(char * restrict s, const char * restrict format, va_list ap);
```

Arguments

s	storage string for output
format	format control string
ap	pointer to a list of arguments

Return Value

Returns number of characters stored in s excluding the terminating null character, or a negative value if an error occurred.

Remarks

To access the variable length argument list, the ap variable must be initialized by the macro va_start and may be reinitialized by additional calls to va_arg. This must be done before the vsprintf function is called. Invoke va_end after the function returns. For more details, see stdarg.h.

When building for MPLAB XC16, this function requires a heap.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The format string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the % character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

%[*flags*][*width*][.*precision*][*length*]*specifier*

The *flags* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.

.....continued

Flag	Meaning
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first character of a conversion result does not include a sign (+ or −) or if a signed conversion results in no characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag − has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
hh	When used with the n conversion specifier, indicates that the pointer argument points to a signed char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.

.....continued	
Modifier	**Meaning**
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used with the a, A, e, E, f, F, g, G conversion specifiers, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [-]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a char type and printed as a single character. When the l modifier is present, the wint_t argument is converted to multibyte characters then printed.
d	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [-]*d.ddde±dd*, where there is one digit before the decimal-point character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
f, F	double	Converted to decimal notation using the general form [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate

.....continued		
Specifier	**Argument value type**	**Printing notes**
i	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of characters written so far is written to the object pointed to by the pointer. No characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (not necessarily an address) held by the pointer.
s	char array, or wchar_t array	A string, whose characters are written up to but not including the terminating null character. When the l modifier is present, wide characters from the array are converted to multibyte characters up to and including a terminating null wide character, which are then written up to but not including the terminating null character.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % character is printed.

Example

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

void errmsg(const char *fmt, ...)
{
  va_list ap;
  char buf[100];

  va_start(ap, fmt);
  vsprintf(buf, fmt, ap);
  va_end(ap);
  printf("Error: %s", buf);
}
int main(void)
{
  int num = 3;

  errmsg("The letter '%c' is not %s\n", 'a', "an integer value.");
  errmsg("Requires %d%s\n", num, " or more characters.\n");
}
```

Example Output

```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

Related Links

5.18.33 printf Function

5.18.55 vsscanf Function

Scans formatted text from a string using a variable-length argument list object.

Include

`<stdio.h>`

Prototype

```
int vsscanf(const char * restrict s, const char * restrict format, va_list arg);
```

Arguments

s	string to read input
format	format control string
arg	variable-argument list object

Return Value

If an input failure occurs before any conversion, the function returns the value of the macro `EOF`; otherwise, it returns the number of input items assigned.

Remarks

To access the variable-length argument list, the `arg` variable must be initialized by the macro `va_start` and may be reinitialized by additional calls to `va_arg`. This must be done before the `vscanf` function is called. Invoke `va_end` after the function returns.

To reduce code size, the functionality of this routine can be customized with each build. See the Smart IO Routines section in your compiler's user's guide for more information.

The `format` string can be composed of white space characters, which reads input up to the first non-white-space character in the input; or ordinary multibyte characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `short int` or `unsigned short int` object referenced by the pointer argument.
`hh`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `signed char` or `unsigned char` object referenced by the pointer argument.
`l`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `long int` or `unsigned long int` object referenced by the pointer argument.
`l`	When used the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` indicates that this conversion specifier assigns to a `double` object referenced by the pointer argument.
`l`	When used the `c`, `s`, or `[` indicates that this conversion specifier assigns to a `wchar_t` object referenced by the pointer argument.

.....continued

Modifier	Meaning
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single character or the number of characters specified by the field width if present. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character as if by a call to the mbrtowc function. No null character or wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 0 for the base argument.
n	int	No input is consumed but the number of characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.

.....continued

Specifier	Receiving object type	Matches
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space characters, which is written to the array argument and appended with a null terminating character. If an l length modifier is present, the input shall be a sequence of multibyte characters, converted to a wide character and terminated with a null wide character.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 16 for the base argument.
%		Matches a single % character. No assignment takes place.
[	char array, or wchar_t array	Matches all the characters in the input that have been specified between the [and trailing] character, unless the character after the opening bracket is a circumflex, ^, in which case a match is only made with characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket character will match the input and the next following right bracket character is the matching right bracket that ends the specification. A null character will be appended to the characters read. If an l length modifier is present, the input shall be a sequence of multibyte characters and a null wide character will be appended to the matched wide characters.

Example

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

void readArgs(const char * s, const char * format, ...)
{
  va_list args;
  va_start(args, format);
  vsscanf(s, format, args);
  va_end(args);
}

int main(void)
{
  int val;
  char str[20];
  char buf[] = "Initiation 0xFF45";

  readArgs(buf, "%s %x", str, &val);
  printf("String and value read: %s and %d\n", str, val);
}
```

Example Output

```
String and value read: Initiation and 65349
```

5.19 <stdlib.h> Utility Functions

The header file `stdlib.h` consists of types, macros and functions that provide text conversions, memory management, searching and sorting abilities and other general utilities.

5.19.1 stdlib.h Types and Macros

5.19.1.1 div_t

A type that holds a quotient and remainder of a signed integer division with operands of type `int`.

Prototype

```
typedef struct { int quot, rem; } div_t;
```

Remarks

This is the structure type returned by the function, `div()`.

5.19.1.2 ldiv_t

A type that holds a quotient and remainder of a signed integer division with operands of type `long`.

Prototype

```
typedef struct { long quot, rem; } ldiv_t;
```

Remarks

This is the structure type returned by the function, `ldiv()`.

5.19.1.3 lldiv_t

A type that holds a quotient and remainder of a signed integer division with operands of type `long`.

Prototype

```
typedef struct { long long quot, rem; } lldiv_t;
```

Remarks

This is the structure type returned by the function, `lldiv()`.

5.19.1.4 EXIT_FAILURE

Reports unsuccessful termination.

Remarks

`EXIT_FAILURE` is a value for the exit function to return an unsuccessful termination status.

Example

See `exit` for example of use.

5.19.1.5 EXIT_SUCCESS

Reports successful termination.

Remarks

`EXIT_SUCCESS` is a value for the exit function to return a successful termination status.

Example

See `exit()` for example of use.

5.19.1.6 MB_CUR_MAX

Maximum number of characters in a multibyte character.

Value

1

5.19.1.7 RAND_MAX

Maximum value capable of being returned by the `rand()` function.

Value

32767

5.19.2 size_t Type

An unsigned integer type used by the result of the `sizeof` operator

Include

`<stddef.h>`

`<stdio.h>`

`<stdlib.h>`

`<string.h>`

`<time.h>`

`<wchar.h>`

Definition

```
typedef unsigned size_t;
```

5.19.3 wchar_t Type

An integer type capable of holding values that can represent distinct codes for all members of the largest extended character set specified among the supported locales.

Include

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

5.19.4 NULL Macro

A constant value which represent a null pointer constant. It's value and type is implementation defined.

Include

`<locale.h>`

`<stddef.h>`

`<stdio.h>`

`<stdlib.h>`

`<string.h>`

`<time.h>`

`<wchar.h>`

Definition

```
#define NULL ((void*)0)
```

5.19.5 _Exit function

Description

Terminates program execution.

Include

`<stdlib.h>`

Prototype

```
void _Exit(int status);
```

Argument

status	exit status

Remarks

When using MPLAB XC8 for PIC, the `_Exit` function causes process execution to halt in an endless loop without calling any functions registered by `atexit`.

For all other compilers, the `_Exit` function resets the processor without calling any functions registered by `atexit`. This function is customizable.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r" )) == NULL)
  {
    printf("Cannot open samp.fil\n");
    _Exit(EXIT_FAILURE);
  }
  else
  {
    printf("Success opening samp.fil\n");
    _Exit(EXIT_SUCCESS);
  }
  printf("This will not be printed");
}
```

Example Output

```
Cannot open samp.fil
```

5.19.6 abort Function

Aborts the current process.

Include

```
<stdlib.h>
```

Prototype

```
void abort(void);
```

Remarks

When using MPLAB XC8 for PIC, the `abort` function will ultimately call `_Exit()`. For all other compilers, `abort()` cause the processor to reset.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

```
int main(void)
{
    double    x, y;

    x = -2.0;
    if(x < -1.0 || x > 1.0) {
        printf("input out of range\n");
        abort();
    }
    y = acos(x);
    printf("the arc cosine of %g is %g\n", x, y);
}
```

Example Output

```
input out of range
```

5.19.7 abs Function

Calculates the absolute value.

Include

<stdlib.h>

Prototype

```
int abs(int i);
```

Argument

i	integer value

Return Value

Returns the absolute value of `i`.

Remarks

A negative number is returned as positive; a positive number is unchanged.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int i;

  i = 12;
  printf("The absolute value of  %d is  %d\n", i, abs(i));

  i = -2;
  printf("The absolute value of  %d is   %d\n", i, abs(i));

  i = 0;
  printf("The absolute value of   %d is   %d\n", i, abs(i));
}
```

Example Output

```
The absolute value of  12 is  12
The absolute value of  -2 is   2
The absolute value of   0 is   0
```

5.19.8 atexit Function

Registers the specified function to be called when the program terminates normally.

Include

`<stdlib.h>`

Prototype

```
int atexit(void(*func)(void));
```

Argument

func	the function to be called on exit

Return Value

Returns a zero if successful; otherwise, returns a non-zero value.

Remarks

For the registered functions to be called, the program must terminate with a call to the `exit` function.

When using the MPLAB XC8 for PIC, a limit of 32 function handlers applies.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void good_msg(void);
void bad_msg(void);
void end_msg(void);

int main(void)
{
  int number;

  atexit(end_msg);
  printf("Enter your favorite number:");
  scanf("%d", &number);
  printf(" %d\n", number);
  if (number == 5)
  {
    printf("Good Choice\n");
    atexit(good_msg);
    exit(0);
  }
  else
  {
    printf("%d!?\n", number);
    atexit(bad_msg);
    exit(0);
  }
}

void good_msg(void)
{
  printf("That's an excellent number\n");
}

void bad_msg(void)
{
  printf("That's an awful number\n");
}

void end_msg(void)
{
  printf("Now go count something\n");
}
```

Example Input 1

With contents of `UartIn.txt` (used as `stdin` input for simulator):

```
5
```

Example Output 1

```
Enter your favorite number: 5
Good Choice
That's an excellent number
Now go count something
```

Example Input 2

With contents of `UartIn.txt` (used as `stdin` input for simulator):

```
42
```

Example Output 2

```
Enter your favorite number: 42
42!?
That's an awful number
Now go count something
```

5.19.9 atof Function

Converts a string to a double precision floating-point value.

Include

`<stdlib.h>`

Prototype

```
double atof(const char *s);
```

Argument

s pointer to the string to be converted

Return Value

Returns the converted value if successful; otherwise, returns 0.

Remarks

The number may consist of the following:

`[`*whitespace*`] [`*sign*`] `*digits*` [.`*digits*`] [ { e | E }[`*sign*`]`*digits*`]`

Optional whitespace followed by an optional sign, then a sequence of one or more digits with an optional decimal point, followed by one or more optional digits and an optional `e` or `E` followed by an optional signed exponent. The conversion stops when the first unrecognized character is reached. The conversion is the same as `strtod(s,0)` except it does no error checking so `errno` will not be set.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char a[] = " 1.28";
  char b[] = "27.835e2";
  char c[] = "Number1";
  double x;
```

```
  x = atof(a);
  printf("String = \"%s\" float = %f\n", a, x);

  x = atof(b);
  printf("String = \"%s\" float = %f\n", b, x);

  x = atof(c);
  printf("String = \"%s\"  float = %f\n", c, x);
}
```

Example Output

```
String = "1.28"     float = 1.280000
String = "27.835e2" float = 2783.500000
String = "Number1"  float = 0.000000
```

5.19.10 atoi Function

Converts a string to an integer.

Include

`<stdlib.h>`

Prototype

```
int atoi(const char *s);
```

Argument

s	string to be converted

Return Value

Returns the converted integer if successful; otherwise, returns 0.

Remarks

The number may consist of the following:

[whitespace] [sign] digits

Optional whitespace followed by an optional sign, then a sequence of one or more digits. The conversion stops when the first unrecognized character is reached. The conversion is equivalent to `(int) strtol(s,0,10)`, except it does no error checking so `errno` will not be set.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{

  char a[] = " -127";
  char b[] = "Number1";
  int x;

  x = atoi(a);
  printf("String = \"%s\"\tint = %d\n", a, x);

  x = atoi(b);
  printf("String = \"%s\"\tint = %d\n", b, x);
}
```

Example Output

```
String = " -127"        int = -127
String = "Number1"      int = 0
```

5.19.11 atol Function

Converts a string to a long integer.

Include

```
<stdlib.h>
```

Prototype

```
long atol(const char *s);
```

Argument

s	string to be converted

Return Value

Returns the converted long integer if successful; otherwise, returns 0.

Remarks

The number may consist of the following:

`[whitespace] [sign] digits`

Optional whitespace followed by an optional sign, then a sequence of one or more digits. The conversion stops when the first unrecognized character is reached. The conversion is equivalent to `(int) strtol(s,0,10)`, except it does no error checking so `errno` will not be set.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char a[] = " -123456";
  char b[] = "2Number";
  long x;

  x = atol(a);
  printf("String = \"%s\"  int = %ld\n", a, x);

  x = atol(b);
  printf("String = \"%s\"  int = %ld\n", b, x);
}
```

Example Output

```
String = " -123456"     int = -123456
String = "2Number"      int = 2
```

5.19.12 atoll Function

Converts a string to a long long integer.

Include

```
<stdlib.h>
```

Prototype

```
long long atoll(const char *s);
```

Argument

s	the string to be converted

Return Value

Returns the converted `long long` integer if successful; otherwise, returns 0.

Remarks

The number may consist of the following:

`[whitespace] [sign] digits`

Optional whitespace followed by an optional sign, then a sequence of one or more digits. The conversion stops when the first unrecognized character is reached. The conversion is equivalent to `(int) strtoll(s, (char **)NULL, 10)`, except it does no error checking, so `errno` will not be set.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char a[] = " -123456";
  char b[] = "2Number";
  long long x;

  x = atoll(a);
  printf("String = \"%s\"  int = %lld\n", a, x);

  x = atoll(b);
  printf("String = \"%s\"  int = %lld\n", b, x);
}
```

Example Output

```
String = " -123456"     int = -123456
String = "2Number"      int = 2
```

5.19.13 bsearch Function

Performs a binary search.

Include

`<stdlib.h>`

Prototype

```
void *bsearch(const void *key, const void *base, size_t nelem, size_t size, int (*cmp)
(const void *ck, const void *ce));
```

Arguments

key	object to search for
base	pointer to the start of the search data
nelem	number of elements
size	size of elements
cmp	pointer to the comparison function

Arguments to the comparison function are as follows.

ck pointer to the key for the search

ce pointer to the element being compared with the key

Return Value

Returns a pointer to the object being searched for if found; otherwise, returns `NULL`.

Remarks

The value returned by the compare function is <0 if `ck` is less than `ce`, 0 if `ck` is equal to `ce` or >0 if `ck` is greater than `ce`.

In the following example, `qsort` is used to sort the list before `bsearch` is called. `bsearch` requires the list to be sorted according to the comparison function. This `comp` uses ascending order.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

#define NUM 7

int comp(const void *e1, const void *e2);

int main(void)
{
  int list[NUM] = {35, 47, 63, 25, 93, 16, 52};
  int x, y;
  int *r;

  qsort(list, NUM, sizeof(int), comp);

  printf("Sorted List:   ");
  for (x = 0; x < NUM; x++)
    printf("%d  ", list[x]);

  y = 25;
  r = bsearch(&y, list, NUM, sizeof(int), comp);
  if (r)
    printf("\nThe value %d was found\n", y);
  else
    printf("\nThe value %d was not found\n", y);

  y = 75;
  r = bsearch(&y, list, NUM, sizeof(int), comp);
  if (r)
    printf("\nThe value %d was found\n", y);
  else
    printf("\nThe value %d was not found\n", y);
}

int comp(const void *e1, const void *e2)
{

  const int * a1 = e1;
  const int * a2 = e2;

  if (*a1 < *a2)
    return -1;
  else if (*a1 == *a2)
    return 0;
  else
    return 1;
}
```

Example Output

```
Sorted List:   16  25  35  47  52  63  93
The value 25 was found
```

```
The value 75 was not found
```

5.19.14 calloc Function

Dynamically allocate and clear a block of memory from the heap for an array of objects.

Include

`<stdlib.h>`

Prototype

```
void * calloc(size_t nelem, size_t size);
```

Arguments

nelem	number of array elements
size	size of each element

Return Value

Returns a pointer to the allocated space if successful; otherwise, returns a null pointer.

Remarks

Although the memory block allocated and whose address is returned will be large enough to hold an object with the specified size, the memory required to allocate that block might be larger than the requested size due to alignment of the allocated memory, and/or additional objects that are used to manage the heap.

Unlike `malloc`, the `calloc` function initializes the memory it allocates to 0.

See your MPLAB XC compiler User's Guide for more information on dynamic memory allocation.

Example

```
/* This program allocates memory for the      */
/* array 'i' of long integers and initializes */
/* them to zero.                              */

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

int main(void)
{
  int x;
  long *i;

  i = (long *)calloc(5, sizeof(long));
  if (i != NULL)
  {
    for (x = 0; x < 5; x++)
      printf("i[%d] = %ld\n", x, i[x]);
    free(i);
  }
  else
    printf("Cannot allocate memory\n");
}
```

Example Output

```
i[0] = 0
i[1] = 0
i[2] = 0
i[3] = 0
i[4] = 0
```

5.19.15 div Function

Calculates the quotient and remainder of two numbers.

Include

<stdlib.h>

Prototype

```
div_t div(int numer, int denom);
```

Arguments

numer	numerator
denom	denominator

Return Value

Returns the quotient and the remainder.

Remarks

The returned quotient will have the same sign as the numerator divided by the denominator. The sign for the remainder will be such that the quotient times the denominator plus the remainder will equal the numerator (`quot * denom + rem = numer`).

If either part of the result cannot be represented, such as when attempting to perform a division by zero, the behavior is undefined. MPLAB XC compiler might raise an exception in such circumstances.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int x, y;
  div_t z;

  x = 7;
  y = 3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the remainder is %d\n\n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the remainder is %d\n\n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the remainder is %d\n\n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For div(%d, %d)\n", x, y);
  z = div(x, y);
  printf("The quotient is %d and the remainder is %d\n\n", z.quot, z.rem);
}
```

Example Output

```
For div(7, 3)
The quotient is 2 and the remainder is 1

For div(7, -3)
The quotient is -2 and the remainder is 1

For div(-5, 3)
The quotient is -1 and the remainder is -2
```

```
For div(7, 7)
The quotient is 1 and the remainder is 0
```

5.19.16 exit Function

Terminates program after clean up.

Include

`<stdlib.h>`

Prototype

`void exit(int status);`

Argument

status	the exit status to return

Remarks

The `exit` function calls any functions registered by `atexit` in reverse order of registration, flushes buffers, closes streams, closes any temporary files created with `tmpfile` and then calls `_Exit`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main()
{
    double result, x=-9.0;

    result = log(x);
    if(errno) {
        printf("terminating execution\n");
        exit(EXIT_FAILURE);
    } else
        printf("Log of %g is %g\n", x, result);
}
```

Example Output

```
terminating execution
```

5.19.17 free Function

Frees dynamically allocated memory.

Include

`<stdlib.h>`

Prototype

`void free(void * ptr);`

Argument

ptr	pointer to the memory to be freed

Remarks

Frees memory previously allocated with `calloc`, `malloc` or `realloc`. If `free` is used on space that has already been deallocated (by a previous call to `free` or by `realloc`) or on space not allocated with `calloc`, `malloc` or `realloc`, the behavior is undefined.

When using the simple implementation of dynamic memory allocation, only a rudimentary merging of freed memory blocks is performed. See your MPLAB XC compiler User's Guide for more information on how dynamic memory allocation is implemented with your compiler.

Example

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

int main(void)
{
  long *i;

  if ((i = (long *)malloc(50 * sizeof(long))) == NULL)
    printf("Cannot allocate memory\n");
  else
  {
    printf("Memory allocated\n");
    free(i);
    printf("Memory freed\n");
  }
}
```

Example Output

```
Memory allocated
Memory freed
```

5.19.18 getenv Function

Get a value for an environment variable.

Attention: This function is not implemented by MPLAB XC8 C compilers.

Include

`<stdlib.h>`

Prototype

```
char * getenv(const char * name);
```

Argument

name	name of environment variable to read

Return Value

Returns a pointer to the value of the environment variable if successful; otherwise, returns a null pointer.

Remarks

This function must be customized to be used as described (see `pic30-libs`). By default, there are no entries in the environment list for `getenv` to find.

Example

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

int main(void)
{
```

```
  char *incvar;

  incvar = getenv("INCLUDE");
  if (incvar != NULL)
    printf("INCLUDE environment variable = %s\n", incvar);
  else
    printf("Cannot find environment variable INCLUDE ");
}
```

Example Output

```
Cannot find environment variable INCLUDE
```

5.19.19 labs Function

Calculates the absolute value of a long integer.

Include

`<stdlib.h>`

Prototype

```
long labs(long i);
```

Argument

i	long integer value

Return Value

Returns the absolute value of `i`.

Remarks

A negative number is returned as positive; a positive number is unchanged.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long i;

  i = 123456;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));

  i = -246834;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));

  i = 0;
  printf("The absolute value of %7ld is %6ld\n",
         i, labs(i));
}
```

Example Output

```
The absolute value of  123456 is 123456
The absolute value of -246834 is 246834
The absolute value of       0 is      0
```

5.19.20 ldiv Function

Calculates the quotient and remainder of two long integers.

Include

`<stdlib.h>`

Prototype

```
ldiv_t ldiv(long numer, long denom);
```

Arguments

numer	numerator
denom	denominator

Return Value

Returns the quotient and the remainder.

Remarks

The returned quotient will have the same sign as the numerator divided by the denominator. The sign for the remainder will be such that the quotient times the denominator plus the remainder will equal the numerator (`quot * denom + rem = numer`). If the denominator is zero, the behavior is undefined.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long x,y;
  ldiv_t z;

  x = 7;
  y = 3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n", z.quot, z.rem);

  x = 7;
  y = 0;
  printf("For ldiv(%ld, %ld)\n", x, y);
  z = ldiv(x, y);
  printf("The quotient is %ld and the "
         "remainder is %ld\n\n",
         z.quot, z.rem);
}
```

Example Output

```
For ldiv(7, 3)
The quotient is 2 and the remainder is 1

For ldiv(7, -3)
The quotient is -2 and the remainder is 1

For ldiv(-5, 3)
The quotient is -1 and the remainder is -2

For ldiv(7, 7)
The quotient is 1 and the remainder is 0

For ldiv(7, 0)
The quotient is -1 and the remainder is 7
```

Example Explanation

In the last example (`ldiv(7,0)`) the denominator is zero, the behavior is undefined.

5.19.21 llabs Function

Calculates the absolute value of a long long integer.

Include

`<stdlib.h>`

Prototype

`long long llabs(long long` **`i`**`);`

Argument

`i`	long integer value

Return Value

Returns the absolute value of `i`.

Remarks

A negative number is returned as positive; a positive number is unchanged.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long long i;

  i = 123456;
  printf("The absolute value of %7lld is %6lld\n", i, llabs(i));

  i = -246834;
  printf("The absolute value of %7lld is %6lld\n", i, llabs(i));

  i = 0;
  printf("The absolute value of %7lld is %6lld\n", i, llabs(i));
}
```

Example Output

```
The absolute value of  123456 is 123456
The absolute value of -246834 is 246834
The absolute value of       0 is      0
```

5.19.22 lldiv Function

Calculates the quotient and remainder of two long integers.

Include

`<stdlib.h>`

Prototype

```
lldiv_t lldiv(long numer, long denom);
```

Arguments

numer	numerator
denom	denominator

Return Value

Returns the quotient and the remainder.

Remarks

The returned quotient will have the same sign as the numerator divided by the denominator. The sign for the remainder will be such that the quotient times the denominator plus the remainder will equal the numerator (`quot * denom + rem = numer`). If the denominator is zero, the behavior is undefined.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  long long x, y;
  lldiv_t z;

  x = 7;
  y = 3;
  printf("For lldiv(%lld, %lld)\n", x, y);
  z = lldiv(x, y);
  printf("The quotient is %lld and the "
         "remainder is %lld\n\n", z.quot, z.rem);

  x = 7;
  y = -3;
  printf("For lldiv(%lld, %lld)\n", x, y);
  z = lldiv(x, y);
  printf("The quotient is %lld and the "
         "remainder is %lld\n\n", z.quot, z.rem);

  x = -5;
  y = 3;
  printf("For lldiv(%lld, %lld)\n", x, y);
  z = lldiv(x, y);
  printf("The quotient is %lld and the "
         "remainder is %lld\n\n", z.quot, z.rem);

  x = 7;
  y = 7;
  printf("For lldiv(%lld, %lld)\n", x, y);
  z = lldiv(x, y);
  printf("The quotient is %lld and the "
         "remainder is %lld\n\n", z.quot, z.rem);

  x = 7;
  y = 0;
  printf("For lldiv(%lld, %lld)\n", x, y);
  z = lldiv(x, y);
  printf("The quotient is %lld and the "
         "remainder is %lld\n\n",
```

```
          z.quot, z.rem);
}
```

Example Output

```
For lldiv(7, 3)
The quotient is 2 and the remainder is 1

For lldiv(7, -3)
The quotient is -2 and the remainder is 1

For lldiv(-5, 3)
The quotient is -1 and the remainder is -2

For lldiv(7, 7)
The quotient is 1 and the remainder is 0

For lldiv(7, 0)
The quotient is -1 and the remainder is 7
```

Example Explanation

In the last example (`ldiv(7,0)`) the denominator is zero, and the behavior is undefined.

5.19.23 malloc Function

Dynamically allocate a block of memory from the heap.

Include

`<stdlib.h>`

Prototype

```
void * malloc(size_t size);
```

Argument

size	number of bytes to dynamically allocate

Return Value

Returns a pointer to the allocated memory if successful; otherwise, returns a null pointer.

Remarks

Although the memory block allocated and whose address is returned will be large enough to hold an object with the specified size, the memory required to allocate that block might be larger than the requested size due to alignment of the allocated memory, and/or additional objects that are used to manage the heap.

Unlike `calloc`, the `malloc` function does not initialize the memory it allocates.

See your MPLAB XC compiler User's Guide for more information on dynamic memory allocation.

Example

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

int main(void)
{
  long *i;

  if ((i = (long *)malloc(50 * sizeof(long))) == NULL)
    printf("Cannot allocate memory\n");
  else
  {
    printf("Memory allocated\n");
    free(i);
    printf("Memory freed\n");
  }
}
```

Example Output

```
Memory allocated
Memory freed
```

5.19.24 mblen Function

Determines the length of a multibyte character.

Attention: This function is not implemented by MPLAB XC8 C compilers.

Include

<wchar.h>

Prototype

```
int mblen(const char * s, size_t n);
```

Arguments

s	a pointer to the multibyte character to check
n	the maximum number of bytes to check

Return Value

If s is a null pointer, the function returns a nonzero or zero value, which indicates if multibyte character encodings, respectively, do or do not have state-dependent encodings.

If s is not a null pointer, the function returns 0 if s points to the null character, returns −1 if the processed bytes do not form a valid multibyte character, or returns the number of bytes of the converted multibyte character.

Remarks

The function determines the number of bytes taken up by the multibyte character pointed to by s.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  int len;
  char * mbs = "żoś";

  setlocale(LC_CTYPE, "");
  while(*mbs) {
    len = mblen(mbs, 4);
    printf("character length %d bytes\n", len);
    mbs += len;
  }
}
```

Example Output

```
character length 2 bytes
character length 1 bytes
character length 2 bytes
```

5.19.25 mbstowcs Function

Converts a multibyte character sequence to a wide string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<stdlib.h>`

Prototype

```
size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);
```

Arguments

pwcs	a pointer to the array to hold the converted multibyte sequence
s	the multibyte character sequence to convert
n	the maximum number of wide characters to store

Return Value

The function returns the number of wide characters, not including the terminating null character (if any), stored to `pwcs`. If `s` cannot be converted, `(size_t)(-1)` is returned.

