MPLAB® XC8 C Compiler User's Guide for AVR® MCU

Notice to Customers

All documentation becomes dated and this manual is no exception. Microchip tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions can differ from those in this document. Please refer to our web site (https://www.microchip.com) to obtain the latest documentation available.

Documents are identified with a "DS" number. This number is located on the bottom of each page, in front of the page number. The numbering convention for the DS number is "DSXXXXXA," where "XXXXX" is the document number and "A" is the revision level of the document.

For the most up-to-date information on development tools, see the MPLAB® IDE online help. Select the Help menu, and then Topics to open a list of available online help files.

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	<u>*File>Save*</u>
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 Development Systems Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata that are related to a specified product family or development tool of interest.

To register, access the Microchip web site at https://www.microchip.com, click on Customer Change Notification and follow the registration instructions.

The Development Systems product group categories are:

Compilers	The latest information on Microchip C compilers, assemblers, linkers and other language tools. These include all MPLAB® C compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
Emulators	The latest information on Microchip in-circuit emulators. This includes the MPLAB REAL ICE™ and MPLAB ICE 2000 in-circuit emulators.
In-Circuit Debuggers	The latest information on the Microchip in-circuit debuggers. This includes MPLAB ICD 3 in-circuit debuggers and PICkitTM 3 debug express.
MPLAB® IDE	The latest information on Microchip MPLAB IDE, the Windows™ Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.
Programmers	The latest information on Microchip programmers. These include production programmers such as MPLAB REAL ICE in-circuit emulator, MPLAB ICD 3 in-circuit debugger and MPLAB PM3 device programmers. Also included are non-production development programmers such as PICSTART® Plus and PICkit 2 and 3.

2. Compiler Overview

The MPLAB XC8 C Compiler is a free-standing, optimizing ISO C99 cross compiler for the C programming language.

It supports all 8-bit PIC® and AVR® microcontrollers; however, this document describes the use of the `xc8-cc` driver and assumes that programs are built for Microchip 8-bit AVR devices. See the MPLAB® XC8 C Compiler User's Guide for PIC® MCU (DS50002737), for information on using this compiler when targeting Microchip PIC devices.

Note: Features described as being part of MPLAB XC8 in this document assume that you are using a Microchip AVR device. These features may differ if you choose to instead compile for a Microchip PIC device.

2.1 Device Description

This compiler guide describes the MPLAB XC8 compiler's support for all 8-bit Microchip AVR devices, including tinyAVR, and AVR XMEGA devices.

The compiler takes advantage of the target device's instruction set, addressing modes, memory, and registers whenever possible. A summary of the device families is shown below. This includes the offset of the special function registers and the offset at which program memory is mapped into the data space (where relevant). See 3.6.2.4 Print-devices for information on finding the full list of devices that are supported by the compiler.

Table 2-1. Summary of Supported Device Families

Family	ArchID	SFR Offset	Mapped Flash Address
avr1	1	0x20	n/a
avr2	2	0x20	n/a
avr25	25	0x20	n/a
avr3	3	0x20	n/a
avr31	31	0x20	n/a
avr35	35	0x20	n/a
avr4	4	0x20	n/a
avr5	5	0x20	n/a
avr51	51	0x20	n/a
avr6	6	0x20	n/a
avrtiny	100	0x0	0x4000
avrxmega2	102	0x0	n/a
avrxmega3	103	0x0	0x8000
avrxmega4	104	0x0	n/a
avrxmega5	105	0x0	n/a
avrxmega6	106	0x0	n/a
avrxmega7	107	0x0	n/a

2.2 C Standards

This compiler is a freestanding implementation that conforms to the ISO/IEC 9899:1990 Standard (referred to as the C90 standard) as well the ISO/IEC 9899:1999 Standard (C99) for programming languages, unless otherwise stated. In addition, language extensions customized for 8-bit AVR embedded-control applications are included.

2.3 Hosts and Licenses

The MPLAB XC8 C Compiler is available for several popular operating systems. See the compiler release notes for those that apply to your compiler version.

The compiler can be run with or without a license. A license can be purchased and applied at any time, permitting a higher level of optimization to be employed. Otherwise, the basic compiler operation, supported devices and available memory when using an unlicensed compiler are identical to those when using a licensed compiler.