Remarks

The function converts a sequence of multibyte characters in the array `s` and that begins in the initial shift state into a sequence of corresponding wide characters, storing no more than `n` converted wide characters (including a terminating null wide character) into the array pointed to by `pwcs`. Conversion stops when a terminating null character is encountered, when an invalid multibyte character sequence is encountered, or when `n` wide characters have been stored. Each conversion takes place as if by a call to the `mbtowc()` function, except that the conversion state of the `mbtowc()` function is not affected.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void printWide(const char * mbstr)
{
    size_t len;
    wchar_t wstr[40];

    len = mbstowcs(wstr, mbstr, 40);
    wprintf(L"Converted wide string \"%ls\" has length %u\n", wstr, len);
}

int main(void)
{
    setlocale(LC_CTYPE, "");
    printWide("It is 75°F");
}
```

Example Output

```
Converted wide string "It is 75°F" has length 10
```

5.19.26 mbtowc Function

Converts a multibyte character to a wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<stdlib.h>`

Prototype

```
int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);
```

Arguments

pwc	a pointer to where the converted multibyte character should be stored
s	the multibyte character to convert
n	The maximum number of bytes to consider for conversion

Return Value

If `s` is a null pointer, the function returns a nonzero or zero value, which indicates if multibyte character encodings, respectively, do or do not have state-dependent encodings.

If `s` is not a null pointer, the function returns 0 if `s` points to the null character, returns −1 if the processed bytes do not form a valid multibyte character, or returns the number of bytes of the converted multibyte character.

The returned value will never exceed `n` or the value of the `MB_CUR_MAX` macro.

Remarks

If a valid multibyte character (including any shift sequences) can be formed from no more than the next `n` bytes of the sequence pointed to by `s`, that multibyte character is converted to a wide character that is then stored to the object pointed to by `pwc`, provided that pointer is not null. If the corresponding wide character is the null wide character, the function is left in the initial conversion state.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

void printWide(const char * mbstr)
{
  size_t len;
  wchar_t wstr;

  mbtowc(NULL, 0, 0); // reset the conversion state
  while(*mbstr) {
    len = mbtowc(&wstr, mbstr, 40);
    wprintf(L"%lc(%d)", wstr, len);
    mbstr += len;
  }
  wprintf(L"\n");
}

int main(void)
{
    setlocale(LC_CTYPE, "");
    printWide("PIC®");
}
```

Example Output

```
P(1)I(1)C(1)®(2)
```

5.19.27 qsort Function

Sort an array of objects.

Include

<stdlib.h>

Prototype

```
void qsort(void * base, size_t nmemb, size_t size, int (* compar)(const void *, const
void *));
```

Argument

base	the array to sort
nmemb	the number of objects in the array
size	the size of each object being sorted
compar	the comparison function to be used

Remarks

The qsort function orders the elements in the base array so that they are in an ascending order, as specified by the function pointed to by compar. The comparison function should take two arguments, and should return an integer value less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int cmp (int *num1, int *num2)
{
  if (*num1 < *num2)
    return -1;
  if (*num1 > *num2)
    return 1;
  return 0;
}

int main (void)
{
  int array[] = {12, 92, 0, -1, 6, -24, 101};
  size_t arraylen = sizeof(array) / sizeof(int);

  qsort (array, arraylen, sizeof(int), (int (*)(const void *, const void *)) cmp);
  printf("Array elements:\n");
  for (int i=0; i != arraylen; i++) {
    printf("%d%s", array[i], i==arraylen-1 ? "\n" : " ");
  }
}
```

Example Output

```
Array elements:
-24 -1 0 6 12 92 101
```

5.19.28 rand Function

Generates a pseudo-random number.

Include

`<stdlib.h>`

Prototype

```
int rand(void);
```

Remarks

Successive calls to the `rand` function generates a pseudo-random number sequence, each number in the range 0 to `RAND_MAX`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int value;

  value = rand();
  printf("Today's lucky number is %d\n", value);
}
```

Example Output

```
Today's lucky number is 8073
```

5.19.29 realloc Function

Changes the size of an allocated block of memory.

Include

`<stdlib.h>`

Prototype

```
void *realloc(void * ptr, size_t size);
```

Arguments

ptr	pointer to object to deallocate
size	number of byte to allocate

Return Value

Returns a pointer to the allocated space if successful; otherwise, returns a null pointer.

Remarks

The function deallocates the memory consumed by the object pointed to by `ptr` and returns a pointer to a new object with the specified size and with the same contents of the old object prior to deallocation, up to the lesser of the previous and new sizes. If the size of the new object is larger than that of the previous object, the additional bytes have indeterminate values.

When using the simple implementation of dynamic memory allocation, reallocating memory will always result in a new allocation being made, regardless of the size of the new memory block.

If `ptr` is a null pointer, a call to `realloc()` behaves like one to `malloc()` with the specified size. The value of `ptr` must otherwise match a pointer returned by the `calloc()`, `malloc()`, or `realloc()` function, and the memory at that address must not have been deallocated by a call to the `free()` or `realloc()` function; otherwise, the

behavior is undefined. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

See your MPLAB XC compiler User's Guide for more information on dynamic memory allocation.

Example

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

int
main(void)
{
  int * block = NULL;  /* assigned NULL so first realloc works as malloc */
  int x;
  unsigned size = 0;

  while(size != 10) {
    x = rand();
    if(x < RAND_MAX / 1000) {
      size++;
      block = realloc(block, size);
      if(block == NULL)
        break;
      block[size-1] = x;
    }
  }
  do {
    printf("%d ", block[size-1]);
  } while(--size);
  printf("\n");
}
```

Example Output

```
456291 198011 524209 13456 84083 591308 86383 1887638 1277844 16807
```

5.19.30 srand Function

Specifies a seed to be used by subsequent calls to the `rand` function.

Include

`<stdlib.h>`

Prototype

```
int srand(unsigned int seed);
```

Argument

seed	the seed to begin the sequence

Remarks

The `srand` function allows the seed for a pseudo-random number sequence to be specified, so that subsequent calls to the `rand` function will return a sequence based on that seed. The same sequence can be repeated by calling `srand` again with the same seed value.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  int value;

  srand(0x100);
```

```
  value = rand();
  printf("Today's lucky number is %d\n", value);
}
```

Example Output

```
Today's lucky number is 663
```

5.19.31 strtod Function

Convert string to a double precision floating-point value.

Include

`<stdlib.h>`

Prototype

```
double strtod(const char * restrict nptr, char ** restrict endptr);
```

Arguments

nptr	the string to attempt to convert
endptr	pointer to the remainder of the string that was not converted

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The `strtod` function attempts to convert the first part of the string pointed to by `nptr` to a `double` floating-point value.

The initial section consists of any white-space characters.

The subject section represents the floating-point constant to convert and consists of an optional plus or minus sign then one of the following.

- Decimal digits optionally containing a decimal-point character, then an optional exponent part, being `e` or `E` followed by an option sign and decimal digits
- A `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part, being `p` or `P`, and an optional sign, and decimal digits.
- one of `INF` or `INFINITY`, ignoring case
- `NAN`, ignoring case, optionally followed by any sequence contain digits or non-digits:

Conversion stops once an unrecognized character is encountered, thus the subject sequence is defined as the longest subsequence of the input string after the initial section and that is of the expected form. A pointer to the position of the first unrecognizable character in the string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The final section consists of characters in the string that were unrecognized in the subject section and which will include the terminating null character of the string. This section of the string will be accessible via the address stored to `endptr`, provided `endptr` is not a null pointer.

The value of the macro `ERANGE` is stored in `errno` if the correct value of the conversion is outside the range of representable values. If the result underflows, ***whether `errno` acquires the value ERANGE is implementation-defined***

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
```

```
{
  char * string = " +0.137e2 mSec";
  char * final;
  double result;

  result = strtod(string, &final);
  printf("The floating-point conversion of the string \"%s\" is %g; final string part is
\"%s\"\n", string, result, final);
}
```

Example Output

```
The floating-point conversion of the string " +0.137e2 mSec" is 13.7; final string part is "
mSec"
```

5.19.32 strtof Function

Convert string to a single precision floating-point value.

Include

<stdlib.h>

Prototype

```
float strtof(const char * restrict nptr, char ** restrict endptr);
```

Argument

nptr	the string to attempt to convert
endptr	pointer to the remainder of the string that was not converted

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The `strtof` function attempts to convert the first part of the string pointed to by `nptr` to a `float` floating-point value.

The initial section consists of any white-space characters.

The subject section represents the floating-point constant to convert and consists of an optional plus or minus sign then one of the following.

- Decimal digits optionally containing a decimal-point character, then an optional exponent part, being `e` or `E` followed by an option sign and decimal digits
- A `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part, being `p` or `P`, and an optional sign, and decimal digits.
- one of `INF` or `INFINITY`, ignoring case
- `NAN`, ignoring case, optionally followed by any sequence contain digits or non-digits:

Conversion stops once an unrecognized character is encountered, thus the subject sequence is defined as the longest subsequence of the input string after the initial section and that is of the expected form. A pointer to the position of the first unrecognizable character in the string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The final section consists of characters in the string that were unrecognized in the subject section and which will include the terminating null character of the string. This section of the string will be accessible via the address stored to `endptr`, provided `endptr` is not a null pointer.

The value of the macro `ERANGE` is stored in `errno` if the correct value of the conversion is outside the range of representable values. If the result underflows, ***whether `errno` acquires the value ERANGE is implementation-defined***

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char * string = " +0.137e2 mSec";
  char * final;
  float result;

  result = strtof(string, &final);
  printf("The floating-point conversion of the string \"%s\" is %g; final string part is
\"%s\"\n", string, result, final);
}
```

Example Output

```
The floating-point conversion of the string " +0.137e2 mSec" is 13.7; final string part is "
mSec"
```

5.19.33 strtol Function

Convert string to long integer value.

Include

`<stdlib.h>`

Prototype

```
long int strtol( const char * restrict nptr, char ** restrict endptr, int base);
```

Arguments

nptr	the string to attempt to convert
endptr	pointer to the remainder of the string that was not converted
base	The base of the conversion

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The `strtol` function attempts to convert the first part of the string pointed to by `nptr` to a `long` integer value.

Any initial whitespace characters in the string are skipped. The following characters representing the integer are assumed to be in a radix specified by the `base` argument. Conversion stops once an unrecognized character is encountered in the string. If the correct converted value is out of range, the value of the macro `ERANGE` is stored in `errno`.

If the value of `base` is zero, the characters representing the integer can be in any valid C constant form (i.e., in decimal, octal, or hexadecimal), but any integer suffix is ignored. If the value of base is between 2 and 36 (inclusive), the expected form of the integer characters is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but again, the integer suffix is ignored. The letters from `a` (or `A`) through `z` (or `Z`) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters `0x` or `0X` may optionally precede the sequence of letters and digits, following the sign if present.

When using MPLAB XC8, the function will set `errno` to `EINVAL` and return 0 if `base` is invalid.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char * string = "-1234abcd";
  char * final;
  long result;

  result = strtol(string, &final, 10);
  printf("The integer conversion of the string \"%s\" is %ld; final string part is \"%s\"\n",
string, result, final);
}
```

Example Output

```
The integer conversion of the string "-1234abcd" is -1234; final string part is "abcd"
```

5.19.34 strtold Function

Convert string to long double precision floating-point value.

Include

`<stdlib.h>`

Prototype

`long double strtold(const char * restrict` **`nptr`**`, char ** restrict` **`endptr`**`);`

Argument

`nptr`	the string to attempt to convert
`endptr`	pointer to the remainder of the string that was not converted

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The `strtold` function attempts to convert the first part of the string pointed to by `nptr` to a `long double` floating-point value.

The initial section consists of any white-space characters.

The subject section represents the floating-point constant to convert and consists of an optional plus or minus sign then one of the following.

- Decimal digits optionally containing a decimal-point character, then an optional exponent part, being `e` or `E` followed by an option sign and decimal digits
- A `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part, being `p` or `P`, and an optional sign, and decimal digits.
- one of `INF` or `INFINITY`, ignoring case
- `NAN`, ignoring case, optionally followed by any sequence contain digits or non-digits:

Conversion stops once an unrecognized character is encountered, thus the subject sequence is defined as the longest subsequence of the input string after the initial section and that is of the expected form. A pointer to the position of the first unrecognizable character in the string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The final section consists of characters in the string that were unrecognized in the subject section and which will include the terminating null character of the string. This section of the string will be accessible via the address stored to `endptr`, provided `endptr` is not a null pointer.

The value of the macro ERANGE is stored in errno if the correct value of the conversion is outside the range of representable values. If the result underflows, ***whether errno acquires the value ERANGE is implementation-defined***

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  char * string = " +0.137e2 mSec";
  char * final;
  long double result;

  result = strtold(string, &final);
  printf("The floating-point conversion of the string \"%s\" is %Lg; final string part is
\"%s\"\n", string, result, final);
}
```

Example Output

```
The floating-point conversion of the string " +0.137e2 mSec" is 13.7; final string part is "
mSec"
```

5.19.35 strtoll Function

Convert string to long long integer value.

Include

<stdlib.h>

Prototype

```
long long int strtoll( const char * restrict nptr, char ** restrict endptr, int base);
```

Arguments

nptr	the string to attempt to convert
endptr	pointer to the remainder of the string that was not converted
base	The base of the conversion

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The strtoll function attempts to convert the first part of the string pointed to by nptr to a long long integer value.

Any initial whitespace characters in the string are skipped. The following characters representing the integer are assumed to be in a radix specified by the base argument. Conversion stops once an unrecognized character is encountered in the string. If the correct converted value is out of range, the value of the macro ERANGE is stored in errno.

If the value of base is zero, the characters representing the integer can be in any valid C constant form (i.e., in decimal, octal, or hexadecimal), but any integer suffix is ignored. If the value of base is between 2 and 36 (inclusive), the expected form of the integer characters is a sequence of letters and digits representing an integer with the radix specified by base, optionally preceded by a plus or minus sign, but again, the integer suffix is ignored. The letters from a (or A) through z (or Z) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

When using MPLAB XC8, the function will set `errno` to `EINVAL` and return 0 if `base` is invalid.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
  char * string = "-1234abcd";
  char * final;
  long long result;

  result = strtoll(string, &final, 10);
  printf("The integer conversion of the string \"%s\" is %lld; final string part is
\"%s\"\n", string, result, final);
}
```

Example Output

```
The integer conversion of the string "-1234abcd" is -1234; final string part is "abcd"
```

5.19.36 strtoul Function

Convert string to unsigned long integer value.

Include

`<stdlib.h>`

Prototype

`unsigned long int strtoul( const char * restrict` **`nptr`**`, char ** restrict` **`endptr`**`, int` **`base`**`);`

Arguments

`nptr`	the string to attempt to convert
`endptr`	pointer to the remainder of the string that was not converted
`base`	The base of the conversion

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The `strtoul` function attempts to convert the first part of the string pointed to by `nptr` to a `unsigned long` integer value.

Any initial whitespace characters in the string are skipped. The following characters representing the integer are assumed to be in a radix specified by the `base` argument. Conversion stops once an unrecognized character is encountered in the string. If the correct converted value is out of range, the value of the macro `ERANGE` is stored in `errno`.

If the value of `base` is zero, the characters representing the integer can be in any valid C constant form (i.e., in decimal, octal, or hexadecimal), but any integer suffix is ignored. If the value of base is between 2 and 36 (inclusive), the expected form of the integer characters is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but again, the integer suffix is ignored. The letters from `a` (or `A`) through `z` (or `Z`) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters `0x` or `0X` may optionally precede the sequence of letters and digits, following the sign if present.

When using MPLAB XC8, the function will set `errno` to `EINVAL` and return 0 if `base` is invalid.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char * string = "-1234abcd";
  char * final;
  unsigned long result;

  result = strtoul(string, &final, 10);
  printf("The integer conversion of the string \"%s\" is %lud; final string part is
\"%s\"\n", string, result, final);
}
```

Example Output

```
The integer conversion of the string "-1234abcd" is 4294966062d; final string part is "abcd"
```

5.19.37 strtoull Function

Convert string to unsigned long long integer value.

Include

`<stdlib.h>`

Prototype

`unsigned long long int strtoull( const char * restrict` **`nptr`**`, char ** restrict` **`endptr`**`, int` **`base`**`);`

Arguments

`nptr`	the string to attempt to convert
`endptr`	pointer to the remainder of the string that was not converted
`base`	The base of the conversion

Return Value

The converted value, or 0 if the conversion could not be performed.

Remarks

The `strtoull` function attempts to convert the first part of the string pointed to by `nptr` to a `unsinged long long` integer value.

Any initial whitespace characters in the string are skipped. The following characters representing the integer are assumed to be in a radix specified by the `base` argument. Conversion stops once an unrecognized character is encountered in the string. If the correct converted value is out of range, the value of the macro `ERANGE` is stored in `errno`.

If the value of `base` is zero, the characters representing the integer can be in any valid C constant form (i.e., in decimal, octal, or hexadecimal), but any integer suffix is ignored. If the value of base is between 2 and 36 (inclusive), the expected form of the integer characters is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but again, the integer suffix is ignored. The letters from `a` (or `A`) through `z` (or `Z`) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters `0x` or `0X` may optionally precede the sequence of letters and digits, following the sign if present.

When using MPLAB XC8, the function will set `errno` to `EINVAL` and return 0 if `base` is invalid.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char * string = "-1234abcd";
  char * final;
  unsigned long long result;

  result = strtoull(string, &final, 10);
  printf("The integer conversion of the string \"%s\" is %llu; final string part is
\"%s\"\n", string, result, final);
}
```

Example Output

```
The integer conversion of the string "-1234abcd" is 18446744073709550382; final string part
is "abcd"
```

5.19.38 system Function

Execute a command.

Attention: This function is not implemented by MPLAB XC8 C compilers.

Include

`<stdlib.h>`

Prototype

```
int system(const char * string);
```

Argument

`string`	command to be executed

Default Behavior

C STD: If string is a null pointer, the system function determines whether the host environment has a command processor. If string is not a null pointer, the system function passes the string pointed to by string to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling system to behave in a non-conforming manner or to terminate.

PREVIOUSLY: As distributed, this function acts as a stub or placeholder for your function. If `string` is not NULL, an error message is written to `stdout` and the program will reset; otherwise, a value of -1 is returned.

5.19.39 wcstombs Function

Converts a wide character string to a multibyte character sequence.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<stdlib.h>`

Prototype

```
size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
```

Arguments

s	a pointer to the object to hold the converted multibyte character sequence
pwcs	the wide character sequence to convert
n	the maximum number of bytes to store in `s`

Return Value

The function returns the number of bytes in the multibyte character sequence, not including the terminating null character (if any). If `s` cannot be converted to a mutlibyte character sequence, `(size_t)(-1)` is returned.

Remarks

The function converts a wide character string indirectly pointed to by `pwcs` beginning in the initial shift state into a sequence of corresponding multibyte characters, storing no more than `n` converted characters (including a terminating null character) into the array pointed to by `s`, provided `dst` is not a null pointer. Each conversion takes place as if by a call to the `wctomb()` function, except that the conversion state of the that function is not affected.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void) {
  const wchar_t * ws = L"Volume: 2L±0.5mL";
  char buffer[20];
  size_t length;

  setlocale(LC_CTYPE, "");

  length = wcstombs(buffer, ws, 20);
  wprintf(L"multibyte string \"%s\" has length %d\n", buffer, length);
}
```

Example Output

```
multibyte string "Volume: 2L±0.5mL" has length 17
```

5.19.40 wctomb Function

Converts a wide character to a multibyte character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

```
<stdlib.h>
```

Prototype

```
int wctomb(char * s, wchar_t wc);
```

Arguments

s	points to the multibyte character
wc	the wide character to be converted

Return Value

Returns zero if `s` points to a null character; otherwise, returns 1.

Remarks

The resulting multibyte character is stored at `s`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

#define SIZE 40

int main(void)
{
  static char  buffer[SIZE];
  wchar_t * wch = L"PIC®";
  int length, idx;

  setlocale(LC_CTYPE, "");
  while(wch[idx]) {
    length = wctomb(buffer, wch[idx]);
    printf( "bytes in multibyte %lc is %i\n", wch[idx], length);
    printf( "converted string \"%s\"\n", buffer);
    idx++;
  }
}
```

Example Output

```
bytes in multibyte P is 1
converted string "P"
bytes in multibyte I is 1
converted string "I"
bytes in multibyte C is 1
converted string "C"
bytes in multibyte ® is 2
converted string "®"
```

5.20 <string.h> String Functions

The header file, `string.h`, consists of types, macros and functions that provide tools to manipulate strings.

5.20.1 size_t Type

An unsigned integer type used by the result of the `sizeof` operator

Include

`<stddef.h>`

`<stdio.h>`

`<stdlib.h>`

`<string.h>`

`<time.h>`

`<wchar.h>`

Definition

```
typedef unsigned size_t;
```

5.20.2 NULL Macro

A constant value which represent a null pointer constant. It's value and type is implementation defined.

Include

<locale.h>

<stddef.h>

<stdio.h>

<stdlib.h>

<string.h>

<time.h>

<wchar.h>

Definition

```
#define NULL ((void*)0)
```

5.20.3 memchr Function

Locates a character in a buffer.

Include

<string.h>

Prototype

```
void *memchr(const void *s, int c, size_t n);
```

Arguments

s	pointer to the buffer
c	character to search for
n	number of characters to check

Return Value

Returns a pointer to the location of the match if successful; otherwise, returns NULL.

Remarks

memchr stops when it finds the first occurrence of c, or after searching n number of characters.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'i', ch2 = 'y';
  char *ptr;
  int res;
  printf("buf1 : %s\n\n", buf1);

  ptr = memchr(buf1, ch1, 50);
if (ptr != NULL)
  {
   res = ptr - buf1 + 1;
   printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);
```

```
  printf("\n");

  ptr = memchr(buf1, ch2, 50);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

Example Output

```
buf1 : What time is it?

i found at position 7

y not found
```

5.20.4 memcmp Function

Compare the contents of two buffers.

Include

`<string.h>`

Prototype

`int memcmp(const void *`**`s1`**`, const void *`**`s2`**`, size_t `**`n`**`);`

Arguments

s1	first buffer
s2	second buffer
n	number of characters to compare

Return Value

Returns a positive number if `s1` is greater than `s2`, zero if `s1` is equal to `s2` or a negative number if `s1` is less than `s2`.

Remarks

This function compares the first `n` characters in `s1` to the first `n` characters in `s2` and returns a value indicating whether the buffers are less than, equal to or greater than each other.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = memcmp(buf1, buf2, 6);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
```

```
    printf("6 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");
  printf("\n");

  res = memcmp(buf1, buf2, 20);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");
  printf("\n");

  res = memcmp(buf1, buf3, 20);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf3 "
           "are equal\n");
  else
    printf("buf3 comes before buf1\n");
}
```

Example Output

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

5.20.5 memcpy Function

Copies characters from one buffer to another.

Include

`<string.h>`

Prototype

```
void *memcpy(void *dst, const void *src, size_t n);
```

Arguments

dst	buffer to copy characters to
src	buffer to copy characters from
n	number of characters to copy

Return Value

Returns `dst`.

Remarks

`memcpy` copies n characters from the source buffer `src` to the destination buffer `dst`. If the buffers overlap, the behavior is undefined.

For MPLAB XC16 functions that can copy to/from specialized memory areas, such as `memcpy_eds` or `memcpy_p2d1624`, see “Functions for Specialized Copying and Initialization” in the *MPLAB XC16 Libraries Reference Guide*.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  memcpy(buf1, buf2, 6);
  printf("buf1 after memcpy of 6 chars of "
         "buf2: \n\t%s\n", buf1);

  printf("\n");

  memcpy(buf1, buf3, 5);
  printf("buf1 after memcpy of 5 chars of "
         "buf3: \n\t%s\n", buf1);
}
```

Example Output

```
buf1 :
buf2 : Where is the time?
buf3 : Why?

buf1 after memcpy of 6 chars of buf2:
        Where

buf1 after memcpy of 5 chars of buf3:
        Why?
```

5.20.6 memmove Function

Copies n characters of the source buffer into the destination buffer, even if the regions overlap.

Include

`<string.h>`

Prototype

```
void *memmove(void *s1, const void *s2, size_t n);
```

Arguments

s1	buffer to copy characters to (destination)
s2	buffer to copy characters from (source)
n	number of characters to copy from s2 to s1

Return Value

Returns a pointer to the destination buffer.

Remarks

If the buffers overlap, the effect is as if the characters are read first from `s2`, then written to `s1`, so the buffer is not corrupted.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "When time marches on";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  memmove(buf1, buf2, 6);
  printf("buf1 after memmove of 6 chars of "
         "buf2: \n\t%s\n", buf1);

  printf("\n");

  memmove(buf1, buf3, 5);
  printf("buf1 after memmove of 5 chars of "
         "buf3: \n\t%s\n", buf1);
}
```

Example Output

```
buf1 : When time marches on
buf2 : Where is the time?
buf3 : Why?

buf1 after memmove of 6 chars of buf2:
        Where ime marches on

buf1 after memmove of 5 chars of buf3:
        Why?
```

5.20.7 memset Function

Copies the specified character into the destination buffer.

Include

`<string.h>`

Prototype

```
void *memset(void *s, int c, size_t n);
```

Arguments

s	buffer
c	character to put in buffer
n	number of times

Return Value

Returns the buffer with characters written to it.

Remarks

The character `c` is written to the buffer `n` times.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[20] = "What time is it?";
  char buf2[20] = "";
  char ch1 = '?', ch2 = 'y';
  char *ptr;
  int res;

  printf("memset(\"%s\", \'%c\',4);\n", buf1, ch1);
  memset(buf1, ch1, 4);
  printf("buf1 after memset: %s\n", buf1);

  printf("\n");
  printf("memset(\"%s\", \'%c\',10);\n", buf2, ch2);
  memset(buf2, ch2, 10);
  printf("buf2 after memset: %s\n", buf2);
}
```

Example Output

```
memset("What time is it?", '?',4);
buf1 after memset: ???? time is it?

memset("", 'y',10);
buf2 after memset: yyyyyyyyyy
```

5.20.8 strcat Function

Appends a copy of the source string to the end of the destination string.

Include

`<string.h>`

Prototype

```
char *strcat(char *s1, const char *s2);
```

Arguments

s1	null terminated destination string to copy to
s2	null terminated source string to be copied

Return Value

Returns a pointer to the destination string.

Remarks

This function appends the source string (including the terminating null character) to the end of the destination string. The initial character of the source string overwrites the null character at the end of the destination string. If the buffers overlap, the behavior is undefined.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
```

```
  printf("buf1 : %s\n", buf1);
  printf("\t(%d characters)\n\n", strlen(buf1));
  printf("buf2 : %s\n", buf2);
  printf("\t(%d characters)\n\n", strlen(buf2));

  strcat(buf1, buf2);
  printf("buf1 after strcat of buf2: \n\t%s\n",
          buf1);
  printf("\t(%d characters)\n", strlen(buf1));
  printf("\n");

  strcat(buf1, "Why?");
  printf("buf1 after strcat of \"Why?\": \n\t%s\n",
         buf1);
  printf("\t(%d characters)\n", strlen(buf1));
}
```

Example Output

```
buf1 : We're here
        (10 characters)

buf2 : Where is the time?
        (18 characters)

buf1 after strcat of buf2:
        We're hereWhere is the time?
        (28 characters)

buf1 after strcat of "Why?":
        We're hereWhere is the time?Why?
        (32 characters)
```

5.20.9 strchr Function

Locates the first occurrence of a specified character in a string.

Include

`<string.h>`

Prototype

`char *strchr(const char *s, int c);`

Arguments

s	pointer to the string
c	character to search for

Return Value

Returns a pointer to the location of the match if successful; otherwise, returns a null pointer.

Remarks

This function searches the string `s` to find the first occurrence of the character, `c`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'm', ch2 = 'y';
  char *ptr;
  int res;
```

```
  printf("buf1 : %s\n\n", buf1);

  ptr = strchr(buf1, ch1);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);
  printf("\n");

  ptr = strchr(buf1, ch2);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

Example Output

```
buf1 : What time is it?

m found at position 8

y not found
```

5.20.10 strcmp Function

Compares two strings.

Include

`<string.h>`

Prototype

`int strcmp(const char *`**`s1`**`, const char *`**`s2`**`);`

Arguments

`s1`	first string
`s2`	second string

Return Value

Returns a positive number if `s1` is greater than `s2`, zero if `s1` is equal to `s2` or a negative number if `s1` is less than `s2`.

Remarks

This function compares successive characters from `s1` and `s2` until they are not equal or the null terminator is reached.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;
```

```
  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = strcmp(buf1, buf2);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("buf1 and buf2 are equal\n");
  else
    printf("buf2 comes before buf1\n");
  printf("\n");

  res = strcmp(buf1, buf3);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("buf1 and buf3 are equal\n");
  else
    printf("buf3 comes before buf1\n");
  printf("\n");

  res = strcmp("Why?", buf3);
  if (res < 0)
    printf("\"Why?\" comes before buf3\n");
  else if (res == 0)
    printf("\"Why?\" and buf3 are equal\n");
  else
    printf("buf3 comes before \"Why?\"\n");
}
```

Example Output

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

buf2 comes before buf1

buf1 comes before buf3

"Why?" and buf3 are equal
```

5.20.11 strcoll Function

Compares one string to another.

Include

`<string.h>`

Prototype

`int strcoll(const char * **s1**, const char * **s2**);`

Arguments

s1	first string
s2	second string

Return Value

Using the locale-dependent rules, it returns a positive number if `s1` is greater than `s2`, zero if `s1` is equal to `s2` or a negative number if `s1` is less than `s2`.

Remarks

Compares the two string arguments, subject to the `LC_COLLATE` category of the current locale. When locales are not supported, this function calls `strcmp`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  int res;

  setlocale(LC_CTYPE, "");
  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  res = strcoll(buf1, buf2);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("buf1 and buf2 are equal\n");
  else
    printf("buf2 comes before buf1\n");
}
```

Example Output

```
buf1 : Where is the time?
buf2 : Where did they go?
buf2 comes before buf1
```

5.20.12 strcpy Function

Copy the source string into the destination string.

Include

`<string.h>`

Prototype

```
char *strcpy(char *s1, const char *s2);
```

Arguments

s1	destination string to copy to
s2	source string to copy from

Return Value

Returns a pointer to the destination string.

Remarks

All characters of `s2` are copied, including the null terminating character. If the strings overlap, the behavior is undefined.

For MPLAB XC16 functions that can copy to/from specialized memory areas, such as `strcpy_eds` or `strcpy_packed`, see “Functions for Specialized Copying and Initialization” in the *MPLAB XC16 Libraries Reference Guide*.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
```

```
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  strcpy(buf1, buf2);
  printf("buf1 after strcpy of buf2: \n\t%s\n\n",
         buf1);

  strcpy(buf1, buf3);
  printf("buf1 after strcpy of buf3: \n\t%s\n",
          buf1);
}
```

Example Output

```
buf1 : We're here
buf2 : Where is the time?
buf3 : Why?

buf1 after strcpy of buf2:
        Where is the time?

buf1 after strcpy of buf3:
        Why?
```

5.20.13 strcspn Function

Calculate the number of consecutive characters at the beginning of a string that are not contained in a set of characters.

Include

`<string.h>`

Prototype

```
size_t strcspn(const char *s1, const char *s2);
```

Arguments

s1	pointer to the string to be searched
s2	pointer to characters to search for

Return Value

Returns the length of the segment in `s1` not containing characters found in `s2`.

Remarks

This function will determine the number of consecutive characters from the beginning of `s1` that are not contained in `s2`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char str1[20] = "hello";
  char str2[20] = "aeiou";
  char str3[20] = "animal";
  char str4[20] = "xyz";
  int res;
```

```
  res = strcspn(str1, str2);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str1, str2, res);

  res = strcspn(str3, str2);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str3, str2, res);

  res = strcspn(str3, str4);
  printf("strcspn(\"%s\", \"%s\") = %d\n",
          str3, str4, res);
}
```

Example Output

```
strcspn("hello", "aeiou") = 1
strcspn("animal", "aeiou") = 0
strcspn("animal", "xyz") = 6
```

Example Explanation

In the first result, `e` is in `s2` so it stops counting after `h`.

In the second result, `a` is in `s2`.

In the third result, none of the characters of `s1` are in `s2` so all characters are counted.

5.20.14 strerror Function

Gets an internal error message.

Include

`<string.h>`

Prototype

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

Argument

`errcode`	number of the error code

Return Value

Returns a pointer to an internal error message string corresponding to the specified error code `errcode`.

Remarks

The array pointed to by `strerror` may be overwritten by a subsequent call to this function, so it is not thread safe.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main()
{
    double result, x=-9.0;

    result = log(x);
    if(errno)
        printf("Log generated %s\n", strerror(errno));
    else
        printf("Log of %g is %g\n", x, result);
}
```

Example Output

```
Log generated Mathematics argument out of domain of function
```

5.20.15 strlen Function

Finds the length of a string.

Include

<string.h>

Prototype

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

Argument

s	the string

Return Value

Returns the length of a string.

Remarks

This function determines the length of the string, not including the terminating null character.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char str1[20] = "We are here";
  char str2[20] = "";
  char str3[20] = "Why me?";

  printf("str1 : %s\n", str1);
  printf("\t(string length = %d characters)\n\n",
         strlen(str1));
  printf("str2 : %s\n", str2);
  printf("\t(string length = %d characters)\n\n",
         strlen(str2));
  printf("str3 : %s\n", str3);
  printf("\t(string length = %d characters)\n\n\n",
         strlen(str3));

}
```

Example Output

```
str1 : We are here
        (string length = 11 characters)

str2 :
        (string length = 0 characters)

str3 : Why me?
        (string length = 7 characters)
```

5.20.16 strncat Function

Append a specified number of characters from the source string to the destination string.

Include

<string.h>

Prototype

```
char *strncat(char *s1, const char *s2, size_t n);
```

Arguments

s1	destination string to copy to
s2	source string to copy from
n	maximum number of characters to append

Return Value

Returns a pointer to the destination string.

Remarks

This function appends up to `n` characters (a null character and characters that follow it are not appended) from the source string to the end of the destination string. If a null character is not encountered, then a terminating null character is appended to the result. If the strings overlap, the behavior is undefined.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";

  printf("buf1 : %s\n", buf1);
  printf("\t(%d characters)\n\n", strlen(buf1));
  printf("buf2 : %s\n", buf2);
  printf("\t(%d characters)\n\n", strlen(buf2));
  printf("buf3 : %s\n", buf3);
  printf("\t(%d characters)\n\n\n", strlen(buf3));

  strncat(buf1, buf2, 6);
  printf("buf1 after strncat of 6 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));
  printf("\n");

  strncat(buf1, buf2, 25);
  printf("buf1 after strncat of 25 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));
  printf("\n");

  strncat(buf1, buf3, 4);
  printf("buf1 after strncat of 4 characters "
         "of buf3: \n\t%s\n", buf1);
  printf("\t(%d characters)\n", strlen(buf1));
}
```

Example Output

```
buf1 : We’re here
        (10 characters)

buf2 : Where is the time?
        (18 characters)

buf3 : Why?
        (4 characters)

buf1 after strncat of 6 characters of buf2:
```

```
        We’re hereWhere
        (16 characters)

buf1 after strncat of 25 characters of buf2:
        We’re hereWhere Where is the time?
        (34 characters)

buf1 after strncat of 4 characters of buf3:
   We’re hereWhere Where is the time?Why?
        (38 characters)
```

5.20.17 strncmp Function

Compare two strings, up to a specified number of characters.

Include

`<string.h>`

Prototype

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Arguments

s1	first string
s2	second string
n	maximum number of characters to compare

Return Value

Returns a positive number if `s1` is greater than `s2`, zero if `s1` is equal to `s2` or a negative number if `s1` is less than `s2`.

Remarks

strncmp returns a value based on the first character that differs between `s1` and `s2`. Characters that follow a null character are not compared.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "Where is the time?";
  char buf2[50] = "Where did they go?";
  char buf3[50] = "Why?";
  int res;

  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n\n", buf3);

  res = strncmp(buf1, buf2, 6);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
    printf("6 characters of buf1 and buf2 "
           "are equal\n");
  else
   printf("buf2 comes before buf1\n");
  printf("\n");

  res = strncmp(buf1, buf2, 20);
  if (res < 0)
    printf("buf1 comes before buf2\n");
  else if (res == 0)
```

```
    printf("20 characters of buf1 and buf2 "
           "are equal\n");
  else
    printf("buf2 comes before buf1\n");
  printf("\n");

  res = strncmp(buf1, buf3, 20);
  if (res < 0)
    printf("buf1 comes before buf3\n");
  else if (res == 0)
    printf("20 characters of buf1 and buf3 "
           "are equal\n");
else
    printf("buf3 comes before buf1\n");
}
```

Example Output

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

5.20.18 strncpy Function

Copy the source string into the destination string, up to the specified number of characters.

Include

`<string.h>`

Prototype

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Arguments

s1	destination string to copy to
s2	source string to copy from
n	number of characters to copy

Return Value

Returns a pointer to the destination string.

Remarks

Copies `n` characters from the source string to the destination string. If the source string is less than `n` characters, the destination is filled with null characters to total `n` characters. If `n` characters were copied and no null character was found, then the destination string will not be null-terminated. If the strings overlap, the behavior is undefined.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "We're here";
  char buf2[50] = "Where is the time?";
  char buf3[50] = "Why?";
  char buf4[7]  = "Where?";
```

```
  printf("buf1 : %s\n", buf1);
  printf("buf2 : %s\n", buf2);
  printf("buf3 : %s\n", buf3);
  printf("buf4 : %s\n", buf4);

  strncpy(buf1, buf2, 6);
  printf("buf1 after strncpy of 6 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));
  printf("\n");

  strncpy(buf1, buf2, 18);
  printf("buf1 after strncpy of 18 characters "
         "of buf2: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));
  printf("\n");

  strncpy(buf1, buf3, 5);
  printf("buf1 after strncpy of 5 characters "
         "of buf3: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));
  printf("\n");

  strncpy(buf1, buf4, 9);
  printf("buf1 after strncpy of 9 characters "
         "of buf4: \n\t%s\n", buf1);
  printf("\t( %d characters)\n", strlen(buf1));
}
```

Example Output

```
buf1 : We're here
buf2 : Where is the time?
buf3 : Why?
buf4 : Where?
buf1 after strncpy of 6 characters of buf2:
        Where here
        ( 10 characters)

buf1 after strncpy of 18 characters of buf2:
        Where is the time?
        ( 18 characters)

buf1 after strncpy of 5 characters of buf3:
        Why?
        ( 4 characters)

buf1 after strncpy of 9 characters of buf4:
        Where?
        ( 6 characters)
```

Example Explanation

Each buffer contains the string shown, followed by null characters for a length of 50. Using `strlen` will find the length of the string up to, but not including, the first null character.

In the first example, 6 characters of `buf2` (“Where “) replace the first 6 characters of `buf1` ("We’re ") and the rest of `buf1` remains the same ("here" plus null characters).

In the second example, 18 characters replace the first 18 characters of `buf1` and the rest remain null characters.

In the third example, 5 characters of `buf3` ("Why?" plus a null terminating character) replace the first 5 characters of `buf1`. `buf1` now actually contains ("Why?", 1 null character, " is the time?", 32 null characters). `strlen` shows 4 characters because it stops when it reaches the first null character.

In the fourth example, since `buf4` is only 7 characters, `strncpy` uses 2 additional null characters to replace the first 9 characters of `buf1`. The result of `buf1` is 6 characters ("Where?") followed by 3 null characters, followed by 9 characters ("the time?"), followed by 32 null characters.

5.20.19 strpbrk Function

Search a string for the first occurrence of a character from a specified set of characters.

Include

`<string.h>`

Prototype

```
char *strpbrk(const char *s1, const char *s2);
```

Arguments

s1	pointer to the string to be searched
s2	pointer to characters to search for

Return Value

Returns a pointer to the matched character in `s1` if found; otherwise, returns a null pointer.

Remarks

This function will search `s1` for the first occurrence of a character contained in `s2`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char str1[20] = "What time is it?";
  char str2[20] = "xyz";
  char str3[20] = "eou?";
  char *ptr;
  int res;

  printf("strpbrk(\"%s\", \"%s\")\n", str1, str2);
  ptr = strpbrk(str1, str2);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("match found at position %d\n", res);
  }
  else
    printf("match not found\n");
  printf("\n");

  printf("strpbrk(\"%s\", \"%s\")\n", str1, str3);
  ptr = strpbrk(str1, str3);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("match found at position %d\n", res);
  }
  else
    printf("match not found\n");
}
```

Example Output

```
strpbrk("What time is it?", "xyz")
match not found

strpbrk("What time is it?", "eou?")
match found at position 9
```

5.20.20 strrchr Function

Search for the last occurrence of a specified character in a string.

Include

<string.h>

Prototype

```
char *strrchr(const char *s, int c);
```

Arguments

s	pointer to the string
c	character to search for

Return Value

Returns a pointer to the location of the match if successful; otherwise, returns a null pointer.

Remarks

This function searches the string `s` to find the last occurrence of the character, `c`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char buf1[50] = "What time is it?";
  char ch1 = 'm', ch2 = 'y';
  char *ptr;
  int res;

  printf("buf1 : %s\n\n", buf1);

  ptr = strrchr(buf1, ch1);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch1, res);
  }
  else
    printf("%c not found\n", ch1);
  printf("\n");

  ptr = strrchr(buf1, ch2);
if (ptr != NULL)
  {
    res = ptr - buf1 + 1;
    printf("%c found at position %d\n", ch2, res);
  }
  else
    printf("%c not found\n", ch2);
}
```

Example Output

```
buf1 : What time is it?

m found at position 8

y not found
```

5.20.21 strspn Function

Calculate the number of consecutive characters at the beginning of a string that are contained in a set of characters.

Include

<string.h>

Prototype

```
size_t strspn(const char *s1, const char *s2);
```

Arguments

s1	pointer to the string to be searched
s2	pointer to characters to search for

Return Value

Returns the number of consecutive characters from the beginning of `s1` that are contained in `s2`.

Remarks

This function stops searching when a character from `s1` is not in `s2`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char str1[20] = "animal";
  char str2[20] = "aeiounm";
  char str3[20] = "aimnl";
  char str4[20] = "xyz";
  int res;

  res = strspn(str1, str2);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str2, res);

  res = strspn(str1, str3);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str3, res);

  res = strspn(str1, str4);
  printf("strspn(\"%s\", \"%s\") = %d\n",
          str1, str4, res);
}
```

Example Output

```
strspn("animal", "aeiounm") = 5
strspn("animal", "aimnl") = 6
strspn("animal", "xyz") = 0
```

Example Explanation

In the first result, `l` is not in `s2`.

In the second result, the terminating null is not in `s2`.

In the third result, `a` is not in `s2` , so the comparison stops.

5.20.22 strstr Function

Search for the first occurrence of a string inside another string.

Include

```
<string.h>
```

Prototype

```
char *strstr(const char *s1, const char *s2);
```

Arguments

s1	pointer to the string to be searched
s2	pointer to substring to be searched for

Return Value

Returns the address of the first element that matches the substring if found; otherwise, returns a null pointer.

Remarks

This function will find the first occurrence of the string `s2` (excluding the null terminator) within the string `s1`. If `s2` points to a zero length string, `s1` is returned.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char str1[20] = "What time is it?";
  char str2[20] = "is";
  char str3[20] = "xyz";
  char *ptr;
  int res;

  printf("str1 : %s\n", str1);
  printf("str2 : %s\n", str2);
  printf("str3 : %s\n\n", str3);

  ptr = strstr(str1, str2);
if (ptr != NULL)
  {
   res = ptr - str1 + 1;
   printf("\"%s\" found at position %d\n",
          str2, res);
  }
  else
    printf("\"%s\" not found\n", str2);
  printf("\n");

  ptr = strstr(str1, str3);
if (ptr != NULL)
  {
    res = ptr - str1 + 1;
    printf("\"%s\" found at position %d\n",
           str3, res);
  }
  else
    printf("\"%s\" not found\n", str3);
}
```

Example Output

```
str1 : What time is it?
str2 : is
str3 : xyz

"is" found at position 11

"xyz" not found
```

5.20.23 strtok Function

Break a string into substrings, or tokens, by inserting null characters in place of specified delimiters.

Include

<string.h>

Prototype

```
char *strtok(char *s1, const char *s2);
```

Arguments

s1	pointer to the null terminated string to be searched
s2	pointer to characters to be searched for (used as delimiters)

Return Value

Returns a pointer to the first character of a token (the first character in a sequence of characters in `s1` that does not appear in the set of characters of `s2`). If no token is found, the null pointer is returned.

Remarks

A sequence of calls to this function can be used to split up a string into substrings (or tokens) by replacing specified characters with null characters. The first time this function is invoked on a particular string, that string should be passed in `s1`. After the first time, this function can continue parsing the string from the last delimiter by invoking it with a null value passed in `s1`.

It skips all leading characters that appear in the string `s2` (delimiters), then skips all characters not appearing in `s2` (this segment of characters is the token), and then overwrites the next character with a null character, terminating the current token. The function, `strtok`, then saves a pointer to the character that follows, from which the next search will start. If `strtok` finds the end of the string before it finds a delimiter, the current token extends to the end of the string pointed to by `s1`. If that was the first call to `strtok`, it does not modify the string (no null characters are written to `s1`). The set of characters that is passed in `s2` need not be the same for each call to `strtok`.

If `strtok` is called with a non-null parameter for `s1` after the initial call, the string becomes the new string to search. The old string previously searched will be lost.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char str1[30] = "Here, on top of the world!";
  char delim[5] = ", .";
  char *word;
  int x;

  printf("str1 : %s\n", str1);
  x = 1;
  word = strtok(str1,delim);
  while (word != NULL)
  {
    printf("word %d: %s\n", x++, word);
    word = strtok(NULL, delim);
  }
}
```

Example Output

```
str1 : Here, on top of the world!
word 1: Here
word 2: on
word 3: top
word 4: of
word 5: the
word 6: world!
```

5.20.24 strxfrm Function

Transforms a string using the locale-dependent rules.

Include

`<string.h>`

Prototype

```
size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

Arguments

s1	destination string
s2	source string to be transformed
n	number of characters to transform

Return Value

Returns the length of the transformed string not including the terminating null character. If `n` is zero, the string is not transformed (`s1` may be a point null in this case) and the length of `s2` is returned.

Remarks

The function transforms a string such that if two transformed strings are compared using `strcmp`, the result would be the same as comparing the original strings using the `strcoll` function. As no locales other than the `"C"` locale are supported, the transformation is equivalent to `strcpy`, except that the length of the destination string is bounded by `n`-1.

5.21 <tgmath.h> Type-generic Math

Attention: This header is not implemented when building with MPLAB XC8 for PIC MCUs.

The header file `tgmath.h` provides mappings from a generic math macro to either a real or complex math function, defined by `<math.h>` or `<complex.h>`. They allows a specific variant of some math functions to be called, based on the type of the arguments passed to the macro. The `<math.h>` and `<complex.h>` headers are included by `<tgmath.h>`.

Real functions in `<math.h>` come in three variants: an unsuffixed function, and a `float` and a `long double` variant, indicated by an `f` and `l` suffix in the function name, for example, `sin()`, `sinf()`, and `sinl()`. Complex functions in `<complex.h>` all use a `c` prefix and again come in an unsuffixed variant and a `float complex` and a `long double complex` variant, indicated by an `f` and `l` suffix in the function name, for example `cexp()`, `cexpf()`, and `cexpl()`.

For all of these real and complex functions, except for the `modf()` function, a type-generic macro is defined by `<tgmath.h>`. When a type-generic macro is used, the following rules determine the called function.

If there are only `c`-prefixed complex function variants of the macro, then one of those function variants is called, regardless of the type of the argument. Where there are real variants of the function, the following rules apply.

If any argument has `long double complex` type and the `c`-prefixed complex variant of the function exists, the function using a `c` prefix and `l` suffix will be called. If the `c`-prefixed variant of the function does not exist, the behavior is undefined. Otherwise, if any argument has `long double` type, the function uses an `l` suffix will be called. For example using the `acos()` type-generic macro with a `long double` argument will call the `acosl()` function; using the `acos()` type-generic macro with a `long double complex` argument will call the `cacosl()` function.

If any argument has `double complex` type and the `c`-prefixed complex variant of the function exists, the function using a `c` prefix and no suffix will be called. If the `c`-prefixed variant of the function does not exist, the behavior is

undefined. Otherwise, if any argument has `double` or integer type, the function with no suffix will be called. For example using the `acos()` type-generic macro with a `int` argument will call the `acos()` function; using the `acos()` type-generic macro with a `double complex` argument will call the `cacos()` function.

If any argument has `float complex` type and the `c`-prefixed complex variant of the function exists, the function using a `c` prefix and `f` suffix will be called. If the `c`-prefixed variant of the function does not exist, the behavior is undefined. Otherwise, if any argument has `float` type, the function uses an `f` suffix will be called. For example using the `acos()` type-generic macro with a `float` argument will call the `acosf()` function; using the `acos()` type-generic macro with a `float complex` argument will call the `cacosf()` function.

The application of these rules for all type-generic macros are indicated in the table below.

Table 5-5. Mappings from type-generic macros to <math.h> and <complex.h> functions

Type-generic macro used	Called function variants with all real arguments	Called function variants with at least one complex argument
acos	acos, acosf, acosl	cacos, cacosf, cacosl
asin	asin, asinf, asinl	casin, casinf, casinl
atan	atan, atanf, atanl	catan, catanf, catanl
acosh	acosh, acoshf, acoshl	cacosh, cacoshf, cacoshl
asinh	asinh, asinhf, asinhl	casinh, casinhf, casinhl
atanh	atanh, atanhf, atanhl	catanh, catanhf, catanhl
cos	cos, cosf, cosl	ccos, ccosf, ccosl
sin	sin, sinf, sinl	csin, csinf, csinl
tan	tan, tanf, tanl	ctan, ctanf, ctanl
cosh	cosh, coshf, coshl	ccosh, ccoshf, ccoshl
sinh	sinh, sinhf, sinhl	csinh, csinhf, csinhl
tanh	tanh, tanhf, tanhl	ctanh, ctanhf, ctanhl
exp	exp, expf, expl	cexp, cexpf, cexpl
log	log, logf, logl	clog, clogf, clogl
pow	pow, powf, powl	cpow, cpowf, cpowl
sqrt	sqrt, sqrtf, sqrtl	csqrt, csqrtf, csqrtl
fabs	fabs, fabsf, fabsl	cfabs, cfabsf, cfabsl
atan2	atan2, atan2f, atan2l	undefined behavior
cbrt	cbrt, cbrtf, cbrtl	undefined behavior
ceil	ceil, ceilf, ceill	undefined behavior
copysign	copysign, copysignf, copysignl	undefined behavior
erf	erf, erff, erfl	undefined behavior
erfc	erfc, erfcf, erfcl	undefined behavior
exp2	exp2, exp2f, exp2l	undefined behavior
expm1	expm1, expm1f, expm1l	undefined behavior

.....continued

Type-generic macro used	Called function variants with all real arguments	Called function variants with at least one complex argument
fdim	fdim, fdimf, fdiml	undefined behavior
floor	floor, floorf, floorl	undefined behavior
fma	fma, fmaf, fmal	undefined behavior
fmax	fmax, fmaxf, fmaxl	undefined behavior
fmin	fmin, fminf, fminl	undefined behavior
fmod	fmod, fmodf, fmodl	undefined behavior
frexp	frexp, frexpf, frexpl	undefined behavior
hypot	hypot, hypotf, hypotl	undefined behavior
ilogb	ilogb, ilogbf, ilogbl	undefined behavior
ldexp	ldexp, ldexpf, ldexpl	undefined behavior
lgamma	lgamma, lgammaf, lgammal	undefined behavior
llrint	llrint, llrintf, llrintl	undefined behavior
llround	llround, llroundf, llroundl	undefined behavior
log10	log10, log10f, log10l	undefined behavior
log1p	log1p, log1pf, log1pl	undefined behavior
log2	log2, log2f, log2l	undefined behavior
logb	logb, logbf, logbl	undefined behavior
lrint	lrint, lrintf, lrintl	undefined behavior
lround	lround, lroundf, lroundl	undefined behavior
nearbyint	nearbyint, nearbyintf, nearbyintl	undefined behavior
nextafter	nextafter, nextafterf, nextafterl	undefined behavior
nexttoward	nexttoward, nexttowardf, nexttowardl	undefined behavior
remainder	remainder, remainderf, remainderl	undefined behavior
remquo	remquo, remquof, remquol	undefined behavior
rint	rint, rintf, rintl	undefined behavior
round	round, roundf, roundl	undefined behavior
scalbn	scalbn, scalbnf, scalbnl	undefined behavior
scalbln	scalbln, scalblnf, scalblnl	undefined behavior
tgamma	tgamma, tgammaf, tgammal	undefined behavior
trunc	trunc, truncf, truncl	undefined behavior
carg	carg, cargf, cargl	carg, cargf, cargl

.....continued		
Type-generic macro used	**Called function variants with all real arguments**	**Called function variants with at least one complex argument**
cimag	cimag, cimagf, cimagl	cimag, cimagf, cimagl
conj	conj, conjf, conjl	conj, conjf, conjl
cproj	cproj, cprojf, cprojl	cproj, cprojf, cprojl
creal	creal, crealf, creall	creal, crealf, creall

5.22 <time.h> Date and Time Functions

The header file time.h consists of types, macros and functions that manipulate date and time.

5.22.1 time.h Types and Macros

5.22.1.1 clock_t

A type that can hold processor time values, as returned by the clock function.

Include

<time.h>

5.22.1.2 struct tm

Structure used to hold the components of a calendar time, called the broken-down time

Include

<time.h>

Definition

```
struct tm {
  int  tm_sec;           /*seconds after the minute ( 0 to 61 )*/
                         /*allows for up to two leap seconds*/
  int  tm_min;           /*minutes after the hour ( 0 to 59 )*/
  int  tm_hour;          /*hours since midnight ( 0 to 23 )*/
  int  tm_mday;          /*day of month ( 1 to 31 )*/
  int  tm_mon;           /*month ( 0 to 11 where January = 0 )*/
  int  tm_year;          /*years since 1900*/
  int  tm_wday;          /*day of week ( 0 to 6 where Sunday = 0 )*/
  int  tm_yday;          /*day of year ( 0 to 365 where January 1 = 0 )*/
  int  tm_isdst;         /*Daylight Savings Time flag */
  long __tm_gmtoff;      /*number of seconds offset from UTC to get local time*/
  const char *__tm_zone; /*name of the time zone*/
}
```

Remarks

If tm_isdst is a positive value, Daylight Savings is in effect. If it is zero, Daylight Saving Time is not in effect. If it is a negative value, the status of Daylight Saving Time is not known.

5.22.1.3 time_t

A type that can hold calendar time values, as returned by the time function.

Include

<time.h>

5.22.2 size_t Type

An unsigned integer type used by the result of the sizeof operator

Include

<stddef.h>

<stdio.h>

<stdlib.h>

<string.h>

<time.h>

<wchar.h>

Definition

```
typedef unsigned size_t;
```

5.22.3 CLOCKS_PER_SEC Macro

The conversion factor to convert the program's processor time to seconds.

Include

<time.h>

Remarks

Divide the value returned by the `clock` function by the value of this macro to determine the time in seconds.

5.22.4 NULL Macro

A constant value which represent a null pointer constant. It's value and type is implementation defined.

Include

<locale.h>

<stddef.h>

<stdio.h>

<stdlib.h>

<string.h>

<time.h>

<wchar.h>

Definition

```
#define NULL ((void*)0)
```

5.22.5 asctime Function

Converts the time structure to a character string.

Include

<time.h>

Prototype

```
char *asctime(const struct tm *tptr);
```

Argument

tptr	time/date structure

Return Value

Returns a pointer to a character string of the following format:

`DDD MMM dd hh:mm:ss YYYY`

`DDD` is day of the week

`MMM` is month of the year

`dd` is day of the month

`hh` is hour

`mm` is minute

`ss` is second

`YYYY` is year

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

volatile int i;

int main(void)
{
  struct tm when;
  time_t whattime;

  when.tm_sec = 30;
  when.tm_min = 30;
  when.tm_hour = 2;
  when.tm_mday = 1;
  when.tm_mon = 1;
  when.tm_year = 103;

  whattime = mktime(&when);
  printf("Day and time is %s\n", asctime(&when));
}
```

Example Output

```
Day and time is Sat Feb  1 02:30:30 2003
```

5.22.6 clock Function

Determines the processor time used.

Attention: This function is not implemented by MPLAB XC8 C compilers.

Include

`<time.h>`

Prototype

```
clock_t clock(void);
```

Return Value

Returns the number of clock ticks of elapsed processor time.

Remarks

Divide the value returned by this function by the value of the `CLOCKS_PER_SEC` macro to determine a time in seconds.

If the target environment cannot measure elapsed processor time, the function returns -1 cast as a `clock_t` (i.e. `(clock_t) -1`).

When using XC16, this function uses the device Timer2 and Timer3 to compute clock ticks. By default, the compiler returns the time as instruction cycles.

Example

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

volatile int i;

int main(void)
{
  clock_t start, stop;
  int ct;

  start = clock();
  for (i = 0; i < 10; i++)
    stop = clock();
  printf("start = %ld\n", start);
  printf("stop = %ld\n", stop);
}
```

Example Output

```
start = 0
stop = 317
```

5.22.7 ctime Function

Converts calendar time to a string representation of local time.

Include

`<time.h>`

Prototype

`char *ctime(const time_t *tod);`

Argument

tod	pointer to stored time

Return Value

Returns the address of a string that represents the local time of the parameter passed.

Remarks

This function is equivalent to `asctime(localtime(tod))`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  time_t whattime;
  struct tm nowtime;

  nowtime.tm_sec = 30;
  nowtime.tm_min = 30;
  nowtime.tm_hour = 2;
  nowtime.tm_mday = 1;
  nowtime.tm_mon = 1;
  nowtime.tm_year = 103;

  whattime = mktime(&nowtime);
```

```
  printf("Day and time %s\n", ctime(&whattime));
}
```

Example Output

```
Day and time Sat Feb  1 02:30:30 2003
```

5.22.8 difftime Function

Find the difference between two times.

Include

<time.h>

Prototype

```
double difftime(time_t t1, time_t t0);
```

Arguments

t1	ending time
t0	beginning time

Return Value

Returns the number of seconds between t1 and t0.

Remarks

By default, the MPLAB XC16 compiler returns the time as instruction cycles so difftime returns the number of ticks between t1 and t0.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

volatile int i;

int main(void)
{
  time_t start, stop;
  double elapsed;

  start = clock();
  for (i = 0; i < 10; i++)
    stop = clock();
  printf("start = %ld\n", start);
  printf("stop = %ld\n", stop);
  elapsed = difftime(stop, start);
  printf("Elapsed time = %.0f\n", elapsed);
}
```

Example Output

```
start = 0
stop = 317
Elapsed time = 317
```

5.22.9 gmtime Function

Converts calendar time to time structure expressed as Universal Time Coordinated (UTC) also known as Greenwich Mean Time (GMT).