2.4 Conventions

Throughout this manual, the term “compiler” is used. It can refer to all, or a subset of, the collection of applications that comprise the MPLAB XC8 C Compiler. When it is not important to identify which application performed an action, it will be attributed to “the compiler.”

In a similar manner, “compiler” is often used to refer to the command-line driver; although specifically, the driver for the MPLAB XC8 C Compiler package is named `xc8-cc`. The driver and its options are discussed in 3.6 Option Descriptions. Accordingly, “compiler options” commonly refers to command-line driver options.

In a similar fashion, “compilation” refers to all or a selection of steps involved in generating an executable binary image from source code.

2.5 Compatible Development Tools

The compiler works with many other Microchip tools, including:

- The MPLAB X IDE (https://www.microchip.com/mplab/mplab-x-ide)
- The MPLAB X Simulator
- The Command-line MDB Simulator—see the Microchip Debugger (MDB) User’s Guide (DS52102)
- All Microchip debug tools and programmers (https://www.microchip.com/mplab/development-boards-and-tools)
- Demonstration boards and Starter kits that support 8-bit devices

3. Command-line Driver

The MPLAB XC8 C Compiler command-line driver, `xc8-cc`, can be invoked to perform all aspects of compilation, including C code generation, assembly and link steps. Its use is the recommended way to invoke the compiler, as it hides the complexity of all the internal applications and provides a consistent interface for all compilation steps. Even if an IDE is used to assist with compilation, the IDE will ultimately call `xc8-cc`.

If you are building a legacy project or would prefer to use the old command-line driver you may instead run the `avr-gcc` driver application and use appropriate command-line options for that driver. Those options may differ to those described in this guide.

This chapter describes the steps that the driver takes during compilation, the files that the driver can accept and produce, as well as the command-line options that control the compiler's operation.

3.1 Invoking The Compiler

This section explains how to invoke `xc8-cc` on the command line and discusses the input files that can be passed to the compiler.

3.1.1 Driver Command-line Format

The `xc8-cc` driver can be used to compile and assemble C and assembly source files, as well as link object files and library archives to form a final program image.

The driver has the following basic command format:

```
xc8-cc [options] files
```

So, for example, to compile and link the C source file `hello.c`, you could use the command:

```
xc8-cc -mcpu=atmega3290p -O2 -o hello.elf hello.c
```

Throughout this manual, it is assumed that the compiler applications are in your console's search path. See 3.1.2 Driver Environment Variables for information on the environment variable that specifies the search locations. Alternatively, use the full directory path along with the driver name when executing the compiler.

It is customary to declare *`options`* (identified by a leading dash "`-`" or double dash "`--`") before the files' names; however, this is not mandatory.

Command-line options are case sensitive, with their format and description being supplied in 3.6 Option Descriptions. Many of the command-line options accepted by `xc8-cc` are common to all the MPLAB XC compilers, to allow greater portability between devices and compilers.

The *`files`* can be any mixture of C and assembler source files, as well as relocatable object files and archive files. While the order in which these files are listed does not directly affect the operation of the program, it can affect the allocation of code or data. Note, that the order of the archive files will dictate the order in which they are searched, and in some situations, this might affect which modules are linked in to the program.

3.1.1.1 Long Command Lines

The `xc8-cc` driver can be passed a command-line file containing driver options and arguments to circumvent any operating-system-imposed limitation on command line length.

A command file is specified by the `@` symbol, which should be immediately followed (i.e., no intermediate space character) by the name of the file containing the arguments. This same system of argument passing can be used by most of the internal applications called by the compiler driver.

Inside the file, each argument must be separated by one or more spaces and can extend over several lines when using a backslash-return sequence. The file can contain blank lines, which will be ignored.

The following is the content of a command file, `xyz.xc8` for example, that was constructed in a text editor and that contains the options and the file names required to compile a project.

```
-mcpu=atmega3290p -Wl,-Map=proj.map -Wa,-a \
-O2 main.c isr.c
```

After this file is saved, the compiler can be invoked with the following command:

```
xc8-cc @xyz.xc8
```

Command files can be used as a simple alternative to a make file and utility, and can conveniently store compiler options and source file names.

3.1.2 Driver Environment Variables

No environment variables are defined or required by the compiler for it to execute.

Adjusting the `PATH` environment variable allows you to run the compiler driver without having to specify the full compiler path.

This variable can be automatically updated when installing the compiler by selecting the **Add xc8 to the path environment variable** checkbox in the appropriate dialog.

Note that the directories specified by the `PATH` variable are only used to locate the compiler driver. Once the driver is running, it will manage access to the internal compiler applications, such as the assembler and linker, etc.

The MPLAB X IDE allows the compiler to be selected via the **Project properties** dialog without the need for the `PATH` variable.

3.1.3 Input File Types

The `xc8-cc` driver distinguishes source files, intermediate files and library files solely by the file type, or extension. The following case-sensitive extensions, listed in Table 3-1 are recognized.

Table 3-1. Input File Types

Extension	File format
`.c`	C source file
`.i`	Preprocessed C source file
`.s`	Assembler source file
`.S`	Assembly source file requiring preprocessing
`.o`	Relocatable object code file
`.a`	Archive (library) file
other	A file to be passed to the linker

There are no compiler restrictions imposed on the names of source files, but be aware of case, name-length, and other restrictions that are imposed by your host operating system.

Avoid using the same base name for assembly and C source files, even if they are located in different directories. So, for example, if a project contains a C source file called `init.c`, do not also add to the project an assembly source file with the name `init.s`. Avoid, also, having source files with the same base name as the MPLAB X IDE project name.

The terms *source file* and *module* are often used interchangeably, but they refer to the source code at different points in the compilation sequence.

A source file is a file that contains all or part of a program. It may contain C code, as well as preprocessor directives and commands. Source files are initially passed to the preprocessor by the compiler driver.

A module is the output of the preprocessor for a given source file, after the inclusion of any header files specified by`#include` preprocessor directives, and after the processing and subsequent removal of other preprocessor directives (with the possible exception of some commands for debugging). Thus, a module is usually the amalgamation of a source file and several header files, and it is this output that is passed to the remainder of the compiler applications. A module is also referred to as a *translation unit*.

Like assembly source files, these terms can also be applied to assembly files, which can be preprocessed and can include other header files.

3.2 The Compilation Sequence

When you compile a project, many internal applications are called by the driver to do the work. This section introduces these internal applications and describes how they relate to the build process, especially when a project consists of multiple source files. This information should be of particular interest if you are using a make system to build projects.