Include

<time.h>

Prototype

```
struct tm *gmtime(const time_t *tod);
```

Argument

tod	pointer to stored time

Return Value

Returns the address of the time structure.

Remarks

This function breaks down the tod value into the time structure of type tm. By default, the compiler returns the time as instruction cycles. With this default, gmtime and localtime will be equivalent, except gmtime will return tm_isdst (Daylight Savings Time flag) as zero to indicate that Daylight Savings Time is not in effect.

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

int main(void)
{
  time_t timer;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */

  newtime = gmtime(&timer);
  printf("UTC time = %s\n", asctime(newtime));
}
```

Example Output

```
UTC time = Mon Oct 20 16:43:02 2003
```

5.22.10 localtime Function

Converts a value to the local time.

Include

<time.h>

Prototype

```
struct tm *localtime(const time_t *tod);
```

Argument

tod	pointer to stored time

Return Value

Returns the address of the time structure.

Remarks

By default, the MPLAB XC16 compiler returns the time as instruction cycles. With this default, localtime and gmtime will be equivalent, except localtime will return tm_isdst (Daylight Savings Time flag) as -1 to indicate that the status of Daylight Savings Time is not known.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  time_t timer;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */

  newtime = localtime(&timer);
  printf("Local time = %s\n", asctime(newtime));
}
```

Example Output

```
Local time = Mon Oct 20 16:43:02 2003
```

5.22.11 mktime Function

Converts local time to a calendar value.

Include

`<time.h>`

Prototype

`time_t mktime(struct tm *`**`tptr`**`);`

Argument

`tptr`	a pointer to the time structure

Return Value

Returns the calendar time encoded as a value of `time_t`.

Remarks

If the calendar time cannot be represented, the function returns -1 cast as a `time_t` (i.e. `(time_t) -1`).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  time_t timer, whattime;
  struct tm *newtime;

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for struct tm */
  newtime = localtime(&timer);
  printf("Local time = %s", asctime(newtime));

  whattime = mktime(newtime);
  printf("Calendar time as time_t = %ld\n",
          whattime);
}
```

Example Output

```
Local time = Mon Oct 20 16:43:02 2003
Calendar time as time_t = 1066668182
```

5.22.12 strftime Function

Formats the time structure to a string based on the format parameter.

Attention: This function is not implemented by MPLAB XC8 C compilers.

Include

```
<time.h>
```

Prototype

```
size_t strftime(char *s, size_t n, const char *format, const struct tm *tptr);
```

Arguments

s	output string
n	maximum length of string
format	format-control string
tptr	pointer to tm data structure

Return Value

Returns the number of characters placed in the array, `s`, if the total, including the terminating null, is not greater than `n`. Otherwise, the function returns 0 and the contents of array `s` are indeterminate.

Remarks

The function places no more than `maxsize` characters into the array pointed to by `s` as controlled by the format string, which consists of zero or more conversion specifiers and ordinary characters. A conversion specifier consists of a `%` character, possibly followed by an `E` or `O` modifier character, followed by a character that determines the behavior of the conversion specifier. All ordinary characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined.

Each conversion specifier is replaced by appropriate characters, which are determined using the `LC_TIME` category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by `timeptr`, as indicated in the description. In the `"C"` locale, the `E` and `O` modifiers are ignored and some specifiers are replaced by different strings, which are indicated in square brackets. The `%g`, `%G`, and `%V` specifiers give values according to the ISO 8601 week-based year. If any of the specified values are outside the normal range, the characters stored are unspecified.

Specifier	Replacement
`%a`	abbreviated weekday name, derived from `tm_wday` ["C" locale: the first three characters of `%A`]
`%A`	full weekday name, derived from `tm_wday` ["C" locale: one of `Sunday`, `Monday`, ... , `Saturday`]
`%b`	abbreviated month name, derived from `tm_mon` ["C" locale: the first three characters of `%B`]
`%B`	full month name, derived from `tm_mon` ["C" locale: one of `January`, `February`, ... , `December`. `%c` equivalent to `%a %b %e %T %Y`]
`%c`	appropriate date and time representation [C locale: equivalent to `%a %b %e %T %Y`]

.....continued

Specifier	Replacement
%C	year, derived from tm_year, divided by 100 and truncated to an integer, as a decimal number (00−99)
%d	day of the month as a decimal number (01-31), derived from tm_mday
%D	equivalent to %m/%d/%y, derived from tm_mon, tm_mday, and tm_year
%e	day of the month, derived from tm_mday, as a decimal number (1−31); a single digit is preceded by a space
%F	equivalent to %Y-%m-%d, derived from tm_year, tm_mon, and tm_mday
%g	last 2 digits of the week-based year, derived from tm_year, tm_wday, and tm_yday, as a decimal number (00−99)
%G	week-based year, derived from tm_year, tm_wday, and tm_yday, as a decimal number (e.g., 1997)
%H	equivalent to %b, derived from tm_mon
%H	hour (24-hour clock), derived from tm_hour, as a decimal number (00−23)
%I	hour (12-hour clock), derived from tm_hour, as a decimal number (01−12)
%j	day of the year, derived from tm_yday, as a decimal number (001−366)
%m	month, derived from tm_mon, as a decimal number (01−12)
%M	minute, derived from tm_min, as a decimal number (00-59)
%n	new-line character
%p	AM/PM designator, derived from tm_hour ["C" locale" one of AM or PM]
%r	12-hour clock time, derived from tm_hour, tm_min, and tm_sec ["C" locale: equivalent to %I:%M:%S %p]
%R	equivalent to %H:%M, derived from tm_hour and tm_min
%S	second, derived from tm_sec, as a decimal number (00-60)
%t	horizontal-tab character
%T	equivalent to %H:%M:%S, derived from tm_hour, tm_min, and tm_sec ["C" locale: equivalent to %T]
%u	ISO 8601 weekday, derived from tm_wday, as a decimal number (1−7), with Monday being 1.
%U	week number of the year, derived from tm_year, tm_wday, tm_yday, where Sunday is the first day of week 1 (00-53)
%V	ISO 8601 week number, derived from tm_year, tm_wday, tm_yday, as a decimal number (01−53)
%w	weekday, derived from tm_wday, as a decimal number (0−6), where Sunday is 0
%W	week number of the year (the first Monday as the first day of week 1), derived from tm_year, tm_wday, tm_yday, as a decimal number (00−53)
%x	appropriate date representation ["C" locale: equivalent to %m/%d/%y]
%X	appropriate time representation[C locale: equivalent to %T]
%y	last 2 digits of the year, derived from tm_year, as a decimal number (00−99)

.....continued	
Specifier	**Replacement**
%Y	year, derived from tm_year, as a decimal number (e.g., 1997)
%z	offset from UTC in ISO 8601 format, derived from tm_isdst, where −0430 implies 4 hours 30 minutes behind UTC, west of Greenwich; or no characters if no time zone is determinable
%Z	time zone name or abbreviation, derived from tm_isdst, or no characters if no time zone is determinable
%%	percent character, %

Some of the above specified can be modified using the E or O modifier characters to indicate an alternative format or specification. In the "C" locale, these modifiers are ignored. If the alternative format or specification does not exist for the current locale, the modifier is ignored.

Table 5-6. Modifiers used with specifiers

Modified specifier	Replacement
%Ec	locale's alternative date and time representation
%EC	name of the base year (period) in the locale's alternative representation
%Ex	locale's alternative date representation
%EX	locale's alternative time representation
%Ey	offset from %EC (year only) in the locale's alternative representation
%EY	locale's full alternative year representation
%Od	day of the month, using the locale's alternative numeric symbols (filled as needed with leading zeros, or with leading spaces if there is no alternative symbol for zero)
%Oe	day of the month, using the locale's alternative numeric symbols (filled as needed with leading spaces)
%OH	hour (24-hour clock), using the locale's alternative numeric symbol
%OI	hour (12-hour clock), using the locale's alternative numeric symbols
%Om	month, using the locale's alternative numeric symbols
%OM	minutes, using the locale's alternative numeric symbol
%OS	seconds, using the locale's alternative numeric symbols
%Ou	ISO 8601 weekday as a number in the locale's alternative representation, where Monday is 1
%OU	week number, using the locale's alternative numeric symbols
%OV	ISO 8601 week number, using the locale's alternative numeric symbols
%Ow	weekday as a number, using the locale's alternative numeric symbols
%OW	week number of the year, using the locale's alternative numeric symbols
%Oy	last 2 digits of the year, using the locale's alternative numeric symbols

Example

See the notes at the beginning of this chapter or section for information on using printf() or scanf() (and other functions reading and writing the stdin or stdout streams) in the example code.

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

```
int main(void)
{
  time_t timer, whattime;
  struct tm *newtime;
  char buf[128];
  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for structure */
  newtime = localtime(&timer);
  strftime(buf, 128, "It was a %A, %d days into the "
           "month of %B in the year %Y.\n", newtime);
  printf(buf);
  strftime(buf, 128, "It was %W weeks into the year "
           "or %j days into the year.\n", newtime);
  printf(buf);
}
```

Example Output

```
It was a Monday, 20 days into the month of October in the year 2003.
It was 42 weeks into the year or 293 days into the year.
```

5.22.13 time Function

Calculates the current calendar time.

Include

`<time.h>`

Prototype

`time_t time(time_t * tod);`

Argument

tod	pointer to storage location for time

Return Value

Returns the calendar time encoded as a value of `time_t`.

Remarks

If the target environment cannot determine the time, the function returns -1 cast as a `time_t`. By default, the compiler returns the time as instruction cycles.

When using XC16, this function uses the device Timer2 and Timer3 to compute the current time.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

volatile int i;

int main(void)
{
  time_t ticks;

  time(0); /* start time */
  for (i = 0; i < 10; i++) /* waste time */
    continue;
  time(&ticks); /* get time */
  printf("Time = %ld\n", ticks);
}
```

Example Output

```
Time = 256
```

5.23 <wchar.h> Wide character utilities

The header file `wchar.h` consists of types, macros, and functions that are useful for conversion and input and output of wide characters. Characters are interpreted according to the Standard C locale.

Attention: This header is implemented only by MPLAB XC32 C compilers.

5.23.1 mbstate_t Type

An object type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters.

Attention: This type is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

5.23.2 size_t Type

An unsigned integer type used by the result of the `sizeof` operator

Include

`<stddef.h>`

`<stdio.h>`

`<stdlib.h>`

`<string.h>`

`<time.h>`

`<wchar.h>`

Definition

```
typedef unsigned size_t;
```

5.23.3 wint_t Type

An integer type used to hold any value corresponding to members of the extended character set, as well as the wide end-of-file marker, WEOF.

Attention: This type is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

<wchar.h>

Definition

```
typedef unsigned wint_t;
```

5.23.4 NULL Macro

A constant value which represent a null pointer constant. It's value and type is implementation defined.

Include

<locale.h>

<stddef.h>

<stdio.h>

<stdlib.h>

<string.h>

<time.h>

<wchar.h>

Definition

```
#define NULL ((void*)0)
```

5.23.5 WEOF Macro

A constant expression of type `wint_t` whose value does not correspond to any member of the extended character set and which indicates end-of-file, that is, no more input from a stream. It is also used as a wide character value that does not correspond to any member of the extended character set.

Attention: This type is implemented only by MPLAB XC32 C compilers.

Include

<wctype.h>

<wchar.h>

5.23.6 btowc Function

Converts a character to an equivalent wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

<wchar.h>

Prototype

```
wint_t btowc(int c);
```

Arguments

c	the character to convert

Return Value

The function returns the character `c` as wide character (type cast to type `wint_t`) provided that `c` is a valid single-byte character in the initial shift state of a multibyte sequence. Otherwise, or if `c` is `EOF`, the function returns `WEOF`.

Remarks

The function determines whether `c` constitutes a valid single-byte character in the initial shift state.

A multibyte character set may have a state-dependent encoding. Sequences begins in an initial shift state and enter other locale-specific shift states when specific multibyte characters are encountered. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  int i, num = 0;
  const char mbs[] = "The quick brown fox\n";

  for(i=0; i!=sizeof(mbs); i++)
    if(btowc(mbs[i]) != WEOF)
      num++;

  wprintf(L"mbs contains %d single-byte characters.\n", num);
}
```

Example Output

```
mbs contains 21 single-byte characters.
```

5.23.7 fgetwc Function

Obtains wide character input from a stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int fgetwc(FILE * stream);
```

Arguments

`stream`	the stream to read input from

Return Value

Returns the next available wide character as a `wchar_t` converted to a `wint_t`. The `WEOF` wide character is returned if the end-of-file indicator for the stream is set, if the stream is at end-of-file, if a read error occurs, or if an encoding error occurs.

Remarks

This function advances the associated file position indicator for the stream (if defined) and sets the end-of file indicator when the end-of-file is reached. If a read error occurs, the error indicator for the stream is set and can be

checked with the `ferror()` function. If an encoding error occurs (which includes too few bytes), `errno` is set to the value of the `EILSEQ` macro.

Example

```
#include <wchar.h>

int main(void)
{
  FILE * myfile;
  wint_t wc;

  if ((myfile = fopen("afile", "r")) == NULL)
    wprintf(L"Cannot open afile\n");
  else
  {
    while( ! feof(myfile)) {
      wc = fgetwc(myfile);
      if(iswprint(wc))
        wprintf(L"char read: %lc\n", wc);
    }
    fclose(myfile);
  }
}
```

Example Input

Content of `afile`.

```
Four score
```

Example Output

```
char read: F
char read: o
char read: u
char read: r
char read:
char read: s
char read: c
char read: o
char read: r
char read: e
```

5.23.8 fgetws Function

Obtains wide string input from a stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t * fgetws(wchar_t * restrict s, int n, FILE * stream);
```

Arguments

s	the wide array in which to store the string
n	the maximum number of bytes written to the array
stream	the stream to read input from

Return Value

After a successful read, the function returns `s`. If end-of-file is encountered and no characters have been read into the array, the the array is not modified and a null pointer is returned. If a read or encoding error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

Remarks

This function reads no more than `n`-1 wide characters from the stream, terminating once encountering a new-line wide character or and end-of-file. A null wide character is written to the array after those wide characters read from the stream.

Example

```
#include <wchar.h>

int main(void)
{
  FILE * myfile;
  wint_t ws[30];

  if ((myfile = fopen("afile", "r")) == NULL)
    wprintf(L"Cannot open afile\n");
  else
  {
    if(fgetws(ws, 30, myfile) == NULL)
      wprintf(L"String read returned error\n");
    else
      wprintf(L"Your string: %ls\n", ws);
    fclose(myfile);
  }
}
```

Example Input

Content of `afile`.

```
Four and 20
```

Example Output

```
Your string: Four and 20
```

5.23.9 fputwc Function

Writes a wide character to a stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

`wint_t fputwc(wchar_t` **`c`**`, FILE *` **`stream`**`);`

Arguments

`c`	the wide character to write
`stream`	the stream to write to

Return Value

The function returns a copy of the wide character written. If a write or encoding error occurs, `WEOF` is returned.

Remarks

The function writes the wide character, `c`, to the output stream pointed to by stream and at the position indicated by the file position indicator for the stream (if defined), advancing the indicator appropriately. If positioning within the file is not supported or if the stream was opened with append mode, the character is appended to the output stream.

The error indicator for the stream will be set on a write error. If an encoding error occurs, `errno` is set to the value of the `EILSEQ` macro.

Example

```
#include <wchar.h>

int main(void)
{
  FILE * myfile;
  wint_t wc;

  if ((myfile = fopen("afile", "w")) == NULL)
    wprintf(L"Cannot open afile\n");
  else
  {
    for(wc = L'0'; iswdigit(wc); wc++)
      fputwc(wc, myfile);
    fputwc(L'\n', myfile);
    fclose(myfile);
  }
}
```

Example Output

Content of `afile`.

```
0123456789
```

5.23.10 fputws Function

Writes a wide string to a stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int fputws(const wchar_t * restrict s, FILE * stream);
```

Arguments

s	the wide string to write
stream	the stream to write to

Return Value

The function returns a non-negative number. If a write or encoding error occurs, it return `EOF`.

Remarks

The function writes the wide string, s, to the output stream pointed to by stream. The null wide character terminating the wide string is not written.

Example

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

```
int main(void)
{
  FILE * myfile;
  wchar_t ws[] = L"One string 4 all";

  if ((myfile = fopen("afile", "w")) == NULL)
    wprintf(L"Cannot open afile\n");
  else
  {
    fputws(ws, myfile);
    fclose(myfile);
  }
}
```

Example Output

Content of `afile`.

```
One string 4 all
```

5.23.11 fwide Function

Sets and determines the width orientation of a stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

`int fwide(FILE * stream, int mode);`

Arguments

stream	the stream to write to
mode	the stream orientation to set

Return Value

The function returns a positive value if, after the call, the stream has wide orientation, a negative value if the stream has byte orientation, or zero if the stream has no orientation.

Remarks

For positive values of `mode`, the function attempts to make the stream wide oriented; for negative values, it attempts to make the stream byte oriented. For a `mode` of zero, the orientation of the stream is not altered. Once a stream has been oriented, subsequent calls to `fwide()` do not change that stream's orientation.

Example

```
#include <wchar.h>

void prStrOrient(int orientation)
{
  if (orientation > 0)
    puts("The stream is wide oriented");
  else if (orientation < 0)
    puts("The stream is byte oriented");
  else
    puts("The stream is not oriented");
}

int main(void)
{
```

```
  FILE * myfile;
  int ret;

  if ((myfile = fopen("afile", "w")) == NULL)
    wprintf(L"Cannot open afile\n");
  else {
    ret = fwide(myfile, 0);
    prStrOrient(ret);
    ret = fwide(myfile, 1);     // set wide orientation
    prStrOrient(ret);
    ret = fwide(myfile, -1);    // no further changes made
    prStrOrient(ret);
    fclose (myfile);
  }
}
```

Example Output

Content of `afile`.

```
The stream is not oriented
The stream is wide oriented
The stream is wide oriented
```

5.23.12 fwprintf Function

Prints formatted text to a stream, using a wide format string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);
```

Arguments

`stream`	pointer to the stream in which to output data
`format`	format control wide string
`...`	optional arguments; see “Remarks”

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

The function writes formatted output to the specified stream.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` wide character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[`*`flags`*`][`*`width`*`][.`*`precision`*`][`*`length`*`]`*`specifier`*

The *`flags`* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first wide character of a conversion result does not include a sign (+ or -) or if a signed conversion results in no wide characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag - has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.

.....continued

Modifier	Meaning
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point wide character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a wide character (as if by calling btowc) and the resulting wide character is written. When the l modifier is present, the wint_t argument is converted to wchar_t and written.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point wide character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.

.....continued

Specifier	Argument value type	Printing notes
f, F	double	Converted to decimal notation using the general form [−]*ddd*.*ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of wide characters written so far is written to the object pointed to by the pointer. No wide characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (address) held by the pointer.
s	char array, or wchar_t array	A string contain multibyte characters that are converted to wide characters (as if by calling mbrtowc) and written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written. When the l modifier is present, wide characters from the array are written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % wide character is printed.

Example

```
#include <wchar.h>

int main(void)
{
  FILE *myfile;
  int y;
  wchar_t s[]=L"Print this string";
  int x = 1;
  wchar_t a = L'\n';

  if ((myfile = fopen("afile", "w")) == NULL)
    printf("Cannot open afile\n");
  else {
    y = fwprintf(myfile, L"%ls %d time%lc", s, x, a);

    printf("Number of characters printed to file = %d\n", y);

    fclose(myfile);
```

```
    }
}
```

Example Output

```
Number of characters printed to file = 25
```

Contents of `afile`:

```
Print this string 1 time
```

5.23.13 fwscanf Function

Scans formatted text from a stream, using a wide format string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);
```

Arguments

`stream`	pointer to the open stream from which to read data
`format`	format control wide string
`...`	optional arguments; see “Remarks”

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. `EOF` is returned if end-of-file is encountered before the first conversion or if an error occurs.

Remarks

This function can read input from the specified stream.

The `format` string can be composed of white space wide characters, which reads input up to the first non-white-space wide character in the input; or ordinary wide characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` wide character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent wide character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent wide character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `short int` or `unsigned short int` object referenced by the pointer argument.

.....continued

Modifier	Meaning
hh	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a signed char or unsigned char object referenced by the pointer argument.
l	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long int or unsigned long int object referenced by the pointer argument.
l	When used the a, A, e, E, f, F, g, or G indicates that this conversion specifier assigns to a double object referenced by the pointer argument.
l	When used the c, s, or [indicates that this conversion specifier assigns to a wchar_t object referenced by the pointer argument.
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single wide character or the number of characters specified by the field width if present. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array of sufficient size. No null character is added. If an l length modifier is present, the input shall be a wide character array of sufficient size. No null wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.

.....continued

Specifier	Receiving object type	Matches
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	int	No input is consumed but the number of wide characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fwprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space wide characters, which is written to the array argument. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large of sufficient size. A terminating null character is appended. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide character array of sufficient size. A terminating null wide character is appended.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
%		Matches a single % wide character. No assignment takes place.

.....continued

Specifier	Receiving object type	Matches
[	`char` array, or `wchar_t` array	Matches all the wide characters in the input that have been specified between the `[` and trailing `]` wide character, unless the wide character after the opening bracket is a circumflex, `^`, in which case a match is only made with wide characters that do not appear in the brackets. If the conversion specifier begins with `[]` or `[^]`, the right bracket wide character will match the input and the next following right bracket wide character is the matching right bracket that ends the specification. If no `l` length modifier is present, characters from the input field are converted to amultibyte character sequence (as if by calling `wcrtomb`). The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. If an `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide array of sufficient size.A terminating null wide character will be appended.

Example

```
#include <wchar.h>

int main(void)
{
  FILE *myfile;
  wchar_t s[30];
  int x;
  wchar_t a;

  if ((myfile = fopen("afile", "w+")) == NULL)
    wprintf(L"Cannot open afile\n");
  else
  {
    fwprintf(myfile, L"%ls %d times%lc", L"Print this string", 100, L'.');

    fseek(myfile, 0L, SEEK_SET);

    fwscanf(myfile, L"%ls", s);
    wprintf(L"%ls\n", s);
    fwscanf(myfile, L"%ls", s);
    wprintf(L"%ls\n", s);
    fwscanf(myfile, L"%ls", s);
    wprintf(L"%ls\n", s);
    fwscanf(myfile, L"%d", &x);
    wprintf(L"%d\n", x);
    fwscanf(myfile, L"%ls", s);
    wprintf(L"%ls\n", s);

    fclose(myfile);
  }
}
```

Example Input

Contents of `afile`:

```
Print this string 100 times.
```

Example Output

```
Print
this
string
100
times.
```

5.23.14 getwc Function

Obtains wide character input from a stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int getwc(FILE * stream);
```

Arguments

stream	the stream to read input from

Return Value

Returns the next available wide character as a `wchar_t` converted to a `wint_t`. The `WEOF` wide character is returned if the end-of-file indicator for the stream is set, if the stream is at end-of-file, if a read error occurs, or if an encoding error occurs.

Remarks

The `getwc()` function is equivalent to `fgetwc()`. It is implemented as a XXXX in MPLAB XC16 and XC32 (, hence it may evaluate stream more than once.)

Example

```
#include <wchar.h>

int main(void)
{
  FILE * myfile;
  wint_t wc;

  if ((myfile = fopen("afile", "r")) == NULL)
    wprintf(L"Cannot open afile\n");
  else
  {
    while( ! feof(myfile)) {
      wc = getwc(myfile);
      if(iswprint(wc))
        wprintf(L"char read: %lc\n", wc);
    }
    fclose(myfile);
  }
}
```

Example Input

Content of `afile`.

```
Four score
```

Example Output

```
char read: F
char read: o
char read: u
char read: r
char read:
char read: s
char read: c
char read: o
```

```
char read: r
char read: e
```

5.23.15 getwchar Function

Obtains wide character input from the `stdin` stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wint_t getwchar(void);
```

Return Value

Returns the next available wide character as a `wchar_t` converted to a `wint_t`. The `WEOF` wide character is returned if the end-of-file indicator for the stream is set, if the stream is at end-of-file, if a read error occurs, or if an encoding error occurs.

Remarks

This function advances the associated file position indicator for `stdin` (if defined) and sets the end-of file indicator when the end-of-file is reached. If a read error occurs, the error indicator for `stdin` is set and can be checked with the `ferror()` function. If an encoding error occurs (which includes too few bytes), `errno` is set to the value of the `EILSEQ` macro.

Example

```
#include <wchar.h>

int main(void)
{
  wint_t wc;

  while( ! feof(stdin)) {
    wc = getwchar();
    if(iswprint(wc))
      wprintf(L"char read: %lc\n", wc);
  }
}
```

Example Input

Present on `stdin`.

```
hi there
```

Example Output

```
char read: h
char read: i
char read:
char read: t
char read: h
char read: e
char read: r
char read: e
```

5.23.16 mbrlen Function

Determines the length of a multibyte character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Arguments

s	a pointer to the multibyte character to check
n	the maximum number of bytes to check
ps	a pointer to a `mbstate_t` object that holds the current conversion state

Return Value

The function returns the length of a multibyte sequence between 0 and `n` inclusive, representing the length; or the value -1 or -2, as per the `mbrtowc()` function. nonzero if `ps` is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, it returns zero.

Remarks

An object of type `mbstate_t` is used to describe the current conversion state from a particular multibyte character sequence to a wide character sequence (or the reverse conversion) based on the current locale. For both conversions, the initial conversion state corresponds to the beginning of a new multibyte character in the initial shift state. A zero-valued `mbstate_t` always represents an initial conversion state.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char * string = "The final frontier";
  mbstate_t state = {0};
  wchar_t   wc;

  mbrtowc(&wc, string, MB_CUR_MAX, &state);
  if(mbsinit(&state))                 // check initial conversion state
    wprintf(L"In initial conversion state\n");
  else
    memset(&state, 0, sizeof(state));   // set initial conversion state
}
```

Example Output

```
In initial conversion state
```

5.23.17 mbrtowc Function

Convert a multibyte character sequence to a wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t *
restrict ps);
```

Arguments

pwc	a pointer to the object to hold the converted wide character
s	the mulitbyte character sequence to convert
n	the maximum number of bytes to convert
ps	a pointer to a `mbstate_t` object that holds the current conversion state

Return Value

The function returns the first of the following that applies (given the current conversion state):

0	if the complete and valid corresponding wide character is the null wide character
a positive value	for all other complete and valid corresponding wide characters, where the returned value is the number of bytes that complete the multibyte character
(size_t)(-2)	if an incomplete (but potentially valid) multibyte character has been determined after processing all `n` bytes
(size_t)(-1)	if an encoding error occurs while processing the multibyte characters

Remarks

If `s` is a null pointer, the function behaves as if it had been called, `mbrtowc(NULL, "", 1, ps).`

For non-null values of `s`, the function inspects at most n bytes of `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If this multibyte character is complete and valid, the value of the corresponding wide character is stored in the object pointed to by `pwc`, provided that is not a null pointer. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

If the processed multibyte character is complete and valid, it is stored to the object pointed to by `pwc`; otherwise, no value is stored.

The function uses and updates the shift state described by `ps`. If `ps` is a null pointer, the function uses its own internal object to hold the shift state.

If an encoding error occurs, the value of the macro `EILSEQ` is stored in `errno`, and the conversion state is unspecified.

An object of type `mbstate_t` is used to describe the current conversion state from a particular multibyte character sequence to a wide character sequence (or the reverse conversion) based on the current locale. For both conversions, the initial conversion state corresponds to the beginning of a new multibyte character in the initial shift state. A zero-valued `mbstate_t` always represents an initial conversion state.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

void printMulti(const char * pt, size_t max)
{
  size_t length;
  wchar_t ws;
  mbstate_t mbs;
```

```
  char needNL = 0;

  mbrlen(NULL, 0, &mbs);  /* initialize conversion state */

  while(max > 0) {
    length = mbrtowc(&ws, pt, max, &mbs);
    if((length==0) || (length>max))
      break;
    else
      needNL = 1;
    wprintf(L"[%lc]", ws);
    pt += length;
    max -= length;
  }
  if(needNL)
    wprintf(L"\n");
}

int main(void)
{
  const char str[] = "a string";

  printMulti(str, sizeof(str));

  return 0;
}
```

Example Output

```
[a][ ][s][t][r][i][n][g]
```

5.23.18 mbsinit Function

Determines whether an object describes an initial conversion state.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int mbsinit(const mbstate_t * ps);
```

Arguments

ps	a pointer to the object to check

Return Value

The function returns nonzero if `ps` is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, it returns zero.