3.2.1 The Compiler Applications

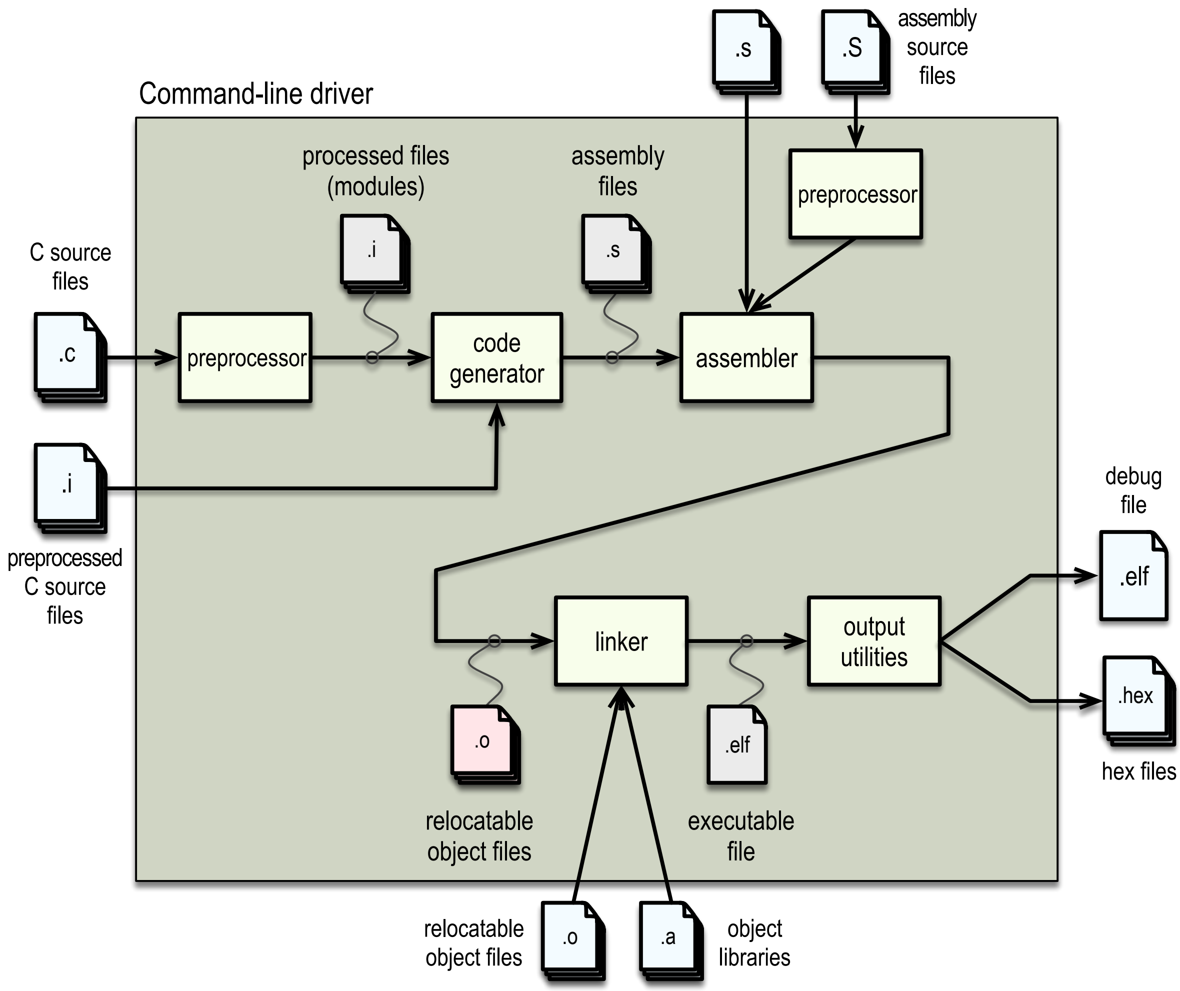
The main internal compiler applications and files are shown in the illustration below.

The large shaded box represents the compiler, which is controlled by the command line driver, `xc8-cc`. You might be satisfied just knowing that C source files (shown on the far left) are passed to the compiler and the resulting output files (shown here as a HEX and ELF debug file on the far right) are produced; however, internally there are many applications and temporary files being produced. An understanding of the internal operation of the compiler, while not necessary, does assist with using the tool.

The driver will call the required compiler applications when required. These applications are located in the compiler's `bin` directories and are shown in the diagram as the smaller boxes inside the driver.

The temporary files produced by each application can also be seen in this diagram and are marked at the point in the compilation sequence where they are generated. The intermediate files for C source are shaded in red. Some of these temporary files remain after compilation has concluded. There are also driver options to request that the compilation sequence halt after execution of a particular application so that the output of that application remains in a file and can be examined.

Figure 3-1. Compiler Applications And Files



It is recommended that only the `hexmate` and archiver (`xc8-ar`) internal applications be executed directly. Their command-line options are described in 5. Utilities.

3.2.2 Single-Step C Compilation

Full compilation of one or more C source files, including the link step, can be performed in just one command using the `xc8-cc` driver.

3.2.2.1 Compiling a Single C File

The following is a simple C program that adds two numbers. To illustrate how to compile and link a program consisting of a single C source file, copy the code into any text editor and save it as a plain text file with the name `ex1.c`.

```
#include <xc.h>

unsigned int
add(unsigned int a, unsigned int b)
{
  return a + b;
}

int
main(void)
{
  unsigned int x, y, z;
  x = 2;
  y = 5;
```

```
  z = add(x, y);

  return 0;
}
```

In the interests of clarity, this code does not specify device configuration bits, nor has any useful purpose.

Compile the program by typing the following command at the prompt in your favorite terminal. For the purpose of this discussion, it is assumed that in your terminal you have changed into the directory containing the source file you just created, and that the compiler is installed in the standard directory location and is in your host's search path.

```
xc8-cc -mcpu=atmega3290p  -o ex1.elf ex1.c
```

This command compiles the `ex1.c` source file for a atmega3290p device and has the output written to `ex1.elf`, which may be loaded into the MPLAB X IDE.

If a hex file is required, for example, to load into a device programmer, then use the following command:

```
avr-objcopy -O ihex a.out ex1.hex
```

This creates an Intel hex file named `ex1.hex`.

The driver will compile the source file, regardless of whether it has changed since the last build command. Development environments (such as MPLAB X IDE) and make utilities must be employed to achieve incremental builds (see 3.2.3 Multi-Step C Compilation).

Unless otherwise specified, an ELF file (this is by default called `a.out`) is produced as the final output.

The intermediate files remain after compilation has completed, but most other temporary files are deleted, unless you use the `-save-temps` option (see 3.6.5.4 Save-temps Option) which preserves all generated files except the run-time start-up file. Note that some generated files can be in a different directory than your project source files (see also 3.6.2.3 O: Specify Output File).

3.2.2.2 Compiling Multiple C Files

This section demonstrates how to compile and link a project, in a single step, that consists of multiple C source files.

Copy the example code shown into a text file called `add.c`.

```
/* add.c */
#include <xc.h>

unsigned int
add(unsigned int a, unsigned int b)
{
  return a + b;
}
```

And place the following code in another file, `ext.c`.

```
/* ex1.c */
#include <xc.h>

unsigned int add(unsigned int a, unsigned int b);

int
main(void) {
  unsigned int x, y, z;
  x = 2;
  y = 5;
  z = add(x, y);

  return 0;
}
```

In the interests of clarity, this code does not specify device configuration bits, nor has any useful purpose.

Compile both files by typing the following at the prompt:

```
xc8-cc -mcpu=atmega3290p ex1.c add.c
```

```
xc8-cc -mcpu=atmega3290p  -o ex1.elf ex1.c add.c
```

This command compiles the modules `ex1.c` and `add.c` in the one step. The compiled modules are linked with the relevant compiler libraries and the executable file `ex1.elf` is created.

3.2.3 Multi-Step C Compilation

A multi-step compilation method can be employed to build projects consisting of one or more C source files. Make utilities can use this feature, taking note of which source files have changed since the last build to speed up compilation. Incremental builds are also be performed by integrated development environments, such as the MPLAB X IDE when selecting the Build Project icon or menu item.

Make utilities typically call the compiler multiple times: once for each source file to generate an intermediate file and once to perform the second stage compilation, which links the intermediate files to form the final output. If only one source file has changed since the last build, the intermediate file corresponding to the unchanged source file need not be regenerated.

For example, the files `ex1.c` and `add.c` are to be compiled using a make utility. The command lines that the make utility could use to compile these files might be something like:

```
xc8-cc -mcpu=atmega3290p -c ex1.c
xc8-cc -mcpu=atmega3290p -c add.c
```

```
xc8-cc -mcpu=atmega3290p  -o ex1.elf ex1.o add.o
```

The `-c` option used with the first two commands will compile the specified file into the intermediate file format, but not link. The resultant intermediate files are linked in the final step to create the final output `ex1.elf`. All the files that constitute the project must be present when performing the second stage of compilation.

The above example uses the command-line driver, `xc8-cc`, to perform the final link step. You can explicitly call the linker application, `avr-ld`, but this is not recommended as the commands are complex and when driving the linker application directly, you must specify linker options, not driver options, as shown above.

You may also wish to generate intermediate files to construct your own library archive files.

3.2.4 Compilation of Assembly Source

Assembly source files that are part of a C project are compiled in a similar way to C source files. The compiler driver knows that these files should be passed to a different set of internal compiler applications and a single build command can contain a mix of C and assembly source files, as in the following example.

```
xc8-cc -mcpu=atmega3290p proj.c spi.s
```

If an assembly source file contains C preprocessor directives that must be preprocessed before passed to the assembler, then ensure the source file uses a `.S` extension, for example `spi.S`.

The compiler can be used to generate assembly files from C source code using the `-S` option. The assembly output can then be used as the basis for your own assembly routines and subsequently compiled using the command-line driver.