Remarks

An object of type `mbstate_t` is used to describe the current conversion state from a particular multibyte character sequence to a wide character sequence (or the reverse conversion) based on the current locale. For both conversions, the initial conversion state corresponds to the beginning of a new multibyte character in the initial shift state. A zero-valued `mbstate_t` always represents an initial conversion state.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
  char * string = "The final frontier";
  mbstate_t state = {0};
  wchar_t   wc;

  mbrtowc(&wc, string, MB_CUR_MAX, &state);
  if(mbsinit(&state))                 // check initial conversion state
    wprintf(L"In initial conversion state\n");
  else
    memset(&state, 0, sizeof(state));   // set initial conversion state
}
```

Example Output

```
In initial conversion state
```

5.23.19 mbsrtowcs Function

Convert a multibyte character sequence to a wide character string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len,
mbstate_t * restrict ps);
```

Arguments

dst	a pointer to the object to hold the converted wide character
src	the mulitbyte character sequence to convert
len	the maximum number of bytes to convert
ps	a pointer to a `mbstate_t` object that holds the current conversion state

Return Value

The function returns the number of multibyte characters successfully converted, not including the terminating null character (if any). If `src` is not a valid multibyte character sequence, `(size_t)(-1)` is returned.

Remarks

The function converts a sequence of multibyte characters indirectly pointed to by `src` and that begins in the conversion state described by the object pointed to by `ps` into a sequence of corresponding wide characters, storing the converted characters (including a terminating null character) into the array pointed to by `dst`, provided `dst` is not a null pointer, Conversion stops when a terminating null wide character is encountered, when an invalid multibyte character sequence is encountered, or when `len` wide characters have been stored. Each conversion takes place as if by a call to the `mbrtowc()` function; however, by having the argument to specify the current conversion state, this function is restartable.

An encoding error occurs if `src` is not a valid multibyte character sequence, and the function stores the value of the macro `EILSEQ` in `errno` and the conversion state is unspecified.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <locale.h>
#include <wchar.h>
#include <string.h>

void printWide(const char* mbstr)
{
    mbstate_t state;
    size_t len;
    wchar_t wstr[40];

    memset(&state, 0, sizeof state);
    len = mbsrtowcs(&wstr[0], &mbstr, 40, &state);
    wprintf(L"Converted wide string \"%ls\" has length %u\n", wstr, len);
}

int main(void)
{
    setlocale(LC_CTYPE, "");
    printWide("It is 75°F");
}
```

Example Output

```
Converted wide string "It is 75°F" has length 10
```

5.23.20 putwc Function

Writes a wide character to a stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wint_t putwc(wchar_t c, FILE * stream);
```

Arguments

c	the wide character to write
stream	the stream to write to

Return Value

The function returns a copy of the wide character written. If a write or encoding error occurs, `WEOF` is returned.

Remarks

The `putwc()` function is equivalent to `fputwc()`. It is implemented as a XXXX in MPLAB XC16 and XC32 (hence it may evaluate stream more than once).

Example

```
#include <wchar.h>

int main(void)
```

```
{
  FILE * myfile;
  wint_t wc;

  if ((myfile = fopen("afile", "w")) == NULL)
    wprintf(L"Cannot open afile\n");
  else
  {
    for(wc = L'0'; iswdigit(wc); wc++)
      putwc(wc, myfile);
    fclose(myfile);
  }
}
```

Example Output

Content of `afile`.

```
0123456789
```

5.23.21 putwchar Function

Writes a wide character to `stdout`.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wint_t putwchar(wchar_t c);
```

Arguments

c	the wide character to write

Return Value

The function returns a copy of the wide character written. If a write or encoding error occurs, `WEOF` is returned.

Remarks

The function writes the wide character, `c`, to `stdout` and at the position indicated by the file position indicator for the stream (if defined), advancing the indicator appropriately. The error indicator for `stdout` will be set on a write error. If an encoding error occurs, `errno` is set to the value of the `EILSEQ` macro.

Example

```
#include <wchar.h>

int main(void)
{
  wint_t wc;

  for(wc = L'0'; iswdigit(wc); wc++)
    putwchar(wc);
  putwchar(L'\n');
}
```

Example Output

Content of `afile`.

```
0123456789
```

5.23.22 ungetwc Function

Pushes a wide character back to an input stream.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wint_t ungetwc(wint_t c, FILE *stream);
```

Arguments

c	the wide character to push back
stream	the stream to accept the wide character

Return Value

The function returns the wide character pushed back, or `WEOF` if the operation fails.

Remarks

The `getwc()` function is equivalent to `fgetwc()`. It is implemented as a XXXX in MPLAB XC16 and XC32 (hence it may evaluate stream more than once).

Example

```
#include <wchar.h>

int main(void)
{
  FILE * myfile;
  wint_t wc;
  wchar_t buffer[256];

  myfile = fopen("afile", "rt");
  if(myfile!=NULL)
    while(!feof (myfile)) {
      wc=getwc(myfile);
      if (wc != WEOF) {
        if (iswlower(wc)) // is the first wide char lower case
          ungetwc(towupper(wc), myfile);  // make it upper case
      else
          ungetwc(wc, myfile);
      fgetws(buffer, 255, myfile);
      fputws(buffer, stdout);
    }
  }
}
```

Example Input

Content of `afile`.

```
The curious CASE of this File
the button on the Shirt
The function of this task
```

Example Output

```
The curious CASE of this File
The button on the Shirt
The function of this task
```

5.23.23 swprintf Function

Prints formatted text to a stream, using a wide format string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, ...);
```

Arguments

s	wide string for output
n	the maximum number of characters to print
format	format control wide string
...	optional arguments; see “Remarks”

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

The function writes formatted output to the specified string.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` wide character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[flags][width][.precision][length]specifier`

The *flags* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first wide character of a conversion result does not include a sign (+ or -) or if a signed conversion results in no wide characters. If the *space* and + flags both appear, the *space* flag is ignored.

.....continued

Flag	Meaning
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag – has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.

.....continued	
Modifier	**Meaning**
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h*.*hhhhp*±*d*, where there is one hexadecimal digit before the decimal-point wide character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a wide character (as if by calling btowc) and the resulting wide character is written. When the l modifier is present, the wint_t argument is converted to wchar_t and written.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d*.*ddd*e±*dd*, where there is one digit before the decimal-point wide character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
f, F	double	Converted to decimal notation using the general form [−]*ddd*.*ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.

.....continued		
Specifier	**Argument value type**	**Printing notes**
n	an integer pointer	The number of wide characters written so far is written to the object pointed to by the pointer. No wide characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (address) held by the pointer.
s	char array, or wchar_t array	A string contain multibyte characters that are converted to wide characters (as if by calling mbrtowc) and written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written. When the l modifier is present, wide characters from the array are written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % wide character is printed.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[30];
  int y;
  wchar_t s[] = L"Print this string";
  int x = 1;
  wchar_t a = L'\n';

  y = swprintf(ws, 30, L"%ls %d time%lc", s, x, a);
  printf("Number of characters printed to wide string buffer = %d\n", y);
}
```

Example Output

```
Number of characters printed to file = 25
```

Contents of afile:

```
Print this string 1 time
```

5.23.24 swscanf Function

Scans formatted text from a wide string, using a wide format string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
```

Arguments

s	wide string for input
format	format control wide string
...	optional arguments; see “Remarks”

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. `EOF` is returned if end-of-file is encountered before the first conversion or if an error occurs.

Remarks

This function can read from the specified wide string.

The `format` string can be composed of white space wide characters, which reads input up to the first non-white-space wide character in the input; or ordinary wide characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` wide character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent wide character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent wide character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a short int or unsigned short int object referenced by the pointer argument.
hh	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a signed char or unsigned char object referenced by the pointer argument.
l	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long int or unsigned long int object referenced by the pointer argument.
l	When used the a, A, e, E, f, F, g, or G indicates that this conversion specifier assigns to a double object referenced by the pointer argument.
l	When used the c, s, or [indicates that this conversion specifier assigns to a wchar_t object referenced by the pointer argument.

.....continued

Modifier	Meaning
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single wide character or the number of characters specified by the field width if present. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array of sufficient size. No null character is added. If an l length modifier is present, the input shall be a wide character array of sufficient size. No null wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
n	int	No input is consumed but the number of wide characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

.....continued

Specifier	Receiving object type	Matches
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fwprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space wide characters, which is written to the array argument. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large of sufficient size. A terminating null character is appended. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide character array of sufficient size. A terminating null wide character is appended.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
%		Matches a single % wide character. No assignment takes place.
[	char array, or wchar_t array	Matches all the wide characters in the input that have been specified between the [and trailing] wide character, unless the wide character after the opening bracket is a circumflex, ^, in which case a match is only made with wide characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket wide character will match the input and the next following right bracket wide character is the matching right bracket that ends the specification. If no l length modifier is present, characters from the input field are converted to amultibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide array of sufficient size.A terminating null wide character will be appended.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[] = L"5 T green 3000000.00";
  int number, items;
  wchar_t letter;
  wchar_t color[10];
  float salary;
```

```
  items = swscanf(ws, L"%d %lc %ls %f", &number, &letter,
              &color, &salary);

  printf("Number of items scanned = %d\n", items);
  printf("Favorite number = %d\n", number);
  printf("Favorite letter = %lc\n", letter);
  printf("Favorite color = %ls\n", color);
  printf("Desired salary = $%.2f\n", salary);

}
```

Example Output

```
Number of items scanned = 4
Favorite number = 5
Favorite letter = T
Favorite color = green
Desired salary = $3000000.00
```

5.23.25 vfwprintf Function

Prints formatted data to a stream, using a wide format string and variable length argument list.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int vfwprintf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
```

Arguments

stream	pointer to the stream in which to output data
format	format control wide string
arg	variable argument list to print

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

The function writes formatted output to the specified stream.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` wide character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[flags][width][.precision][length]specifier`

The *flags* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.

.....continued	
Flag	**Meaning**
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and − flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first wide character of a conversion result does not include a sign (+ or −) or if a signed conversion results in no wide characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag − has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.

.....continued

Modifier	Meaning
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point wide character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a wide character (as if by calling btowc) and the resulting wide character is written. When the l modifier is present, the wint_t argument is converted to wchar_t and written.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point wide character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.

.....continued

Specifier	Argument value type	Printing notes
f, F	double	Converted to decimal notation using the general form [−]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of wide characters written so far is written to the object pointed to by the pointer. No wide characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (address) held by the pointer.
s	char array, or wchar_t array	A string contain multibyte characters that are converted to wide characters (as if by calling mbrtowc) and written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written. When the l modifier is present, wide characters from the array are written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % wide character is printed.

Example

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

FILE * myfile;

void errmsg(const wchar_t * fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  vfwprintf(myfile, fmt, ap);
  va_end(ap);
}

int main(void)
{
  int num = 3;
```

```
  if ((myfile = fopen("afile.txt", "w")) == NULL)
    printf("Cannot open afile.txt\n");
  else
  {
    errmsg(L"Error: The letter '%c' is not %s\n", 'a', "an integer value.");
    errmsg(L"Error: Requires %d%ls%lc", num, L" or more characters.", L'\n');
  }
  fclose(myfile);
}
```

Example Output

Contents of `afile.txt`:

```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

5.23.26 vfwscanf Function

Scans formatted text from a stream, using a wide format string and a variable length argument list.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int vfwscanf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
```

Arguments

`stream`	pointer to the open stream from which to read data
`format`	format control wide string
`arg`	variable argument list of objects to be assigned values read in

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. `EOF` is returned if end-of-file is encountered before the first conversion or if an error occurs.

Remarks

This function can read input from the specified stream. The `arg` argument must have been initialized by the `va_start` macro.

The `format` string can be composed of white space wide characters, which reads input up to the first non-white-space wide character in the input; or ordinary wide characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` wide character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent wide character followed by optional fields and a required type as shown here:

`%[*][width][length]specifier`

The presence of `*` immediately after the percent wide character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *length* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a short int or unsigned short int object referenced by the pointer argument.
hh	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a signed char or unsigned char object referenced by the pointer argument.
l	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long int or unsigned long int object referenced by the pointer argument.
l	When used the a, A, e, E, f, F, g, or G indicates that this conversion specifier assigns to a double object referenced by the pointer argument.
l	When used the c, s, or [indicates that this conversion specifier assigns to a wchar_t object referenced by the pointer argument.
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single wide character or the number of characters specified by the field width if present. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array of sufficient size. No null character is added. If an l length modifier is present, the input shall be a wide character array of sufficient size. No null wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.

.....continued

Specifier	Receiving object type	Matches
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	int	No input is consumed but the number of wide characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fwprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space wide characters, which is written to the array argument. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large of sufficient size. A terminating null character is appended. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide character array of sufficient size. A terminating null wide character is appended.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
%		Matches a single % wide character. No assignment takes place.

.....continued

Specifier	Receiving object type	Matches
[	char array, or wchar_t array	Matches all the wide characters in the input that have been specified between the [and trailing] wide character, unless the wide character after the opening bracket is a circumflex, ^, in which case a match is only made with wide characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket wide character will match the input and the next following right bracket wide character is the matching right bracket that ends the specification. If no l length modifier is present, characters from the input field are converted to amultibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide array of sufficient size.A terminating null wide character will be appended.

Example

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

int wideRead(FILE * stream, const wchar_t * format, ...)
{
  va_list args;
  int items;

  va_start (args, format);
  items = vfwscanf (stream, format, args);
  va_end (args);

  return items;
}

int main ()
{
  FILE * myfile;
  int val, items;
  wchar_t str[100];

  myfile = fopen ("afile.txt", "r");

  if (myfile!=NULL) {
    items = wideRead( myfile, L"%ls%d", str, &val);
    printf("%d items read in\n", items);
    fclose(myfile);
  }
}
```

Example Input

Contents of afile.txt:

```
In 1901, it began
```

Example Output

```
2 items read in
```

5.23.27 vswprintf Function

Prints formatted data to a wide character array using a wide format string and variable length argument list.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);
```

Arguments

s	pointer to the stream in which to output data
n	maximum number of wide characters to print
format	format control wide string
arg	variable argument list to print

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

The function writes formatted output to the specified wide character array.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` wide character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[`*`flags`*`][`*`width`*`][.`*`precision`*`][`*`length`*`]`*`specifier`*

The *`flags`* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first wide character of a conversion result does not include a sign (+ or -) or if a signed conversion results in no wide characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag - has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.

.....continued

Modifier	Meaning
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point wide character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a wide character (as if by calling btowc) and the resulting wide character is written. When the l modifier is present, the wint_t argument is converted to wchar_t and written.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point wide character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
f, F	double	Converted to decimal notation using the general form [−]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of wide characters written so far is written to the object pointed to by the pointer. No wide characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (address) held by the pointer.

.....continued		
Specifier	**Argument value type**	**Printing notes**
s	char array, or wchar_t array	A string contain multibyte characters that are converted to wide characters (as if by calling mbrtowc) and written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written. When the l modifier is present, wide characters from the array are written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % wide character is printed.

Example

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

int prWstr(wchar_t * ws, const wchar_t * fmt, ...)
{
  va_list ap;
  int n;

  va_start(ap, fmt);
  n = vswprintf(ws, 30, fmt, ap);
  va_end(ap);

  return n;
}

wchar_t wbuf[30];

int main(void)
{
  int y;
  wchar_t s[] = L"Print this string";
  int x = 1;
  wchar_t a = L'\n';

  y = prWstr(wbuf, L"%ls %d time%lc", s, x, a);
  printf("Number of characters printed to wide string buffer = %d\n", y);
}
```

Example Output

```
Number of characters printed to wide string buffer = 25
```

5.23.28 vswscanf Function

Scans formatted text from a wide string, using a wide format string and a variable length argument list.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int vswscanf(wchar_t * restrict s, const wchar_t * restrict format, va_list arg);
```

Arguments

s	pointer to the wide character array from which to read data
format	format control wide string
arg	variable argument list of objects to be assigned values read in

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. `EOF` is returned if end-of-file is encountered before the first conversion or if an error occurs.

Remarks

This function can read input from the specified wide string.

The `format` string can be composed of white space wide characters, which reads input up to the first non-white-space wide character in the input; or ordinary wide characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` wide character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent wide character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent wide character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `short int` or `unsigned short int` object referenced by the pointer argument.
`hh`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `signed char` or `unsigned char` object referenced by the pointer argument.
`l`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `long int` or `unsigned long int` object referenced by the pointer argument.
`l`	When used the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` indicates that this conversion specifier assigns to a `double` object referenced by the pointer argument.
`l`	When used the `c`, `s`, or `[` indicates that this conversion specifier assigns to a `wchar_t` object referenced by the pointer argument.

.....continued

Modifier	Meaning
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single wide character or the number of characters specified by the field width if present. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array of sufficient size. No null character is added. If an l length modifier is present, the input shall be a wide character array of sufficient size. No null wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
n	int	No input is consumed but the number of wide characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

.....continued

Specifier	Receiving object type	Matches
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fwprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space wide characters, which is written to the array argument. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large of sufficient size. A terminating null character is appended. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide character array of sufficient size. A terminating null wide character is appended.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
%		Matches a single % wide character. No assignment takes place.
[	char array, or wchar_t array	Matches all the wide characters in the input that have been specified between the [and trailing] wide character, unless the wide character after the opening bracket is a circumflex, ^, in which case a match is only made with wide characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket wide character will match the input and the next following right bracket wide character is the matching right bracket that ends the specification. If no l length modifier is present, characters from the input field are converted to amultibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide array of sufficient size.A terminating null wide character will be appended.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[] = L"9 X blue 5000000.00";
  int number, items;
  wchar_t letter;
  wchar_t color[10];
  float salary;
```

```
  items = swscanf(ws, L"%d %lc %ls %f", &number, &letter,
              &color, &salary);

  printf("Number of items scanned = %d\n", items);
  printf("Favorite number = %d\n", number);
  printf("Favorite letter = %lc\n", letter);
  printf("Favorite color = %ls\n", color);
  printf("Desired salary = $%.2f\n", salary);
}
```

Example Output

```
Number of items scanned = 4
Favorite number = 9
Favorite letter = X
Favorite color = blue
Desired salary = $5000000.00
```

5.23.29 vwprintf Function

Prints formatted data to the `stdout` stream using a wide format string and variable length argument list.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int vwprintf(const wchar_t * restrict format, va_list arg);
```

Arguments

format	format control wide string
arg	variable argument list to print

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

The function writes formatted output to the `stdout` stream. The `arg` argument must be initialized by the `va_start` macro.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` wide character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[flags][width][.precision][length]specifier`

The *flags* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
-	Left justify the conversion result within a given field width.

.....continued	
Flag	**Meaning**
0	Use 0 for the pad character instead of space (which is the default) for d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, except when converting an infinity or NaN. If the 0 and − flags both appear, the 0 flag is ignored. For d, i, o, u, x, andX conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.
+	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first wide character of a conversion result does not include a sign (+ or −) or if a signed conversion results in no wide characters. If the *space* and + flags both appear, the *space* flag is ignored.
#	Convert the result to an alternative form. Specifically, for o conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For x (or X) conversions, 0x (or 0X) is prefixed to nonzero results. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *width* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag − has been used.

If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *precision* field indicates:

- the minimum number of digits to appear for the d, i, o, u, x, and X conversions
- the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions
- the maximum number of significant digits for the g and G conversions
- the maximum number of bytes to be written for s conversions

The precision takes the form of a period, ., followed either by an asterisk, * or by an optional decimal integer. If neither the * or integer is specified, the precision is assumed to be zero. If the asterisk, *, is used instead of a decimal number, the int argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *length* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a short int or unsigned short int.
h	When used with the n conversion specifier, indicates that the pointer argument points to a short int.
hh	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a signed char or unsigned char.
j	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is aintmax_t or uintmax_t.
j	When used with the n conversion specifier, indicates that the pointer points to a intmax_t.
l	When used with d, i, o, u, x, or X conversion specifiers, converts the argument value to a long int or unsigned long int.

.....continued

Modifier	Meaning
l	When used with the n conversion specifier, indicates that the pointer points to a long int.
l	When used with the c conversion specifier, indicates that the argument value is a wide character (wint_t type).
l	When used with the s conversion specifier, indicates that the argument value is a wide string (wchar_t type).
l	When used the e, E, f, F, g, G, has no effect.
ll	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a long long int or unsigned long long int.
ll	When used with the n conversion specifier, indicates that the pointer points to a long long int.
ll	MPLAB XC16: When used with the s conversion specifier, indicates that the string pointer is an __eds__ pointer. Other compilers: The modifier is silently ignored.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
t	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a ptrdiff_t or the corresponding unsigned integer type.
t	When used with the n conversion specifier, indicates that the pointer points to a ptrdiff_t.
z	When used with d, i, o, u, x, or X conversion specifiers, indicates that the argument value is a size_t_t or the corresponding signed integer type.
z	When used with the n conversion specifier, indicates that the pointer points to a size_t_t.

The conversion *specifiers* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [−]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point wide character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a wide character (as if by calling btowc) and the resulting wide character is written. When the l modifier is present, the wint_t argument is converted to wchar_t and written.
d	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [−]*d.ddde±dd*, where there is one digit before the decimal-point wide character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.

.....continued

Specifier	Argument value type	Printing notes
f, F	double	Converted to decimal notation using the general form [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of wide characters written so far is written to the object pointed to by the pointer. No wide characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (address) held by the pointer.
s	char array, or wchar_t array	A string contain multibyte characters that are converted to wide characters (as if by calling mbrtowc) and written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written. When the l modifier is present, wide characters from the array are written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % wide character is printed.

Example

```
#include <wchar.h>

void errmsg(const wchar_t * fmt, ...)
{
  va_list ap;

  va_start(ap, fmt);
  vwprintf(fmt, ap);
  va_end(ap);
}

int main(void)
{
  int num = 3;

  errmsg(L"Error: The letter '%c' is not %s\n", 'a', "an integer value.");
  errmsg(L"Error: Requires %d%ls%lc", num, L" or more characters.", L'\n');
}
```

Example Output

```
Error: The letter 'a' is not an integer value.
Error: Requires 3 or more characters.
```

5.23.30 vwscanf Function

Scans formatted text from the `stdin` stream, using a wide format string and a variable length argument list.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int vwscanf(const wchar_t * restrict format, va_list arg);
```

Arguments

`format`	format control wide string
`arg`	variable argument list of objects to be assigned values read in

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. `EOF` is returned if end-of-file is encountered before the first conversion or if an error occurs.

Remarks

This function can read input from the `stdin` stream. The `arg` argument must have been initialized by the `va_start` macro.

The `format` string can be composed of white space wide characters, which reads input up to the first non-white-space wide character in the input; or ordinary wide characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` wide character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent wide character followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`length`*`]`*`specifier`*

The presence of `*` immediately after the percent wide character indicates that the read and converted value should not be assigned to any object.

The *`width`* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *`length`* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `short int` or `unsigned short int` object referenced by the pointer argument.
`hh`	When used with `d`, `i`, `o`, `u`, `x`, `X`, or `n` indicates that this conversion specifier assigns to a `signed char` or `unsigned char` object referenced by the pointer argument.

.....continued

Modifier	Meaning
l	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long int or unsigned long int object referenced by the pointer argument.
l	When used the a, A, e, E, f, F, g, or G indicates that this conversion specifier assigns to a double object referenced by the pointer argument.
l	When used the c, s, or [indicates that this conversion specifier assigns to a wchar_t object referenced by the pointer argument.
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single wide character or the number of characters specified by the field width if present. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array of sufficient size. No null character is added. If an l length modifier is present, the input shall be a wide character array of sufficient size. No null wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.

.....continued

Specifier	Receiving object type	Matches
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	int	No input is consumed but the number of wide characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fwprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space wide characters, which is written to the array argument. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large of sufficient size. A terminating null character is appended. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide character array of sufficient size. A terminating null wide character is appended.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
%		Matches a single % wide character. No assignment takes place.
[	char array, or wchar_t array	Matches all the wide characters in the input that have been specified between the [and trailing] wide character, unless the wide character after the opening bracket is a circumflex, ^, in which case a match is only made with wide characters that do not appear in the brackets. If the conversion specifier begins with [] or [^], the right bracket wide character will match the input and the next following right bracket wide character is the matching right bracket that ends the specification. If no l length modifier is present, characters from the input field are converted to amultibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide array of sufficient size.A terminating null wide character will be appended.

Example

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

int ReadWideStuff (const wchar_t * format, ...)
{
  va_list args;
  int items;

  va_start (args, format);
  items = vwscanf (format, args);
  va_end (args);

  return items;
}

int main ()
{
  int val, items;
  wchar_t str[100];

  items = ReadWideStuff(L"%ls%d", str, &val);
  printf("%d items read in\n", items);
}
```

Example Input

Provided on the `stdin` stream:

```
Message 4 you
```

Example Output

```
2 items read in
```

5.23.31 wcrtomb Function

Convert a wide character to a multibyte character sequence.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
```

Arguments

s	the mulitbyte character sequence to convert
wc	a pointer to the array to hold the wide character
ps	a pointer to a `mbstate_t` object that holds the current conversion state

Return Value

The function returns the number of bytes stored in the array object `s` (including any shift sequences). If `wc` is not a valid wide character, `(size_t)(-1)` is returned.

Remarks

If `s` is a null pointer, the function is equivalent to the call `wcrtomb(buf, L'\0', ps)`, where `buf` represents an internal buffer

If `s` is not a null pointer, the function determines the number of bytes needed to represent the multibyte character sequence that corresponds to the wide character given by `wc` (including any shift sequences), and stores at most `MB_CUR_MAX` bytes of that multibyte character representation in the array whose first element is pointed to by `s`. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

An encoding error occurs if `wc` is not a valid wide character, and the function will store the value of the macro `EILSEQ` in `errno` and the conversion state is unspecified.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void) {
  const wchar_t * pt = L"The wide ocean";
  char buffer [MB_CUR_MAX];
  size_t length, i;
  mbstate_t mbs;
  char needNL = 0;

  mbrlen (NULL, 0, &mbs);   /* initialize mbs */

  while (*pt) {
    length = wcrtomb(buffer, *pt, &mbs);
    if((length==0) || (length>MB_CUR_MAX))
      break;
    else
      needNL = 1;
    putchar ('[');
    for(i=0; i<length; i++)
      putchar(buffer[i]);
    putchar(']');
    pt++;
  }
  if(needNL)
    putchar('\n');
}
```

Example Output

```
[T][h][e][ ][w][i][d][e][ ][o][c][e][a][n]
```

5.23.32 wcscat Function

Concatenate a wide string with another.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t * wcscat( wchar_t * restrict s1, const wchar_t * restrict s2);
```

Arguments

s1	the wide string to append to
s2	the wide string to append

Return Value

A copy of `s1`.

Remarks

The function writes a copy of the wide string pointed to by `s2` (including the terminating null wide character) to the location of the null wide character at the end of the wide string pointed to by `s1`.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[40] = L"A long";

  wcscat(ws, L" time ago...");
  wprintf(L"the complete wide string is \"%ls\"\n", ws);
}
```

Example Output

```
the complete wide string is "A long time ago..."
```

5.23.33 wcschr Function

Locates the first occurrence of a wide character in a wide string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Arguments

s	the wide string in which to search
c	the wide character to search for

Return Value

The function returns a pointer to the wide character, or a null pointer if the wide character was not found.

Remarks

The function locates the first occurrence of the wide character `c` in the wide string pointed to by `s`. The terminating null wide character is searchable.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
```

```
{
  wchar_t ws[50] = L"What time is it?";
  wchar_t wc1 = L'm', wc2 = L'y';
  wchar_t *ptr;
  int res;

  wprintf(L"ws : %ls\n\n", ws);

  ptr = wcschr(ws, wc1);
  if (ptr != NULL)
  {
    res = ptr - ws + 1;
    wprintf(L"%c found at position %d\n", wc1, res);
  }
  else
    wprintf(L"%c not found\n", wc1);
  wprintf(L"\n");

  ptr = wcschr(ws, wc2);
  if (ptr != NULL)
  {
    res = ptr - ws + 1;
    wprintf(L"%c found at position %d\n", wc2, res);
  }
  else
    wprintf(L"%c not found\n", wc2);
}
```

Example Output

```
ws : What time is it?

m found at position 8

y not found
```

5.23.34 wcscmp Function

Compare two wide strings.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int wcscmp(const wchar_t * restrict s1, const wchar_t * restrict s2);
```

Arguments

s1	one wide string to compare
s2	the other wide string to compare

Return Value

Returns a positive integer if `s1` is greater than `s2`, zero if `s1` is equal to `s2` or a negative number if `s1` is less than `s2`.

Remarks

The function returns a value based on the first wide character that differs between `s1` and `s2`. Wide characters that follow a null wide character are not compared.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  wchar_t buf1[50] = L"Where is the time?";
  wchar_t buf2[50] = L"Where did they go?";
  wchar_t buf3[50] = L"Why?";
  int res;

  wprintf(L"buf1 : %ls\n", buf1);
  wprintf(L"buf2 : %ls\n", buf2);
  wprintf(L"buf3 : %ls\n\n", buf3);

  res = wcscmp(buf1, buf2);
  if (res < 0)
    wprintf(L"buf1 comes before buf2\n");
  else if (res == 0)
    wprintf(L"buf1 and buf2 are equal\n");
  else
   wprintf(L"buf2 comes before buf1\n");
  wprintf(L"\n");

  res = wcscmp(buf1, buf3);
  if (res < 0)
    wprintf(L"buf1 comes before buf3\n");
  else if (res == 0)
    wprintf(L"buf1 and buf3 are equal\n");
  else
    wprintf(L"buf3 comes before buf1\n");
  wprintf(L"\n");

  res = wcscmp(L"Why?", buf3);
  if (res < 0)
    wprintf(L"\"Why?\" comes before buf3\n");
  else if (res == 0)
    wprintf(L"buf1 and \"Why?\" are equal\n");
  else
    wprintf(L"buf3 comes before \"Why?\"\n");
}
```

Example Output

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

buf2 comes before buf1

buf1 comes before buf3

buf1 and "Why?" are equal
```

5.23.35 wcscoll Function

Compare two wide strings based on current locale.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int wcscoll(const wchar_t * restrict s1, const wchar_t * restrict s2);
```

Arguments

s1	one wide string to compare
s2	the other wide string to compare

Return Value

Returns a positive integer if `s1` is greater than `s2`, zero if `s1` is equal to `s2` or a negative number if `s1` is less than `s2`.

Remarks

The function returns a value based on the first wide character that differs between `s1` and `s2`. Wide characters that follow a null wide character are not compared.

The comparison is based on the `LC_COLLATE` category of the current locale. As MPLAB XC8 does not implement locales, `wcscoll()` is equivalent to `wcscmp()` for that implementation.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t buf1[50] = L"Where is the time?";
  wchar_t buf2[50] = L"Where did they go?";
  wchar_t buf3[50] = L"Why?";
  int res;

  wprintf(L"buf1 : %ls\n", buf1);
  wprintf(L"buf2 : %ls\n", buf2);
  wprintf(L"buf3 : %ls\n\n", buf3);

  res = wcscoll(buf1, buf2);
  if (res < 0)
    wprintf(L"buf1 comes before buf2\n");
  else if (res == 0)
    wprintf(L"buf1 and buf2 are equal\n");
  else
   wprintf(L"buf2 comes before buf1\n");
  wprintf(L"\n");

  res = wcscoll(buf1, buf3);
  if (res < 0)
    wprintf(L"buf1 comes before buf3\n");
  else if (res == 0)
    wprintf(L"buf1 and buf3 are equal\n");
  else
    wprintf(L"buf3 comes before buf1\n");
  wprintf(L"\n");

  res = wcscoll(L"Why?", buf3);
  if (res < 0)
    wprintf(L"\"Why?\" comes before buf3\n");
  else if (res == 0)
    wprintf(L"buf1 and \"Why?\" are equal\n");
  else
    wprintf(L"buf3 comes before \"Why?\"\n");
}
```

Example Output

The output of this program might vary based on the `LC_COLLATE` category of the current locale.

5.23.36 wcscpy Function

Copy a wide string to an array.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t * wcscpy( wchar_t * restrict s1, const wchar_t * restrict s2);
```

Arguments

s1	the array to hold the copied wide string
s2	the wide string to copy

Return Value

A copy of `s1`.

Remarks

The function copies the wide string pointed to by `s2` into the array `s1`. The null wide character is copied.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[40];

  wcscpy(ws, L"a literal wide string");
  wprintf(L"the wide string referenced by ws is \"%ls\"\n", ws);
}
```

Example Output

```
the wide string referenced by ws is "a literal wide string"
```

5.23.37 wcscspn Function

Calculate the number of consecutive wide characters at the beginning of a wide string that are not contained in a set of wide characters.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t wcscspn(const wchar_t * s1, const wchar_t * s2);
```

Arguments

s1	the wide string in which to search
s2	the wide characters

Return Value

Returns the length of the segment in `s1` not containing wide characters found in `s2`.

Remarks

This function will determine the number of consecutive wide characters from the beginning of the wide string `s1` that are not contained in the wide string `s2`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  wchar_t ws1[20] = L"hello";
  wchar_t ws2[20] = L"aeiou";
  wchar_t ws3[20] = L"animal";
  wchar_t ws4[20] = L"xyz";
  int res;

  res = wcscspn(ws1, ws2);
  wprintf(L"wcscspn(\"%ls\", \"%ls\") = %d\n", ws1, ws2, res);

  res = wcscspn(ws3, ws2);
  wprintf(L"wcscspn(\"%ls\", \"%ls\") = %d\n", ws3, ws2, res);

  res = wcscspn(ws3, ws4);
  wprintf(L"wcscspn(\"%ls\", \"%ls\") = %d\n", ws3, ws4, res);
}
```

Example Output

```
wcscspn("hello", "aeiou") = 1
wcscspn("animal", "aeiou") = 0
wcscspn("animal", "xyz") = 6
```

5.23.38 wcsftime Function

Formats the time structure to a wide string based on the format parameter.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize, const wchar_t * restrict format,
const struct tm * restrict timeptr);
```

Arguments

s	the array of wide characters to modify
maxsize	the maximum number of wide characters to be written
format	the wide string contain format specifiers
timeptr	pointer to `tm` data structure

Return Value

Returns the number of wide characters placed into the array pointed to by `s` (excluding the terminating null wide character) if that number is not more than `maxsize`; otherwise the function returns 0 and the contents of the array are indeterminate.

Remarks

The function places no more than `maxsize` wide characters into the array pointed to by `s` as controlled by the format wide string, which consists of zero or more conversion specifiers and ordinary wide characters. A conversion specifier

consists of a % wide character, possibly followed by an E or O modifier wide character, followed by a wide character that determines the behavior of the conversion specifier. All ordinary wide characters (including the terminating null wide character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined.

Each conversion specifier is replaced by appropriate wide characters, which are determined using the LC_TIME category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by timeptr, as indicated in the description. In the "C" locale, the E and O modifiers are ignored and some specifiers are replaced by different wide strings, which are indicated in square brackets. The %g, %G, and %V specifiers give values according to the ISO 8601 week-based year. If any of the specified values are outside the normal range, the characters stored are unspecified.

Specifier	Replacement
%a	abbreviated weekday name, derived from tm_wday ["C" locale: the first three characters of %A]
%A	full weekday name, derived from tm_wday ["C" locale: one of Sunday, Monday, ... , Saturday]
%b	abbreviated month name, derived from tm_mon ["C" locale: the first three characters of %B]
%B	full month name, derived from tm_mon ["C" locale: one of January, February, ... , December. %c equivalent to %a %b %e %T %Y]
%c	appropriate date and time representation [C locale: equivalent to %a %b %e %T %Y]
%C	year, derived from tm_year, divided by 100 and truncated to an integer, as a decimal number (00−99)
%d	day of the month as a decimal number (01-31), derived from tm_mday
%D	equivalent to %m/%d/%y, derived from tm_mon, tm_mday, and tm_year
%e	day of the month, derived from tm_mday, as a decimal number (1−31); a single digit is preceded by a space
%F	equivalent to %Y-%m-%d, derived from tm_year, tm_mon, and tm_mday
%g	last 2 digits of the week-based year, derived from tm_year, tm_wday, and tm_yday, as a decimal number (00−99)
%G	week-based year, derived from tm_year, tm_wday, and tm_yday, as a decimal number (e.g., 1997)
%H	equivalent to %b, derived from tm_mon
%H	hour (24-hour clock), derived from tm_hour, as a decimal number (00−23)
%I	hour (12-hour clock), derived from tm_hour, as a decimal number (01−12)
%j	day of the year, derived from tm_yday, as a decimal number (001−366)
%m	month, derived from tm_mon, as a decimal number (01−12)
%M	minute, derived from tm_min, as a decimal number (00-59)
%n	new-line character
%p	AM/PM designator, derived from tm_hour ["C" locale" one of AM or PM]
%r	12-hour clock time, derived from tm_hour, tm_min, and tm_sec ["C" locale: equivalent to %I:%M:%S %p]
%R	equivalent to %H:%M, derived from tm_hour and tm_min
%S	second, derived from tm_sec, as a decimal number (00-60)
%t	horizontal-tab character

.....continued	
Specifier	**Replacement**
%T	equivalent to %H:%M:%S, derived from tm_hour, tm_min, and tm_sec ["C" locale: equivalent to %T]
%u	ISO 8601 weekday, derived from tm_wday, as a decimal number (1−7), with Monday being 1.
%U	week number of the year, derived from tm_year, tm_wday, tm_yday, where Sunday is the first day of week 1 (00-53)
%V	ISO 8601 week number, derived from tm_year, tm_wday, tm_yday, as a decimal number (01−53)
%w	weekday, derived from tm_wday, as a decimal number (0−6), where Sunday is 0
%W	week number of the year (the first Monday as the first day of week 1), derived from tm_year, tm_wday, tm_yday, as a decimal number (00−53)
%x	appropriate date representation ["C" locale: equivalent to %m/%d/%y]
%X	appropriate time representation[C locale: equivalent to %T]
%y	last 2 digits of the year, derived from tm_year, as a decimal number (00−99)
%Y	year, derived from tm_year, as a decimal number (e.g., 1997)
%z	offset from UTC in ISO 8601 format, derived from tm_isdst, where −0430 implies 4 hours 30 minutes behind UTC, west of Greenwich; or no characters if no time zone is determinable
%Z	time zone name or abbreviation, derived from tm_isdst, or no characters if no time zone is determinable
%%	percent character, %

Some of the above specified can be modified using the E or O modifier characters to indicate an alternative format or specification. In the "C" locale, these modifiers are ignored. If the alternative format or specification does not exist for the current locale, the modifier is ignored.

Table 5-7. Modifiers used with specifiers

Modified specifier	Replacement
%Ec	locale’s alternative date and time representation
%EC	name of the base year (period) in the locale’s alternative representation
%Ex	locale’s alternative date representation
%EX	locale’s alternative time representation
%Ey	offset from %EC (year only) in the locale’s alternative representation
%EY	locale’s full alternative year representation
%Od	day of the month, using the locale’s alternative numeric symbols (filled as needed with leading zeros, or with leading spaces if there is no alternative symbol for zero)
%Oe	day of the month, using the locale’s alternative numeric symbols (filled as needed with leading spaces)
%OH	hour (24-hour clock), using the locale’s alternative numeric symbol
%OI	hour (12-hour clock), using the locale’s alternative numeric symbols
%Om	month, using the locale’s alternative numeric symbols

.....continued	
Modified specifier	**Replacement**
%OM	minutes, using the locale's alternative numeric symbol
%OS	seconds, using the locale's alternative numeric symbols
%Ou	ISO 8601 weekday as a number in the locale's alternative representation, where Monday is 1
%OU	week number, using the locale's alternative numeric symbols
%OV	ISO 8601 week number, using the locale's alternative numeric symbols
%Ow	weekday as a number, using the locale's alternative numeric symbols
%OW	week number of the year, using the locale's alternative numeric symbols
%Oy	last 2 digits of the year, using the locale's alternative numeric symbols

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  time_t timer, whattime;
  struct tm * newtime;
  wchar_t buf[128];

  timer = 1066668182; /* Mon Oct 20 16:43:02 2003 */
  /* localtime allocates space for structure */
  newtime = localtime(&timer);
  wcsftime(buf, 128, L"It was a %A, %d days into the "
           L"month of %B in the year %Y.\n", newtime);
  wprintf(buf);
  wcsftime(buf, 128, L"It was %W weeks into the year "
           L"or %j days into the year.\n", newtime);
  wprintf(buf);
}
```

Example Output

```
It was a Tuesday, 21 days into the month of October in the year 2003.
It was 42 weeks into the year or 294 days into the year.
```

5.23.39 wcslen Function

Determines the length of a wide string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t wcslen(const wchar_t * s);
```

Arguments

s the wide string

Return Value

Returns the nummber of wide characters present in the string, excluding the terminating null wide character.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  wchar_t ws1[20] = L"We are here";
  wchar_t ws2[20] = L"";
  wchar_t ws3[20] = L"Why me?";

  wprintf(L"ws1 : %ls\n", ws1);
  wprintf(L"\t(wsing length = %d characters)\n\n", wcslen(ws1));
  wprintf(L"ws2 : %ls\n", ws2);
  wprintf(L"\t(wsing length = %d characters)\n\n", wcslen(ws2));
  wprintf(L"ws3 : %ls\n", ws3);
  wprintf(L"\t(wsing length = %d characters)\n\n\n", wcslen(ws3));

}
```

Example Output

```
ws1 : We are here
     (wsing length = 11 characters)

ws2 :
     (wsing length = 0 characters)

ws3 : Why me?
     (wsing length = 7 characters)
```

5.23.40 wcsncat Function

Concatenate a wide string with another.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t * wcsncat( wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Arguments

s1 the wide string to append to

s2 the wide string to append

n the maximum number of wide characters to append

Return Value

A copy of `s1`.

Remarks

The function appends no more than `n` wide characters of the wide string pointed to by `s2` to the location of the null wide character at the end of the wide string pointed to by `s1`, stopping if a null wide character is encountered. A null wide character is then appended to the end of `s1`, implying that up to `n`+1 wide characters could be appended.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[40] = L"A long";

  wcsncat(ws, L" time ago...", 5);
  wprintf(L"the complete wide string is \"%ls\"\n", ws);
}
```

Example Output

```
the complete wide string is "A long time"
```

5.23.41 wcsncmp Function

Compare two wide strings.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int wcsncmp(const wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Arguments

s1	one wide string to compare
s2	the other wide string to compare
n	the maximum number of wide characters to compare

Return Value

Returns a positive integer if `s1` is greater than `s2`, zero if `s1` is equal to `s2` or a negative number if `s1` is less than `s2`.

Remarks

The function returns a value based on the first wide character that differs between `s1` and `s2` after at most `n` comparisons. Wide characters that follow a null wide character are not compared.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t buf1[50] = L"Where is the time?";
  wchar_t buf2[50] = L"Where did they go?";
  wchar_t buf3[50] = L"Why?";
  int res;

  wprintf(L"buf1 : %ls\n", buf1);
  wprintf(L"buf2 : %ls\n", buf2);
  wprintf(L"buf3 : %ls\n\n", buf3);

  res = wcsncmp(buf1, buf2, 6);
```

```
  if (res < 0)
    wprintf(L"buf1 comes before buf2\n");
  else if (res == 0)
    wprintf(L"6 characters of buf1 and buf2 are equal\n");
  else
   wprintf(L"buf2 comes before buf1\n");
  wprintf(L"\n");

  res = wcsncmp(buf1, buf2, 20);
  if (res < 0)
    wprintf(L"buf1 comes before buf2\n");
  else if (res == 0)
    wprintf(L"20 characters of buf1 and buf2 are equal\n");
  else
    wprintf(L"buf2 comes before buf1\n");
  wprintf(L"\n");

  res = wcsncmp(buf1, buf3, 20);
  if (res < 0)
    wprintf(L"buf1 comes before buf3\n");
  else if (res == 0)
    wprintf(L"20 characters of buf1 and buf3 are equal\n");
  else
    wprintf(L"buf3 comes before buf1\n");
}
```

Example Output

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

5.23.42 wcsncpy Function

Copy a wide string to an array.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t * wcsncpy( wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Arguments

s1	the array to hold the copied wide string
s2	the wide string to copy
n	the maximum number of wide characters to copy

Return Value

A copy of `s1`.

Remarks

The function copies the wide string pointed to by `s2` into the array `s1`. No further wide characters are read from `s1` if `n` wide characters have been copied or a null wide character has been copied. If less than `n` wide characters have been written to `s1`, null wide characters are appended to `s1` until it is `n` wide characters wide.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[40];

  wcsncpy(ws, L"a literal wide string", 40);
  wprintf(L"the wide string referenced by ws is \"%ls\"\n", ws);
}
```

Example Output

```
the wide string referenced by ws is "a literal wide string"
```

5.23.43 wcspbrk Function

Search a wide string for the first occurrence of a wide character from a specified set of wide characters.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

`wchar_t *wcspbrk(const wchar_t * s1, const wchar_t * s2);`

Arguments

s1	the wide string in which to search
s2	the wide characters to search for

Return Value

Returns a pointer to the matched wide character in `s1` if found; otherwise, returns a null pointer.

Remarks

This function will search `s1` for the first occurrence of a character contained in `s2`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  wchar_t ws1[20] = L"What time is it?";
  wchar_t ws2[20] = L"xyz";
  wchar_t ws3[20] = L"eou?";
  wchar_t *ptr;
  int res;

  wprintf(L"wcspbrk(\"%ls\", \"%ls\")\n", ws1, ws2);
  ptr = wcspbrk(ws1, ws2);
  if (ptr != NULL)
```

```
  {
    res = ptr - ws1 + 1;
    wprintf(L"match found at position %d\n", res);
  }
  else
    wprintf(L"match not found\n");
  wprintf(L"\n");

  wprintf(L"wcspbrk(\"%ls\", \"%ls\")\n", ws1, ws3);
  ptr = wcspbrk(ws1, ws3);
  if (ptr != NULL)
  {
    res = ptr - ws1 + 1;
    wprintf(L"match found at position %d\n", res);
  }
  else
    wprintf(L"match not found\n");
}
```

Example Output

```
wcspbrk("What time is it?", "xyz")
match not found

wcspbrk("What time is it?", "eou?")
match found at position 9
```

5.23.44 wcsrchr Function

Locates the last occurrence of a wide character in a wide string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t * wcsrchr(const wchar_t *s, wchar_t c);
```

Arguments

s	the wide string in which to search
c	the wide character to search for

Return Value

The function returns a pointer to the wide character, or a null pointer if the wide character was not found.

Remarks

The function locates the last occurrence of the wide character `c` in the wide string pointed to by `s`. The terminating null wide character is searchable.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[50] = L"What time is it?";
  wchar_t wc1 = L'i', wc2 = L'y';
  wchar_t *ptr;
```

```
  int res;

  wprintf(L"ws : %ls\n\n", ws);

  ptr = wcsrchr(ws, wc1);
  if (ptr != NULL)
  {
    res = ptr - ws + 1;
    wprintf(L"%c found at position %d\n", wc1, res);
  }
  else
    wprintf(L"%c not found\n", wc1);
  wprintf(L"\n");

  ptr = wcsrchr(ws, wc2);
  if (ptr != NULL)
  {
    res = ptr - ws + 1;
    wprintf(L"%c found at position %d\n", wc2, res);
  }
  else
    wprintf(L"%c not found\n", wc2);
}
```

Example Output

```
ws : What time is it?

i found at position 14

y not found
```

5.23.45 wcsrtombs Function

Convert a wide character string to a multibyte character sequence.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len,
mbstate_t * restrict ps);
```

Arguments

dst	a pointer to the object to hold the converted multibyte character sequence
src	the wide character sequence to convert
len	the maximum number of multibyte characters to store
ps	a pointer to a `mbstate_t` object that holds the current conversion state

Return Value

The function returns the number of bytes in the multibyte character sequence, not including the terminating null character (if any). If `src` cannot be converted to a mutlibyte character sequence, `(size_t)(-1)` is returned.

Remarks

The function converts a wide character string indirectly pointed to by `src` beginning in the conversion state, described by the object pointed to by `ps`, into a sequence of corresponding multibyte characters, storing the converted characters (including a terminating null character) into the array pointed to by `dst`, provided `dst` is not a

null pointer. Conversion stops when encountering a wide character that cannot be converted, or when `len` multibyte characters have been stored. Each conversion takes place as if by a call to the `wcrtomb()` function. By having the argument to specify the current conversion state, this function is restartable.

An encoding error occurs if `src` is not a valid multibyte character sequence, when the function stores the value of the macro `EILSEQ` in `errno`, and the conversion state is unspecified.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void) {
  const wchar_t * ws = L"Volume: 2L±0.5mL";
  char buffer [20];
  size_t length;
  mbstate_t mbs;

  setlocale(LC_CTYPE, "");
  mbrlen (NULL, 0, &mbs);   /* initialize mbs */

  length = wcsrtombs(buffer, &ws, 20, &mbs);
  wprintf(L"multibyte string \"%s\" has length %d\n", buffer, length);
}
```

Example Output

```
multibyte string "Volume: 2L±0.5mL" has length 17
```

5.23.46 wcsspn Function

Calculate the number of consecutive wide characters at the beginning of a wide string that are contained in a set of wide characters.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t wcsspn(const wchar_t * s1, const wchar_t * s2);
```

Arguments

s1	the wide string in which to search
s2	the wide characters to search for

Return Value

Returns the length of the segment in `s1` containing wide characters found in `s2`.

Remarks

This function will determine the number of consecutive wide characters from the beginning of the wide string `s1` that are contained in the wide string `s2`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  wchar_t ws1[20] = L"hello";
  wchar_t ws2[20] = L"aeiou";
  wchar_t ws3[20] = L"animal";
  wchar_t ws4[20] = L"xyz";
  int res;

  res = wcsspn(ws1, ws2);
  wprintf(L"wcsspn(\"%ls\", \"%ls\") = %d\n", ws1, ws2, res);

  res = wcsspn(ws3, ws2);
  wprintf(L"wcsspn(\"%ls\", \"%ls\") = %d\n", ws3, ws2, res);

  res = wcsspn(ws3, ws4);
  wprintf(L"wcsspn(\"%ls\", \"%ls\") = %d\n", ws3, ws4, res);
}
```

Example Output

```
wcsspn("hello", "aeiou") = 0
wcsspn("animal", "aeiou") = 1
wcsspn("animal", "xyz") = 0
```

5.23.47 wcsstr Function

Locates the first occurrence of a wide string in a wide string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t *wcschr(const wchar_t *s1, const wchar_t * s2);
```

Arguments

s1	the wide string in which to search
s2	the wide string to search for

Return Value

Returns the address of the first element that matches the wide string if found; otherwise, returns a null pointer.

Remarks

This function will find the first occurrence of the wide string `s2` (excluding the null terminator) within the string `s1`. If `s2` points to a zero length string, `s1` is returned.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
```

```
  wchar_t ws1[20] = L"What time is it?";
  wchar_t ws2[20] = L"is";
  wchar_t ws3[20] = L"xyz";
  wchar_t *ptr;
  int res;

  wprintf(L"ws1 : %ls\n", ws1);
  wprintf(L"ws2 : %ls\n", ws2);
  wprintf(L"ws3 : %ls\n\n", ws3);

  ptr = wcsstr(ws1, ws2);
  if (ptr != NULL)
  {
   res = ptr - ws1 + 1;
   wprintf(L"\"%ls\" found at position %d\n", ws2, res);
  }
  else
    wprintf(L"\"%ls\" not found\n", ws2);
  wprintf(L"\n");

  ptr = wcsstr(ws1, ws3);
  if (ptr != NULL)
  {
    res = ptr - ws1 + 1;
    wprintf(L"\"%ls\" found at position %d\n", ws3, res);
  }
  else
    wprintf(L"\"%ls\" not found\n", ws3);
}
```

Example Output

```
ws1 : What time is it?
ws2 : is
ws3 : xyz

"is" found at position 11

"xyz" not found
```

5.23.48 wcstof Function

Convert wide string to a single precision floating-point value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
float wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Arguments

nptr	the wide string to attempt to convert
endptr	pointer to store the address of the remainder of the wide string that was not converted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, `HUGE_VAL` is returned with a sign being that of the correct value. If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number in the return type.

Remarks

The wcstof function attempts to convert a portion of the wide string pointed to by nptr to a float floating-point value.

The initial section consists of any white-space wide characters.

The subject section represents the floating-point constant to convert and consists of an optional plus or minus sign then one of the following.

- Decimal digits optionally containing a decimal-point wide character, then an optional exponent part, being e or E followed by an option sign and decimal digits
- A 0x or 0X, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary exponent part, being p or P, and option sign, and decimal digits.
- one of INF or INFINITY, ignoring case
- NAN, ignoring case, optionally followed by any sequence contain digits or non-digits:

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial section and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

The final section consists of wide characters in the wide string that were unrecognized in the subject section and which will include the terminating null wide character of the wide string. This section of the wide string will be accessible via the address stored to endptr, provided endptr is not a null pointer.

The value of the macro ERANGE is stored in errno if the correct value of the conversion is outside the range of representable values. The errno variable is not set to ERANGE if the result underflows.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[] = L"2000.5 -6.0E-3 0x70FF INFINITY";
  wchar_t * pEnd;
  float d1, d2, d3, d4;

  d1 = wcstof(ws, &pEnd);
  d2 = wcstof(pEnd, &pEnd);
  d3 = wcstof(pEnd, &pEnd);
  d4 = wcstof(pEnd, NULL);
  wprintf(L"The converted string values are: %g, %g, %g, and %g.\n", d1, d2, d3, d4);
}
```

Example Output

```
The converted string values are: 2000.5, -0.006, 28927, and inf.
```

5.23.49 wcstod Function

Convert wide string to a double precision floating-point value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

<wchar.h>

Prototype

```
double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Arguments

nptr the wide string to attempt to convert

endptr pointer to store the address of the remainder of the wide string that was not converted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, HUGE_VAL is returned with a sign being that of the correct value. If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number in the return type.

Remarks

The wcstod function attempts to convert a portion of the wide string pointed to by nptr to a double floating-point value.

The initial section consists of any white-space wide characters.

The subject section represents the floating-point constant to convert and consists of an optional plus or minus sign then one of the following.

- Decimal digits optionally containing a decimal-point wide character, then an optional exponent part, being e or E followed by an option sign and decimal digits
- A 0x or 0X, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary exponent part, being p or P, and option sign, and decimal digits.
- one of INF or INFINITY, ignoring case
- NAN, ignoring case, optionally followed by any sequence contain digits or non-digits:

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial section and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

The final section consists of wide characters in the wide string that were unrecognized in the subject section and which will include the terminating null wide character of the wide string. This section of the wide string will be accessible via the address stored to endptr, provided endptr is not a null pointer.

The value of the macro ERANGE is stored in errno if the correct value of the conversion is outside the range of representable values. The errno variable is not set to ERANGE if the result underflows.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[] = L"2000.5 -6.0E-3 0x70FF INFINITY";
  wchar_t * pEnd;
  double d1, d2, d3, d4;

  d1 = wcstod(ws, &pEnd);
  d2 = wcstod(pEnd, &pEnd);
  d3 = wcstod(pEnd, &pEnd);
  d4 = wcstod(pEnd, NULL);
  wprintf(L"The converted string values are: %g, %g, %g, and %g.\n", d1, d2, d3, d4);
}
```

Example Output

```
The converted string values are: 2000.5, -0.006, 28927, and inf.
```

5.23.50 wcstold Function

Convert wide string to a long double precision floating-point value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

<wchar.h>

Prototype

```
long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Arguments

nptr	the wide string to attempt to convert
endptr	pointer to store the address of the remainder of the wide string that was not converted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, HUGE_VAL is returned with a sign being that of the correct value. If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number in the return type.

Remarks

The wcstold function attempts to convert a portion of the wide string pointed to by nptr to a long double floating-point value.

The initial section consists of any white-space wide characters.

The subject section represents the floating-point constant to convert and consists of an optional plus or minus sign then one of the following.

- Decimal digits optionally containing a decimal-point wide character, then an optional exponent part, being e or E followed by an option sign and decimal digits
- A 0x or 0X, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary exponent part, being p or P, and option sign, and decimal digits.
- one of INF or INFINITY, ignoring case
- NAN, ignoring case, optionally followed by any sequence contain digits or non-digits:

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial section and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

The final section consists of wide characters in the wide string that were unrecognized in the subject section and which will include the terminating null wide character of the wide string. This section of the wide string will be accessible via the address stored to endptr, provided endptr is not a null pointer.

The value of the macro ERANGE is stored in errno if the correct value of the conversion is outside the range of representable values. The errno variable is not set to ERANGE if the result underflows.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[] = L"2000.5 -6.0E-3 0x70FF INFINITY";
  wchar_t * pEnd;
  long double d1, d2, d3, d4;

  d1 = wcstold(ws, &pEnd);
  d2 = wcstold(pEnd, &pEnd);
  d3 = wcstold(pEnd, &pEnd);
```

```
  d4 = wcstold(pEnd, NULL);
  wprintf(L"The converted string values are: %Lg, %Lg, %Lg, and %Lg.\n", d1, d2, d3, d4);
}
```

Example Output

```
The converted string values are: 2000.5, -0.006, 28927, and inf.
```

5.23.51 wcstok Function

Break a wide string into tokens.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t * wcschr(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t **
restrict ptr);
```

Arguments

s1	the wide string to tokenize
s2	a wide string of characters used as delimiters
ptr	a pointer to a user-supplied pointer used to hold information over subsequent calls to this function

Return Value

Returns the address of the first element that matches the wide string if found; otherwise, returns a null pointer.

Remarks

This function will find the first occurrence of the wide string `s2` (excluding the null terminator) within the string `s1`. If `s2` points to a zero length string, `s1` is returned.

The address of the wide string to tokenize is passed as the first argument in a tokenizing sequence. Subsequent calls should specify a null pointer as the first argument and the object pointed to by `ptr` is required to have the value stored by the previous call in the sequence. The wide character string used as the delimiters may be different with each call.

The first call in the sequence searches the wide string pointed to by `s1` for the first wide character that is *not* contained in the wide string pointed to by `s2`. If no match is found, and a null pointer is returned; otherwise this wide character becomes the start of the first token.

If a wide character from the `s2` wide string can be subsequently found from the beginning of this token, it is replaced with a null wide character, thus correctly terminating the token as a wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by `s1`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[] = L"- This, as they say, is the end.";
  wchar_t * pwc;
  wchar_t * ptr;
```

```
  wprintf (L"\"%ls\"  is tokenized:\n", ws);
  pwc = wcstok (ws, L" ,.-", &ptr);
  while (pwc != NULL)
  {
    wprintf (L"%ls\n", pwc);
    pwc = wcstok (NULL, L" ,.-", &ptr);
  }
}
```

Example Output

```
"- This, as they say, is the end."  is tokenized:
This
as
they
say
is
the
end
```

5.23.52 wcstol Function

Convert wide string to a long integer value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
```

Arguments

nptr	the wide string to attempt to convert
endptr	pointer to store the address of the remainder of the wide string that was not converted
base	the radix in which the value of the wide string should be interpreted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, `LONG_MIN` or `LONG_MAX` is returned based on the sign of the correct value.

Remarks

The `wcstol` function attempts to convert a portion of the wide string pointed to by `nptr` to a `long int`value.

The initial sequence consists of any white-space wide characters.

The subject sequence represents the integer constant to convert and whose radix is given by `base`.

If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant whose radix is determined by the sequence itself. For example, a leading `0x` implies a hexadecimal constant; a leading `0` implies an octal constant. The sequence may be preceded by a plus or minus sign, but not including an integer suffix.

If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters (`a` (or `A`) through `z` (or `Z`) ascribed the values 10 through 35) and digits representing an integer with the radix specified by `base`. Only letters and digits whose ascribed values are less than that of `base` are permitted. The sequence may be preceded by a plus or minus sign, but not including an integer suffix. If the value of `base` is 16, the wide characters `0x` or `0X` may prefix the sequence of letters and digits, following the sign wide character if present.

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial sequence and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The final sequence consists of wide characters in the wide string that were unrecognized in the subject sequence and which will include the terminating null wide character of the wide string. This sequence of the wide string will be accessible via the address stored to `endptr`, provided `endptr` is not a null pointer.

The value of the macro `ERANGE` is stored in `errno` if the correct value of the conversion is outside the range of representable values.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[] = L"2001 60c0c0 -110111010011010010000 7000 07000";
  wchar_t * pEnd;
  long int li1, li2, li3, li4, li5;

  li1 = wcstol(ws, &pEnd, 10);
  li2 = wcstol(pEnd, &pEnd, 16);
  li3 = wcstol(pEnd, &pEnd, 2);
  li4 = wcstol(pEnd, &pEnd, 16);
  li5 = wcstol(pEnd, NULL, 0);
  wprintf(L"The converted string values in decimal are: %ld, %ld, %ld, %ld, and %ld.\n", li1,
li2, li3, li4, li5);
}
```

Example Output

```
The converted string values in decimal are: 2001, 6340800, -3624224, 28672, and 3584.
```

5.23.53 wcstoll Function

Convert wide string to a long long integer value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

`long long int wcstoll(const wchar_t * restrict` **`nptr`**`, wchar_t ** restrict` **`endptr`**`, int` **`base`**`);`

Arguments

`nptr`	the wide string to attempt to convert
`endptr`	pointer to store the address of the remainder of the wide string that was not converted
`base`	the radix in which the value of the wide string should be interpreted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, `LLONG_MIN` or `LLONG_MAX` is returned based on the sign of the correct value.

Remarks

The wcstoll function attempts to convert a portion of the wide string pointed to by nptr to a long long intvalue.

The initial sequence consists of any white-space wide characters.

The subject sequence represents the integer constant to convert and whose radix is given by base.

If the value of base is zero, the expected form of the subject sequence is that of an integer constant whose radix is determined by the sequence itself. For example, a leading 0x implies a hexadecimal constant; a leading 0 implies an octal constant. The sequence may be preceded by a plus or minus sign, but not including an integer suffix.

If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters (a (or A) through z (or Z) ascribed the values 10 through 35) and digits representing an integer with the radix specified by base. Only letters and digits whose ascribed values are less than that of base are permitted. The sequence may be preceded by a plus or minus sign, but not including an integer suffix. If the value of base is 16, the wide characters 0x or 0X may prefix the sequence of letters and digits, following the sign wide character if present.

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial sequence and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

The final sequence consists of wide characters in the wide string that were unrecognized in the subject sequence and which will include the terminating null wide character of the wide string. This sequence of the wide string will be accessible via the address stored to endptr, provided endptr is not a null pointer.

The value of the macro ERANGE is stored in errno if the correct value of the conversion is outside the range of representable values.

Example

```
#include <wchar.h>

int main ()
{
  wchar_t ws[] = L"2001 60c0c0 -110111010011010010000 7000 07000";
  wchar_t * pEnd;
  long long int lli1, lli2, lli3, lli4, lli5;

  lli1 = wcstoll(ws, &pEnd, 10);
  lli2 = wcstoll(pEnd, &pEnd, 16);
  lli3 = wcstoll(pEnd, &pEnd, 2);
  lli4 = wcstoll(pEnd, &pEnd, 16);
  lli5 = wcstoll(pEnd, NULL, 0);
  wprintf(L"The converted string values in decimal are: %lld, %lld, %lld, %lld, and %lld.\n",
lli1, lli2, lli3, lli4, lli5);
}
```

Example Output

```
The converted string values in decimal are: 2001, 6340800, -3624224, 28672, and 3584.
```

5.23.54 wcstoul Function

Convert wide string to an unsigned long integer value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

<wchar.h>

Prototype

```
unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
int base);
```

Arguments

nptr	the wide string to attempt to convert
endptr	pointer to store the address of the remainder of the wide string that was not converted
base	the radix in which the value of the wide string should be interpreted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, `ULONG_MAX` is returned.

Remarks

The `wcstoul` function attempts to convert a portion of the wide string pointed to by `nptr` to a `unsigned long int`value.

The initial sequence consists of any white-space wide characters.

The subject sequence represents the integer constant to convert and whose radix is given by `base`.

If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant whose radix is determined by the sequence itself. For example, a leading `0x` implies a hexadecimal constant; a leading `0` implies an octal constant. The sequence may be preceded by a plus or minus sign, but not including an integer suffix.

If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters (`a` (or `A`) through `z` (or `Z`) ascribed the values 10 through 35) and digits representing an integer with the radix specified by `base`. Only letters and digits whose ascribed values are less than that of `base` are permitted. The sequence may be preceded by a plus or minus sign, but not including an integer suffix. If the value of `base` is 16, the wide characters `0x` or `0X` may prefix the sequence of letters and digits, following the sign wide character if present.

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial sequence and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The final sequence consists of wide characters in the wide string that were unrecognized in the subject sequence and which will include the terminating null wide character of the wide string. This sequence of the wide string will be accessible via the address stored to `endptr`, provided `endptr` is not a null pointer.

The value of the macro `ERANGE` is stored in `errno` if the correct value of the conversion is outside the range of representable values.

Example

```
#include <wchar.h>

int main ()
{
  wchar_t ws[] = L"2001 60c0c0 -110111010011010010000 7000 07000";
  wchar_t * pEnd;
  unsigned long int uli1, uli2, uli3, uli4, uli5;

  uli1 = wcstoul(ws, &pEnd, 10);
  uli2 = wcstoul(pEnd, &pEnd, 16);
  uli3 = wcstoul(pEnd, &pEnd, 2);
  uli4 = wcstoul(pEnd, &pEnd, 16);
  uli5 = wcstoul(pEnd, NULL, 0);
  wprintf(L"The converted string values in decimal are: %lu, %lu, %lu, %lu, and %lu.\n",
uli1, uli2, uli3, uli4, uli5);
}
```

Example Output

```
The converted string values in decimal are: 2001, 6340800, 18446744073705927392, 28672, and
3584.
```

5.23.55 wcstoull Function

Convert wide string to an unsigned long long integer value.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
unsoigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict
endptr, int base);
```

Arguments

nptr	the wide string to attempt to convert
endptr	pointer to store the address of the remainder of the wide string that was not converted
base	the radix in which the value of the wide string should be interpreted

Return Value

The converted value, or 0 if the conversion could not be performed. If the correct value of the conversion is outside the range of representable values, `ULLONG_MAX` is returned.

Remarks

The `wcstoull` function attempts to convert a portion of the wide string pointed to by `nptr` to a `unsigned long long int`value.

The initial sequence consists of any white-space wide characters.

The subject sequence represents the integer constant to convert and whose radix is given by `base`.

If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant whose radix is determined by the sequence itself. For example, a leading `0x` implies a hexadecimal constant; a leading `0` implies an octal constant. The sequence may be preceded by a plus or minus sign, but not including an integer suffix.

If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters (`a` (or `A`) through `z` (or `Z`) ascribed the values 10 through 35) and digits representing an integer with the radix specified by `base`. Only letters and digits whose ascribed values are less than that of `base` are permitted. The sequence may be preceded by a plus or minus sign, but not including an integer suffix. If the value of `base` is 16, the wide characters `0x` or `0X` may prefix the sequence of letters and digits, following the sign wide character if present.

Conversion stops once an unrecognized wide character is encountered, thus the subject sequence is defined as the longest subsequence of the input wide string after the initial sequence and that is of the expected form. A pointer to the position of the first unrecognizable wide character in the wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The final sequence consists of wide characters in the wide string that were unrecognized in the subject sequence and which will include the terminating null wide character of the wide string. This sequence of the wide string will be accessible via the address stored to `endptr`, provided `endptr` is not a null pointer.

The value of the macro `ERANGE` is stored in `errno` if the correct value of the conversion is outside the range of representable values.

Example

```
#include <wchar.h>

int main ()
{
  wchar_t ws[] = L"2001 60c0c0 -110111010011010010000 7000 07000";
  wchar_t * pEnd;
  unsigned long long int ulli1, ulli2, ulli3, ulli4, ulli5;

  ulli1 = wcstoull(ws, &pEnd, 10);
  ulli2 = wcstoull(pEnd, &pEnd, 16);
  ulli3 = wcstoull(pEnd, &pEnd, 2);
  ulli4 = wcstoull(pEnd, &pEnd, 16);
  ulli5 = wcstoull(pEnd, NULL, 0);
  wprintf(L"The converted string values in decimal are: %llu, %llu, %llu, %llu, and %llu.\n",
ulli1, ulli2, ulli3, ulli4, ulli5);
}
```

Example Output

```
The converted string values in decimal are: 2001, 6340800, 18446744073705927392, 28672, and
3584.
```

5.23.56 wcsxfrm Function

Transforms a wide string based on locale.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
size_t wcsxfrm( wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Arguments

s1	the array to hold the transformed wide string
s2	the wide string to transform
n	the maximum number of wide characters to transform

Return Value

The function returns the length of the transformed wide string (not including the terminating null wide character). If the value returned is `n` or greater, the contents of the array pointed to by `s1` are indeterminate.

Remarks

The function transforms the wide string pointed by `s2` according to the current locale and places no more than `n` wide characters of the result and a terminating null wide character into the array pointed to by `s1`. A null pointer may be specified for `s1` if `n` is zero, in which case the function can be used to obtain the size of the transformed wide string.

Example

```
#include <wchar.h>

int main(void)
{
   wchar_t * wcs;
   wchar_t  buffer[40];
   int      length;
```

```
    printf("Enter a wide string\n");
    wcs = fgetws(buffer, 40, stdin);
    length = wcsxfrm(NULL, wcs, 0);
    printf("string length once transformed to current locale: %d\n", length);
}
```

Example Output

The output of this program might vary based on the `LC_COLLATE` category of the current locale.

5.23.57 wctob Function

Converts a wide character to an equivalent single-byte character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int wctob(wint_t c);
```

Arguments

c	the wide character to convert

Return Value

The function returns the character `c` as single-byte character (converted to type `int`) provided that `c` is a valid single-byte character in the initial shift state of a multibyte sequence. Otherwise, the function returns `EOF`.

Remarks

The function determines whether `c` is a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

A multibyte character set may have a state-dependent encoding. Sequences begins in an initial shift state and enter other locale-specific shift states when specific multibyte characters are encountered. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  int i, num = 0;
  const char mbs[] = "The quick brown fox\n";

  for(i=0; i!=sizeof(mbs); i++)
    if(wctob(mbs[i]) != EOF)
      ++num;

  wprintf(L"mbs contains %d wide characters that translate to single-byte characters.\n",
num);
}
```

Example Output

```
mbs contains 21 wide characters that translate to single-byte characters.
```

5.23.58 wmemchr Function

Locates the first occurrence of a wide character in part of a wide string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t *wcschr(const wchar_t *s, wchar_t c, size_t n);
```

Arguments

s	the wide string in which to search
c	the wide character to search for
n	the number of wide character to search

Return Value

The function returns a pointer to the wide character, or a null pointer if the wide character was not found.

Remarks

The function locates the first occurrence of the wide character `c` in the first **n** wide characters in the wide string pointed to by `s`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[50] = L"What time is it?";
  wchar_t ch1 = L'i', ch2 = L'y';
  wchar_t * ptr;
  int res;
  wprintf(L"ws : %ls\n\n", ws);

  ptr = wmemchr(ws, ch1, 50);
  if (ptr != NULL)
  {
   res = ptr - ws + 1;
   wprintf(L"%lc found at position %d\n", ch1, res);
  }
  else
    wprintf(L"%lc not found\n", ch1);
  wprintf(L"\n");

  ptr = wmemchr(ws, ch2, 50);
  if (ptr != NULL)
  {
    res = ptr - ws + 1;
    wprintf(L"%lc found at position %d\n", ch2, res);
  }
  else
    wprintf(L"%lc not found\n", ch2);
}
```

Example Output

```
ws : What time is it?

i found at position 7

y not found
```

5.23.59 wmemcmp Function

Compare wide characters in objects.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int wmemcmp(const wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Arguments

s1	one object to compare
s2	the other object to compare
n	the maximum number of wide characters to compare

Return Value

Returns a positive integer if `s1` is greater than `s2`, zero if `s1` is equal to `s2` or a negative number if `s1` is less than `s2`.

Remarks

The function returns a value based on the first wide character that differs between `s1` and `s2` after at most `n` comparisons. It differs to `wcsncmp()` in that the comparison continues past terminating null wide characters until all `n` wide characters have been compared.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t buf1[50] = L"Where is the time?";
  wchar_t buf2[50] = L"Where did they go?";
  wchar_t buf3[50] = L"Why?";
  int res;

  wprintf(L"buf1 : %ls\n", buf1);
  wprintf(L"buf2 : %ls\n", buf2);
  wprintf(L"buf3 : %ls\n\n", buf3);

  res = wmemcmp(buf1, buf2, 6);
  if (res < 0)
    wprintf(L"buf1 comes before buf2\n");
  else if (res == 0)
    wprintf(L"6 characters of buf1 and buf2 are equal\n");
  else
   wprintf(L"buf2 comes before buf1\n");
  wprintf(L"\n");

  res = wmemcmp(buf1, buf2, 20);
  if (res < 0)
    wprintf(L"buf1 comes before buf2\n");
  else if (res == 0)
```

```
    wprintf(L"20 characters of buf1 and buf2 are equal\n");
  else
    wprintf(L"buf2 comes before buf1\n");
  wprintf(L"\n");

  res = wmemcmp(buf1, buf3, 20);
  if (res < 0)
    wprintf(L"buf1 comes before buf3\n");
  else if (res == 0)
    wprintf(L"20 characters of buf1 and buf3 are equal\n");
  else
    wprintf(L"buf3 comes before buf1\n");
}
```

Example Output

```
buf1 : Where is the time?
buf2 : Where did they go?
buf3 : Why?

6 characters of buf1 and buf2 are equal

buf2 comes before buf1

buf1 comes before buf3
```

5.23.60 wmemcpy Function

Copy wide characters to an object.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
wchar_t * wmemcpy( wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Arguments

s1	the object to hold the copied wide characters
s2	the object from which wide characters are copied
n	the number of wide characters to copy

Return Value

A copy of `s1`.

Remarks

The function copies `n` wide characters from the object pointed to by `s2` into the object pointed to by `s1`.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[40];

  wmemcpy(ws, L"a literal wide string", 10);
  ws[10] = L'\0';
  wprintf(L"the wide string copied to ws is \"%ls\"\n", ws);
}
```

Example Output

```
the wide string copied to ws is "a literal "
```

5.23.61 wmemmove Function

Copy wide characters to an object, handling overlap of source and destination objects.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

<wchar.h>

Prototype

```
wchar_t * wmemmove( wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Arguments

s1	the object to hold the copied wide characters
s2	the object from which wide characters are copied
n	the number of wide characters to copy

Return Value

A copy of s1.

Remarks

The function copies n wide characters from the object pointed to by s2 into the object pointed to by s1. The memory used by the n wide characters in the destination location may overlap with that of the n wide characters in the source object.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws[40] = L"A string   to be sure";

  wmemmove(&ws[9], &ws[11], 11);
  wprintf(L"the wide string is \"%ls\"\n", ws);
}
```

Example Output

```
the wide string is "A string to be sure"
```

5.23.62 wmemset Function

Sets locations in an object to a specified value

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

<wchar.h>

Prototype

```
wchar_t * wmemset(wchar_t * s, wchar_t c, size_t n);
```

Arguments

s	the object to modify
c	the value to set
n	the number of locations to set

Return Value

A copy of `s`.

Remarks

The function sets the first `n` wide characters of the object pointed to by `s` to be the value `c`.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t ws1[20] = L"What time is it?";
  wchar_t ws2[20] = L"";
  wchar_t ch1 = L'?', ch2 = L'y';
  wchar_t * ptr;
  int res;

  printf("wmemset(\"%ls\", \'%lc\',4);\n", ws1, ch1);
  wmemset(ws1, ch1, 4);
  printf("ws1 after wmemset: %ls\n", ws1);

  printf("\n");
  printf("wmemset(\"%ls\", \'%lc\',10);\n", ws2, ch2);
  wmemset(ws2, ch2, 10);
  printf("ws2 after wmemset: %ls\n", ws2);
}
```

Example Output

```
wmemset("What time is it?", '?',4);
ws1 after wmemset: ???? time is it?

wmemset("", 'y',10);
ws2 after wmemset: yyyyyyyyyy
```

5.23.63 wprintf Function

Prints formatted text to the `stdout` stream, using a wide format string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

```
<wchar.h>
```

Prototype

```
int wprintf(const wchar_t * restrict format, ...);
```

Arguments

format	format control wide string

`...` optional arguments; see “Remarks”

Return Value

Returns number of characters generated or a negative number if an error occurs.

Remarks

The function writes formatted output to the `stdout` stream. The `arg` argument must be initialized by the `va_start` macro.

The `format` string is composed of text, which is copied unchanged to the output stream, and conversion specifications, which begin with the `%` wide character and which fetch zero or more of the optional arguments, converting them according to the corresponding conversion specifier when required, and then writing the result to the output stream. If the are less arguments than required by the conversion specifications, the output is undefined. If there are more arguments than required by the conversion specifications, the additional arguments are unused.

Each conversion specification begins with a percent sign followed by optional fields and a required type as shown here:

`%[`*`flags`*`][`*`width`*`][.`*`precision`*`][`*`length`*`]`*`specifier`*

The *`flags`* modify the meaning of the conversion specification. The are described in the following table.

Flag	Meaning
`-`	Left justify the conversion result within a given field width.
`0`	Use 0 for the pad character instead of space (which is the default) for `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, except when converting an infinity or NaN. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and`X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.
`+`	Write a plus sign for positive signed conversion results.
space	Prefix a space when the first wide character of a conversion result does not include a sign (`+` or `-`) or if a signed conversion results in no wide characters. If the *space* and `+` flags both appear, the *space* flag is ignored.
`#`	Convert the result to an alternative form. Specifically, for `o` conversions, it increases the precision, ensuring the first digit of the result is a zero, if and only if necessary. (If the value and precision are both 0, a single 0 is printed). For `x` (or `X`) conversions, `0x` (or `0X`) is prefixed to nonzero results. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. For `g` and `G` conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

The *`width`* field indicated how many characters the value should consume. If the converted value has fewer characters than the field width, it is padded with spaces on the left, unless the left adjustment flag `-` has been used.

If the asterisk, `*`, is used instead of a decimal number, the `int` argument before the argument to be converted will be used for the field width. If the result is less than the field width, pad characters will be used on the left to fill the field. If the result is greater than the field width, the field is expanded to accommodate the value without padding.

The *`precision`* field indicates:

- the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions
- the number of digits to appear after the decimal-point character for `a`, `A`, `e`, `E`, `f`, and `F` conversions
- the maximum number of significant digits for the `g` and `G` conversions
- the maximum number of bytes to be written for `s` conversions

The precision takes the form of a period, `.`, followed either by an asterisk, `*` or by an optional decimal integer. If neither the `*` or integer is specified, the precision is assumed to be zero. If the asterisk, `*`, is used instead of a

decimal number, the `int` argument before the argument to be converted will be used for the precision. If a precision appears with any other conversion specifier, the behavior is undefined.

The *`length`* modifier specifies the size of the argument as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
`h`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, converts the argument value to a `short int` or `unsigned short int`.
`h`	When used with the `n` conversion specifier, indicates that the pointer argument points to a `short int`.
`hh`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, converts the argument value to a `signed char` or `unsigned char`.
`j`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, indicates that the argument value is a`intmax_t` or `uintmax_t`.
`j`	When used with the `n` conversion specifier, indicates that the pointer points to a `intmax_t`.
`l`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, converts the argument value to a `long int` or `unsigned long int`.
`l`	When used with the `n` conversion specifier, indicates that the pointer points to a `long int`.
`l`	When used with the `c` conversion specifier, indicates that the argument value is a wide character (`wint_t` type).
`l`	When used with the `s` conversion specifier, indicates that the argument value is a wide string (`wchar_t` type).
`l`	When used the `e`, `E`, `f`, `F`, `g`, `G`, has no effect.
`ll`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, indicates that the argument value is a `long long int` or `unsigned long long int`.
`ll`	When used with the `n` conversion specifier, indicates that the pointer points to a `long long int`.
`ll`	MPLAB XC16: When used with the `s` conversion specifier, indicates that the string pointer is an `__eds__` pointer. Other compilers: The modifier is silently ignored.
`L`	When used the `a`, `A`, `e`, `E`, `f`, `F`, `g`, `G` conversion specifier, indicates the argument value is a `long double`.
`t`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, indicates that the argument value is a `ptrdiff_t` or the corresponding unsigned integer type.
`t`	When used with the `n` conversion specifier, indicates that the pointer points to a `ptrdiff_t`.
`z`	When used with `d`, `i`, `o`, `u`, `x`, or `X` conversion specifiers, indicates that the argument value is a `size_t_t` or the corresponding signed integer type.
`z`	When used with the `n` conversion specifier, indicates that the pointer points to a `size_t_t`.

The conversion *`specifiers`* specify the type of conversion to be applied to the argument and how that value should be printed, as described in the following table.

Specifier	Argument value type	Printing notes
a, A	double	Converted to the general form [-]0x*h.hhhhp±d*, where there is one hexadecimal digit before the decimal-point wide character and the number of hexadecimal digits after it is equal to the precision.
c	char or wint_t	The integer argument value is converted to a wide character (as if by calling btowc) and the resulting wide character is written. When the l modifier is present, the wint_t argument is converted to wchar_t and written.
d	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
e, E	double	Converted to the general form [-]*d.ddde±dd*, where there is one digit before the decimal-point wide character and the number of digits after it is equal to the precision, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
f, F	double	Converted to decimal notation using the general form [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification, which is 6 if that is missing. If the precision is zero and the # flag is not specified, no decimal-point wide character appears.
g, G	double	takes the form of e, f, or E or F in the case of G, as appropriate
i	signed int	Converted to signed decimal with the general form [-]*dddd*. The precision specifies the minimum number of digits to appear.
n	an integer pointer	The number of wide characters written so far is written to the object pointed to by the pointer. No wide characters are printed.
o	unsigned int	Converted to unsigned octal with the general form *dddd*. The precision specifies the minimum number of digits to appear
p	void *	Printed as the value (address) held by the pointer.
s	char array, or wchar_t array	A string contain multibyte characters that are converted to wide characters (as if by calling mbrtowc) and written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written. When the l modifier is present, wide characters from the array are written. No more wide characters than the specified precision are printed. If no precision has been specified or is greater than the number of converted characters, a null wide character will be written.
u	unsigned int	Converted to unsigned decimal with the general form *dddd*. The precision specifies the minimum number of digits to appear
x	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters abcdef are printed for digits above 9. The precision specifies the minimum number of digits to appear
X	unsigned int	Converted to unsigned hexadecimal notation with the general form *dddd*. The letters ABCDEF are printed for digits above 9. The precision specifies the minimum number of digits to appear
%		No argument is converted and a % wide character is printed.

Example

```
#include <wchar.h>

int main(void)
{
  int y;
  wchar_t s[]=L"Print this string";
  int x = 1;
  wchar_t a = L'\n';

  y = wprintf(L"%ls %d time%lc", s, x, a);

  printf("Number of characters printed to wide string buffer = %d\n", y);
}
```

Example Output

```
Print this string 1 time
Number of characters printed to wide string buffer = 25
```

5.23.64 wscanf Function

Scans formatted text from the `stdin` stream, using a wide format string.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wchar.h>`

Prototype

```
int wscanf(const wchar_t * restrict format, ...);
```

Arguments

`format`	format control wide string
`...`	optional arguments; see “Remarks”

Return Value

Returns the number of items successfully converted and assigned. If no items are assigned, a 0 is returned. `EOF` is returned if end-of-file is encountered before the first conversion or if an error occurs.

Remarks

This function can read input from the `stdin` stream.

The `format` string can be composed of white space wide characters, which reads input up to the first non-white-space wide character in the input; or ordinary wide characters, which if not read in that order, will trigger an error and leave the remaining characters in the input, unread. The format string can also contain conversion specifications, which begin with the `%` wide character and that specify the expected input sequences and how they are to be converted for assignment to objects pointed to by the pointer arguments.

Each conversion specification begins with a percent wide character followed by optional fields and a required type as shown here:

`%[*][width][length]specifier`

The presence of `*` immediately after the percent wide character indicates that the read and converted value should not be assigned to any object.

The *width* is a decimal integer that must be non-zero and indicates the maximum field width (in characters) of the input that is read to match the specification.

The *length* modifier specifies the size of the object in which the value will be assigned, as described in the following table. Their use with other conversion specifiers results in undefined behavior.

Modifier	Meaning
h	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a short int or unsigned short int object referenced by the pointer argument.
hh	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a signed char or unsigned char object referenced by the pointer argument.
l	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long int or unsigned long int object referenced by the pointer argument.
l	When used the a, A, e, E, f, F, g, or G indicates that this conversion specifier assigns to a double object referenced by the pointer argument.
l	When used the c, s, or [indicates that this conversion specifier assigns to a wchar_t object referenced by the pointer argument.
ll	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a long long int or unsigned long long int object referenced by the pointer argument.
L	When used the a, A, e, E, f, F, g, G conversion specifier, indicates the argument value is a long double.
j	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a intmax_t or uintmax_t object referenced by the pointer argument.
t	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a ptrdiff_t object referenced by the pointer argument.
z	When used with d, i, o, u, x, X, or n indicates that this conversion specifier assigns to a size_t object referenced by the pointer argument.

The conversion *specifiers* specify the type of conversion to be applied to the read sequence.

Specifier	Receiving object type	Matches
a	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
c	char array, or wchar_t array	Matches a single wide character or the number of characters specified by the field width if present. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array of sufficient size. No null character is added. If an l length modifier is present, the input shall be a wide character array of sufficient size. No null wide character is added.
d	signed int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtod function for the first argument when using a value of 10 for the base argument.
e	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.

.....continued

Specifier	Receiving object type	Matches
f	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
g	double	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected by the strtod function for the first argument.
i	signed int	Converted to signed decimal with the general form [−]*dddd*. The precision specifies the minimum number of digits to appear.
n	int	No input is consumed but the number of wide characters this call has so far read from the input stream is written to the argument with no increment of the assignment count returned by the function. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
o	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
p	void * *	Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fwprintf function. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.
s	char array, or wchar_t array	Matches a sequences of non-white-space wide characters, which is written to the array argument. If no l length modifier is present, characters from the input field are converted to a multibyte character sequence (as if by calling wcrtomb). The corresponding argument shall be a pointer to the initial element of a character array large of sufficient size. A terminating null character is appended. If an l length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide character array of sufficient size. A terminating null wide character is appended.
x	unsigned int	Matches an optionally signed decimal integer, whose format is the same as expected by the strtoul function for the first argument when using a value of 8 for the base argument.
%		Matches a single % wide character. No assignment takes place.

.....continued

Specifier	Receiving object type	Matches
[	`char` array, or `wchar_t` array	Matches all the wide characters in the input that have been specified between the `[` and trailing `]` wide character, unless the wide character after the opening bracket is a circumflex, `^`, in which case a match is only made with wide characters that do not appear in the brackets. If the conversion specifier begins with `[]` or `[^]`, the right bracket wide character will match the input and the next following right bracket wide character is the matching right bracket that ends the specification. If no `l` length modifier is present, characters from the input field are converted to amultibyte character sequence (as if by calling `wcrtomb`). The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. If an `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of a wide array of sufficient size.A terminating null wide character will be appended.

Example

```
#include <wchar.h>

int main(void)
{
  wchar_t s[30];
  int x;
  wchar_t a;

  wscanf(L"%ls", s);
  wprintf(L"%ls\n", s);
  wscanf(L"%d", &x);
  wprintf(L"%d\n", x);
  wscanf(L"%ls", s);
  wprintf(L"%ls\n", s);
}
```

Example Input

Provided by the `stdin` stream.

```
Message 4 you
```

Example Output

```
Message
4
you
```

5.24 <wctype.h> Wide character classification and mapping utilities

The header file `wctype.h` consists of functions that are useful for classifying and mapping wide characters. Characters are interpreted according to the Standard C locale.

Attention: This header is implemented only by MPLAB XC32 C compilers.

5.24.1 wchar_t Type

An integer type capable of holding values that can represent distinct codes for all members of the largest extended character set specified among the supported locales.

Include

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

5.24.2 wctrans_t Type

A scalar type that can hold values which represent locale-specific character mappings.

Attention: This type is implemented only by MPLAB XC32 C compilers.

Include

```
<wctype.h>
```

5.24.3 wint_t Type

An integer type used to hold any value corresponding to members of the extended character set, as well as the wide end-of-file marker, WEOF.

Attention: This type is implemented only by MPLAB XC32 C compilers.

Include

```
<wctype.h>
<wchar.h>
```

Definition

```
typedef unsigned wint_t;
```

5.24.4 WEOF Macro

A constant expression of type `wint_t` whose value does not correspond to any member of the extended character set and which indicates end-of-file, that is, no more input from a stream. It is also used as a wide character value that does not correspond to any member of the extended character set.

Attention: This type is implemented only by MPLAB XC32 C compilers.

Include

```
<wctype.h>
<wchar.h>
```

5.24.5 iswalnum Function

Test for a alphanumeric wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswalnum(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is alphanumeric; otherwise, returns a zero.

Remarks

Alphanumeric wide characters are included within the ranges `L'A'-L'Z'`, `L'a'-L'z'` or `L'0'-L'9'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
    if(iswalnum(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 0, 2, 3, 4, 5, 6, 7, 10, 12, 13, 14, 15, 17, 19, 20,
21, 22
```

5.24.6 iswalpha Function

Test for a alphabetic wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

<wctype.h>

Prototype

```
int iswalpha(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is alphabetic; otherwise, returns a zero.

Remarks

Alphanumeric wide characters are included within the ranges `L'A'-L'Z'` or `L'a'-L'z'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
    if(iswalpha(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 0, 2, 3, 4, 5, 6, 7, 12, 13, 14, 15, 19, 20, 21, 22
```

5.24.7 iswblank Function

Test for a space or tab wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

<wctype.h>

Prototype

```
int iswblank(wint_t wc);
```

Argument

wc The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is a space or tab wide character; otherwise, returns a zero.

Remarks

A wide character is considered to be a white-space wide character if it is one of the following: space (`L' '`) or horizontal tab (`L'\t'`).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
    if(iswblank(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 1, 9, 11, 16, 18
```

5.24.8 iswcntrl Function

Test for a control wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswcntrl(wint_t wc);
```

Argument

wc The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is a control wide character; otherwise, returns a zero.

Remarks

A control wide character does not occupy any printing position on a display when printed.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
    if(iswcntrl(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 24
```

5.24.9 iswctype Function

Test the property of a wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswctype(wint_t wc, wctype_t desc);
```

Arguments

wc	The wide character to test.
desc	The wide character property to test for.

Return Value

Returns a non-zero integer value if the wide character, `wc`, has the property described by `desc`; otherwise, returns a zero.

Remarks

See 5.24.22 wctype Function, which can be used to obtain a description of the desired wide character property.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
            if(iswctype(wc, wctype("space"))) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 1, 9, 11, 16, 18, 24
```

5.24.10 iswdigit Function

Test for a decimal digit wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

`int iswdigit(wint_t wc);`

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is a decimal digit wide character; otherwise, returns a zero.

Remarks

A wide character is considered to be a decimal digit wide character if it is in the range of `L'0'-L'9'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>
```

```
int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
    if(iswdigit(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 10, 17
```

5.24.11 iswgraph Function

Test for a graphical wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswgraph(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is a graphical wide character; otherwise, returns a zero.

Remarks

A wide character is considered to be a graphical wide character if it is any printable character but not a white space wide character.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
```

```
    wc = wStr[idx];
    if(iswgraph(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 0, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 14, 15, 17, 19, 20,
21, 22, 23
```

5.24.12 iswlower Function

Test for a lowercase alphabetic wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswlower(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is a lowercase alphabetic wide character; otherwise, returns a zero.

Remarks

A wide character is considered to be a lowercase wide character if it is in the range of `L'a'-L'z'`

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
    if(iswlower(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
```

```
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 2, 3, 4, 5, 6, 7, 12, 13, 14, 15, 19, 20, 21, 22
```

5.24.13 iswprint Function

Test for a printable wide character (includes a wide space).

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswprint(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is printable; otherwise, returns a zero.

Remarks

A printable wide character occupies at least one printing position on a display when printed.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
    if(iswprint(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23
```

5.24.14 iswpunct Function

Test for a punctuation wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswpunct(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is a punctuation wide character; otherwise, returns a zero.

Remarks

A wide character is considered to be a punctuation wide character if it is a printable wide character which is neither a space nor an alphanumeric wide character. Punctuation wide characters consist of the following:

```
L'!'  L'"'  L'#'  L'$'  L'%'  L'&'  L'''  L'('  L')'  L';'  L'<'  L'='  L'>'
L'?'  L'@'  L'['  L'\'  L']'  L'*'  L'+'  L','  L'-'  L'.'  L'/'  L':'  L'^'
L'_'  L'{'  L'|'  L'}'  L'~'
```

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  wint_t wStr[] = L"A string, 2 wide 4 some.\n";
  wint_t wc;
  unsigned idx = 0;
  bool found = false;

  do {
    wc = wStr[idx];
    if(iswpunct(wc)) {
      if( ! found)
        printf("Found matching wide char at position: ");
      printf("%s%u", found ? ", " : "", idx);
      found = true;
    }
    idx++;
  } while(wStr[idx]);
  printf("\n");
}
```

Example Output

```
Found matching wide char at position: 8, 23
```

5.24.15 iswspace Function

Test for a white-space wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswspace(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is white-space wide character; otherwise, returns a zero.

Remarks

A wide character is considered to be a white-space wide character if it is one of the following: space (`L' '`), form feed (`L'\f'`), newline (`L'\n'`), carriage return (`L'\r'`), horizontal tab (`L'\t'`), or vertical tab (`L'\v'`).

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
    wint_t wStr[] = L"A string, 2 wide 4 some.\n";
    wint_t wc;
    unsigned idx = 0;
    bool found = false;

    do {
        wc = wStr[idx];
        if(iswspace(wc)) {
            if( ! found)
                printf("Found matching wide char at position: ");
            printf("%s%u", found ? ", " : "", idx);
            found = true;
        }
        idx++;
    } while(wStr[idx]);
    printf("\n");
}
```

Example Output

```
Found matching wide char at position: 1, 9, 11, 16, 18, 24
```

5.24.16 iswupper Function

Test for a uppercase alphabetic wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswupper(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is a uppercase alphabetic wide character; otherwise, returns a zero.

Remarks

A wide character is considered to be a lowercase wide character if it is in the range of `L'A'-L'Z'`

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
    wint_t wStr[] = L"A string, 2 wide 4 some.\n";
    wint_t wc;
    unsigned idx = 0;
    bool found = false;

    do {
        wc = wStr[idx];
        if(iswupper(wc)) {
            if( ! found)
                printf("Found matching wide char at position: ");
            printf("%s%u", found ? ", " : "", idx);
            found = true;
        }
        idx++;
    } while(wStr[idx]);
    printf("\n");
}
```

Example Output

```
Found matching wide char at position: 0
```

5.24.17 iswxdigit Function

Test for a hexadecimal digit wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int iswxdigit(wint_t wc);
```

Argument

wc	The wide character to test.

Return Value

Returns a non-zero integer value if the wide character, `wc`, is a hexadecimal digit wide character; otherwise, returns a zero.

Remarks

A wide character is considered to be a hexadecimal digit wide character if it is in the range of `L'0'-L'9'`, `L'a'-L'f'`, or `L'A'-L'F'`.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
    wint_t wStr[] = L"A string, 2 wide 4 some.\n";
    wint_t wc;
    unsigned idx = 0;
    bool found = false;

    do {
        wc = wStr[idx];
        if(iswxdigit(wc)) {
            if( ! found)
                printf("Found matching wide char at position: ");
            printf("%s%u", found ? ", " : "", idx);
            found = true;
        }
        idx++;
    } while(wStr[idx]);
    printf("\n");
}
```

Example Output

```
Found matching wide char at position: 0, 10, 14, 15, 17, 22
```

5.24.18 towctrans Function

Map a wide character with a property.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
wint_t towctrans(wint_t wc, wctrans_t desc);
```

Arguments

wc	The wide character to map.
desc	The wide character property to map with.

Return Value

Returns a wide integer value being the wide character, `wc`, mapped with the property described by `desc`.

Remarks

See 5.24.21 wctrans Function, which can be used to obtain a description of the desired wide character mapping.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
        wint_t wStr[] = L"A string, 2 wide 4 some.\n";
        wint_t wc;
        unsigned idx = 0;

        printf("String in lower case: ");
        do {
                wc = wStr[idx];
                putwchar(towctrans(wc, wctrans("tolower")));
                idx++;
        } while(wStr[idx]);
        printf("\n");
}
```

Example Output

```
String in lower case: a string, 2 wide 4 some.
```

5.24.19 towlower Function

Convert a wide character to a lowercase alphabetical wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int towlower(wint_t wc);
```

Argument

wc	The wide character to convert.

Return Value

Returns the corresponding lowercase alphabetical wide character if the argument, `wc`, was originally uppercase; otherwise, returns the original wide character.

Remarks

Only uppercase alphabetical wide characters may be converted to lowercase.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
    wint_t wStr[] = L"A string, 2 wide 4 some.\n";
    wint_t wc;
    unsigned idx = 0;

    printf("String in lower case: ");
    do {
        wc = wStr[idx];
        putwchar(towlower(wc));
        idx++;
    } while(wStr[idx]);
    printf("\n");
}
```

Example Output

```
String in lower case: a string, 2 wide 4 some.
```

5.24.20 towupper Function

Convert a wide character to an uppercase alphabetical wide character.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

```
int towupper(wint_t wc);
```

Argument

wc	The wide character to convert.

Return Value

Returns the corresponding uppercase alphabetical wide character if the argument, `wc`, was originally lowercase; otherwise, returns the original wide character.

Remarks

Only lowercase alphabetical wide characters may be converted to uppercase.

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
    wint_t wStr[] = L"A string, 2 wide 4 some.\n";
    wint_t wc;
    unsigned idx = 0;

    printf("String in upper case: ");
    do {
        wc = wStr[idx];
        putwchar(towupper(wc));
        idx++;
    } while(wStr[idx]);
    printf("\n");
}
```

Example Output

```
String in upper case: A STRING, 2 WIDE 4 SOME.
```

5.24.21 wctrans Function

Returns a value that describes a mapping between wide characters and that can be used by `towctrans()`.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

`wctrans_t wctrans(const char * property);`

Argument

`property`	The wide character mapping property.

Return Value

Returns a value with type `wctrans_t` that describes a mapping of wide characters identified by the string argument, `property`

Remarks

The string arguments are shown in the following table along with the wide character function which maps with the same wide character property.

Table 5-8. Wide character properties

String argument to `wctrans()`	Function that maps with the same wide character property
`"tolower"`	`towlower()`
`"toupper"`	`towupper()`

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

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

int main(void)
{
        wint_t wStr[] = L"A string, 2 wide 4 some.\n";
        wint_t wc;
        unsigned idx = 0;

        printf("String in upper case: ");
        do {
                wc = wStr[idx];
                putwchar(towctrans(wc, wctrans("toupper")));
                idx++;
        } while(wStr[idx]);
        printf("\n");
}
```

Example Output

```
String in upper case: A STRING, 2 WIDE 4 SOME.
```

5.24.22 wctype Function

Returns a value that describes a class of wide characters and that can be used by `iswctype()`.

Attention: This function is implemented only by MPLAB XC32 C compilers.

Include

`<wctype.h>`

Prototype

`int wctype(const char * property);`

Argument

`property`	The wide character property to test for.

Return Value

Returns a value with type `wctype_t` that describes a class of wide characters identified by the string argument, `property`

Remarks

The string arguments are shown in the following table along with the wide character function which tests for the same wide character property.

Table 5-9. Wide character properties

String argument to `wctype()`	Function that tests for the same wide character property
`"alnum"`	`iswalnum()`
`"alpha"`	`iswalpha()`
`"blank"`	`iswblank()`

.....continued

String argument to `wctype()`	Function that tests for the same wide character property
"cntrl"	iswcntrl()
"digit"	iswdigit()
"graph"	iswgraph()
"lower"	iswlower()
"print"	iswprint()
"punct"	iswpunct()
"space"	iswspace()
"upper"	iswupper()
"xdigit"	isxdigit()

Example

See the notes at the beginning of this chapter or section for information on using `printf()` or `scanf()` (and other functions reading and writing the `stdin` or `stdout` streams) in the example code.

```
#include <wctype.h>
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
    wint_t wStr[] = L"A string, 2 wide 4 some.\n";
    wint_t wc;
    unsigned idx = 0;
    bool found = false;

    do {
        wc = wStr[idx];
             if(iswctype(wc, wctype("digit"))) {
            if( ! found)
                printf("Found matching wide char at position: ");
            printf("%s%u", found ? ", " : "", idx);
            found = true;
        }
        idx++;
    } while(wStr[idx]);
    printf("\n");
}
```

Example Output

```
Found matching wide char at position: 10, 17
```

6. Syscall Interface

Some of the MPLAB XC compilers use architecture-specific macros and functions to explicitly perform system calls that are needed by many other standard library functions. To reduce the resources used, these syscall functions are designed to have limited functionality.

With the exception of `sbrk()`, which is fully implemented, the syscall functions described in the sections that follow are provided as weak stub definitions. These stubs provide reasonable behavior where the functionality is unimplemented, generally returning an error or trivial result. These functions can be modified by user code as required and included into a project to customize the behavior of certain library functions.

Those stubs that always fail (e.g. `open`) can optionally use the `.gnu.warning` mechanism to cause linker warnings when linked. This is controlled by the `__DISABLE_WARNINGS__` macro used by `arch/mchp/mchp.h`.

6.1 _exit Function

Terminate program execution.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<unistd.h>`

Prototype

```
void _exit(int status);
```

Argument

`status`	exit status

Remarks

This function is implemented as a stub that must be completed to suite the application.

This is a helper function called by the `exit()` Standard C Library function.

Default Behavior

As distributed, this function flushes `stdout` and terminates. The parameter status is the same as that passed to the `exit()` standard C library function.

Source File

`_exit.c`

6.2 access Function

Determine accessibility of a file

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<unistd.h>`

Prototype

```
int access(const char * filename, int how);
```

Arguments

filename	name of the file to be accessed
how	type of access permitted

Return Value

The return value is 0 if the access is permitted, and -1 otherwise. For the value -1, `errno` is set to indicate the kind of error. Appropriate values may be `EACCES`, `ENOENT` or `EROFS`, among others.

Remarks

This function is implemented as a stub that must be completed to suite the application.

The `access` function checks to see whether the file named by `filename` can be accessed in the way specified by the `how` argument. The argument either can be the bitwise OR of the flags `R_OK`, `W_OK`, `X_OK`, or the existence test `F_OK`

This function uses the *real* user and group IDs of the calling process, rather than the *effective* IDs, to check for access permission. As a result, if you use the function from a `setuid` or `setgid` program, it gives information relative to the user who actually ran the program.

Source File

`access.c`

6.3 brk Function

Set the end of the process's data space.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

None

Prototype

```
int brk(void * endds);
```

Argument

endds	pointer to the end of the data segment

Return Value

Returns 0 if successful; otherwise, returns -1.

Remarks

This function is implemented as a stub that must be completed to suite the application.

This helper function is used by the Standard C Library function `malloc()`.

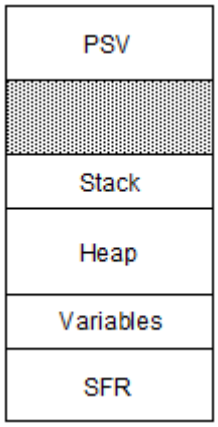
`brk()` is used to dynamically change the amount of space allocated for the calling process's data segment. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases.

Newly allocated space is uninitialized.

Default Behavior

A static variable is used to point to the first free heap location. If the argument `endds` is zero, the function sets the variable to the address of the start of the heap and returns zero. If the argument `endds` is non-zero and has a value less than `_eheap`, the function sets the variable to the value of `endds` and returns zero. Otherwise, the variable is unchanged and the function returns -1.

The argument `endds` must be within the heap range (see data space memory map below).



Since the stack is located immediately above the heap, using `brk()` or `sbrk()` has little effect on the size of the dynamic memory pool. The `brk()` and `sbrk()` functions are primarily intended for use in run-time environments where the stack grows downward and the heap grows upward.

The linker allocates a block of memory for the heap if the `-Wl,--heap=n` option is specified, where n is the desired heap size in characters. The starting and ending addresses of the heap are reported in variables: `_heap` and `_eheap`, respectively, and which are declared:

```
extern uint8_t _heap;              // heap start
extern uint8_t _eheap;             // heap end
```

For MPLAB XC16, using the linker's heap size option is the standard way of controlling heap size, rather than relying on `brk()` and `sbrk()`.

Source File

`brk.c`

6.4 close Function

Closes the file associated with the file descriptor.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<unistd.h>`

Prototype

```
int close(int handle);
```

Argument

`handle`	handle (file descriptor) referring to an opened file

Return Value

Returns 0 if the file is successfully closed or -1 on an error.

Remarks

This function is implemented as a stub that must be completed to suite the application.

This helper function is called by the `fclose()` Standard C Library function.

This function should be implemented to close a file in a manner that is relevant to your application. A file need not necessarily be associated with a storage device. This function should return -1 to signal an error and a strict implementation will set `errno` to some appropriate value such as `EBADF` or `EIO`.

Default Behavior

As distributed, this function passes the file handle to the simulator, which issues a close in the host file system.

Source File

`close.c`

6.5 creat Function

Obsolete function

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<fcntl.h>`

Prototype

```
int creat(const char * filename, mode_t mode)
```

Arguments

filename	name of the file to open
mode	the file permission bits

Return Value

On success, `creat` returns a file descriptor to the open file; on error it returns -1 and should set `errno` to indicate the error.

Remarks

This function is implemented as a stub that must be completed to suite the application.

Opens the specified filename for writing, truncating to length zero, and creating it if it does not exist.

Source File

`creat.c`

6.6 fcntl Function

Performs the operation specified by `command` on the file descriptor `filedes`.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<fcntl.h>`

Prototype

`int fcntl(int filedes, int command, …)`

Arguments

filedes	file descriptor on which the **command** is performed
command	command performed on file descriptor. Some commands require additional arguments to be supplied
...	the data associated with the command to be executed

Return Value

The return value and error conditions are dependent on the individual commands.

Remarks

This function is implemented as a stub that must be completed to suite the application.

The following table shows the available commands.

F_DUPFD	Duplicate the file descriptor (return another file descriptor pointing to the same open file).
F_GETFD	Get flags associated with the file descriptor.
F_SETFD	Set flags associated with the file descriptor.
F_GETFL	Get flags associated with the open file.
F_SETFL	Set flags associated with the open file.
F_GETLK	Test a file lock.
F_SETLK	Set or clear a file lock.
F_SETLKW	Like F_SETLK, but wait for completion.
F_OFD_GETLK	Test an open file description lock.
F_OFD_SETLK	Set or clear an open file description lock
F_OFD_SETLKW	Like F_OFD_SETLK, but block until lock is acquired.
F_GETOWN	Get process or process group ID to receive SIGIO signals.
F_SETOWN	Set process or process group ID to receive SIGIO signals.

Source File

`fcntl.c`

6.7 getpid Function

Return the process ID of the current process.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<unistd.h>`

Prototype

```
pid_t getpid (void)
```

Return Value

The `pid_t` data type is a signed integer type which is capable of representing a process ID. In the GNU C Library, this is an `int`.

Remarks

This function is implemented as a stub that must be completed to suite the application.

A process ID uniquely identifies a process only during the lifetime of the process. As a rule of thumb, this means that the process must still be running.

Source File

`getpid.c`

6.8 isatty Function

Test whether a file descriptor refers to a terminal

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

```
<unistd.h>
```

Prototype

```
int isatty(int fd);
```

Argument

`fd`	file descriptor to be tested

Return Value

Returns 1 if `fd` is an open file descriptor referring to a terminal; otherwise 0 is returned, and `errno` is set to indicate the error. Suitable values may be `EBADF` or `ENOTTY`.

Remarks

This function is implemented as a stub that must be completed to suite the application.

The `isatty()` function tests whether `fd` is an open file descriptor referring to a terminal.

Source File

`isatty.c`

6.9 link Function

Create a new file.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

<unistd.h>

Prototype

```
int link(const char * existing, const char * new);
```

Arguments

existing	filename from which to link
new	destination filename of link

Return Value

Zero is returned to indicate success, and -1 is returned to indicate an error condition.

Remarks

This function is implemented as a stub that must be completed to suite the application.

This function is not provided by default. Its purpose, in a file system, is to create a new filename, `new`, which contains the same data as the file named `existing`. The `errno` object should also be set on error. This function is used by `rename`.

Source File

`link.c`

6.10 lseek Function

Move a file pointer to a specified location.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

<unistd.h>

Prototype

```
long lseek(int handle, long offset, int origin);
```

Arguments

handle	handle (file descriptor) referring to an opened file
offset	the number of characters from the origin
origin	the position from which to start the seek

Return Value

Returns the offset, in characters, of the new position from the beginning of the file. The function returns -1 to indicate an error and sets `errno`. Appropriate values might be `EBADF` or `EINVAL`.

Remarks

This function is implemented as a stub that must be completed to suite the application.

This helper function is called by the Standard C Library functions `fgetpos()`, `ftell()`, `fseek()`, `fsetpos` and `rewind()`.

The origin argument may be one of the following values (as defined in <stdio.h>):

Origin	
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End-of-file

Default Behavior

As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.

Source File

`lseek.c`

6.11 open Function

Open a file.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

None

Prototype

```
int open(const char *name, int access, int mode);
```

Arguments

name	name of the file to be opened
access	access method to open file
mode	type of access permitted

Return Value

If successful, the function returns a file handle: a small positive integer. This handle is then used on subsequent low-level file I/O operations. A return value of -1 indicates an error, and the global variable `errno` is set to the value of the symbolic constant, `EFOPEN`, defined in `<errno.h>`.

Remarks

This function is implemented as a stub that must be completed to suite the application.

This helper function is called by the Standard C Library functions `fopen()` and `freopen()`.

The access flag is a union of one of the following access methods and zero or more access qualifiers:

- 0 – Open a file for reading.
- 1 – Open a file for writing.
- 2 – Open a file for both reading and writing.

The following access qualifiers must be supported:

- 0x0008 – Move file pointer to end-of-file before every write operation.
- 0x0100 – Create and open a new file for writing.
- 0x0200 – Open the file and truncate it to zero length.
- 0x4000 – Open the file in text (translated) mode.

- 0x8000 – Open the file in binary (untranslated) mode.

The mode parameter may be one of the following:

- 0x0100 – Reading only permitted.
- 0x0080 – Writing permitted (implies reading permitted).

Default Behavior

As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.

Source File

`open.c`

6.12 read Function

Read data from a file.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<unistd.h>`

Prototype

```
int read(int handle, void *buffer, unsigned int len);
```

Arguments

handle	handle (file descriptor) referring to an opened file
buffer	points to the storage location for read data
len	the maximum number of characters to read

Return Value

This function should be implemented to returns the number of characters read, which may be less than `len` if there are fewer than `len` characters left in the file or if the file was opened in text mode, in which case, each carriage return-linefeed (CR-LF) pair is replaced with a single linefeed character. Only the single linefeed character is counted in the return value. The replacement does not affect the file pointer. If the function tries to read at end-of-file, it returns 0. If the handle is invalid, or the file is not open for reading or the file is locked, the function returns -1 and `errno` is set to indicate the kind of error. Appropriate values may be `EBADF` or `EINVAL`, among others.

Remarks

This function is implemented as a stub that must be completed to suite the application.

This helper function is called by the Standard C Library functions `fgetc()`, `fgets()`, `fread()` and `gets()`.

Default Behavior

As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.

Source File

`read.c`

6.13 sbrk Function

Extend the process' data space by a given increment.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

None

Prototype

```
void * sbrk(int incr);
```

Argument

incr	number of characters to increment/decrement

Return Value

Return the start of the new space allocated or -1 for errors.

Remarks

This is a helper function called by the Standard C Library function `malloc()`.

`sbrk()` adds `incr` characters to the break value and changes the allocated space accordingly. `incr` can be negative, in which case the amount of allocated space is decreased.

`sbrk()` is used to dynamically change the amount of space allocated for the calling process's data segment. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases.

Default Behavior

A static variable is used to point to the first free heap location. If adding `incr` number of bytes will exceed `_eheap`, `sbrk()` will return -1. Otherwise, the function updates the first free location on heap and return a pointer to latest allocated memory.

The linker allocates a block of memory for the heap if the `-Wl,--heap=`*n* option is specified, where *n* is the desired heap size in characters. The starting and ending addresses of the heap are reported in variables: `_heap` and `_eheap`, respectively, and which are declared:

```
extern uint8_t _heap;              // heap start
extern uint8_t _eheap;             // heap end
```

For MPLAB XC16, using the linker's heap size option is the standard way of controlling heap size, rather than relying on `brk()` and `sbrk()`.

See also `brk()`.

Source File

`sbrk.c`

6.14 unlink Function

Low level command to remove a file link.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<unistd.h>`

Prototype

```
int unlink(const char * name);
```

Argument

name	file to be removed

Return Value

Returns zero if successful and -1 to signify an error.

Remarks

This function is implemented as a stub that must be completed to suite the application.

This function is not provided by default and is required for `remove` and `rename`. This function deletes a link between a filename and the file contents. The contents are also deleted when the last link is destroyed. A file may have multiple links to it if the `link` function has been used.

Source File

`unlink.c`

6.15 write Function

Write data to a file.

Attention: This interface function is implemented only by MPLAB XC32 C compilers.

Include

`<unistd.h>`

Prototype

```
int write(int handle, void * buffer, size_t count);
```

Arguments

handle	handle (file descriptor) referring to an opened file
buffer	points to the storage location of data to be written
count	the number of characters to write

Return Value

If successful, write returns the number of characters actually written. A return value of -1 indicates an error, in which case `errno` should be set to indicate the type of error. Suitable values may be `EBADF` or `EINVAL`, among others.

Remarks

This function is implemented as a stub that must be completed to suite the application.

This is a helper function called by the Standard C Library function `fflush()`.

If the actual space remaining on the disk is less than the size of the buffer, the function trying to write to the disk write fails and does not flush any of the buffer's contents to the disk. If the file is opened in text mode, each linefeed character is replaced with a carriage return-linefeed pair in the output. The replacement does not affect the return value.

Default Behavior

As distributed, the parameters are passed to the host file system through the simulator. The return value is the value returned by the host file system.

Source File

`write.c`

7. Document Revision History

Revision A (October 2021)

- Initial release of this document.

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods being used in attempts to breach the code protection features of the Microchip devices. We believe that these methods require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Attempts to breach these code protection features, most likely, cannot be accomplished without violating Microchip's intellectual property rights.
- Microchip is willing to work with any customer who is concerned about the integrity of its code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is "unbreakable." Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Information contained in this publication is provided for the sole purpose of designing with and using Microchip products. Information regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications.

THIS INFORMATION IS PROVIDED BY MICROCHIP “AS IS”. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

AgileSwitch, APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, IdealBridge, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, Inter-Chip Connectivity, JitterBlocker, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SMART-I.S., storClad, SQI, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, TSHARC, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

ISBN: 978-1-5224-9067-8

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4485-5910 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-72400 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-72884388 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820