3.3 Runtime Files

In addition to the C and assembly source files and user-defined libraries specified on the command line, the compiler can also link into your project compiler-generated source files and pre-compiled library files, whose content falls into the following categories:

- C standard library routines

- Implicitly called arithmetic library routines
- The runtime start-up code

3.3.1 Library Files

The C standard libraries contain a standardized collection of functions, such as string, math and input/output routines. The usage and operation of these functions is described in 7. Library Functions. For more information on creating and using your own libraries, see 4.10 Libraries.

These libraries are built multiple times with a permuted set of options. When the compiler driver is called to compile and link an application, the driver chooses the appropriate target library that has been built with the same options. You do not normally need to specify the search path for the standard libraries, nor manually include library files into your project.

3.3.1.1 Location and Naming Convention

The standard libraries, such as `libc.a` are found in the `avr/avr/lib` directory. Emulation routines for operations not natively supported in hardware are part of `libgcc.a`, found in `avr/lib/gcc/avr/`. The `libm.a` math library is also automatically linked in, as is `lib`*`devicename`*`.a` (e.g. `libatxmega128b1.a`) that contains device-specific routines for working with watch dog timers, power management, eeprom access, etc., (see 7. Library Functions for more information).

3.3.2 Startup and Initialization

The runtime startup code performs initialization tasks that must be executed before the `main()` function in the C program is executed. For information on the tasks performed by this code, see 4.9 Main, Runtime Startup and Reset.

The compiler will select the appropriate runtime startup code, based on the selected target device and other compiler options.

3.3.2.1 Runtime Startup Code

Pre-built object files, which contain the runtime startup code, are provided with the compiler. These form part of the device family packs, and can be found in `<dfp>/avr/lib`, under the relevant directory for your project's target device.

Should you require any special initialization to be performed immediately after Reset, you should write a powerup initialization routine (described later in 4.9.3 The Powerup Routine).

3.4 Compiler Output

There are many files created by the compiler during compilation. A large number of these are temporary or intermediate files that are deleted after compilation is complete; however, some files remain for programming or debugging the device, and options that halt compilation mid-process leave behind intermediate files, which may be inspected.

3.4.1 Output Files

The common output file types and case-sensitive extensions are shown in Table 3-2.

Table 3-2. Common Output Files

Extension	File Type	How created
`file``.hex`	Intel HEX	`avr-objcopy` application
`file``.elf`	ELF (Executable and Linkable Format) with Dwarf debugging information	`-o` option
`.s`	Assembly file	`-S` option
`.i`	Preprocessed C file	`-E` and `-o` option

The default behavior of `xc8-cc` is to produce an ELF file called `a.out`, unless you override that name using the `-o` option.

This behavior can be changed by using the `-gcoff` option, which generate a COFF file (described in 3.6.5.2 G: Produce Debugging Information Option).

The ELF/DWARF file is used by debuggers to obtain debugging information about the project and allows for more accurate debugging compared to the COFF format. Some aspects of the project's operation might not even be available to your debugger when using COFF. The MPLAB X IDE will typically request the compiler to produce an ELF file.

The names of many output files use the same base name as the source file from which they were derived. For example the source file `input.c` will create an object file called `input.o` when the `-c` option is used.

3.4.2 Diagnostic Files

Two valuable files that can be produced by the compiler are the assembly list file, generated by the assembler and the map file, generated by the linker. These are generated by options, shown in Table 3-3.

Table 3-3. Diagnostic Files

Extension	File Type	How Created
file.lst	Assembly list file	`-Wa,-a=file.lst` driver option
file.map	Map file	`-Wl,-Map=file.map` driver option

The assembly list file contains the mapping between the original source code and the generated assembly code. It is useful for information such as how C source was encoded, or how assembly source may have been optimized. It is essential when confirming if compiler-produced code that accesses objects is atomic and shows the region in which all objects and code are placed.

The assembler option to create a listing file is `-a` and can be passed to the assembler from the driver using the driver option `-Wa,-a=`*file*`.lst`, for example.

The map file shows information relating to where objects were positioned in memory. It is useful for confirming if user-defined linker options were correctly processed and for determining the exact placement of objects and functions.

The linker option to create a map file in the linker application is `-Map` *file*, and this option can be passed to the linker from the driver using the driver option `-Wl,-Map=`*file*`.map`, for example.

One map file is produced when you build a project, assuming that the linker was executed and ran to completion.

3.5 Compiler Messages

All compiler applications use textual messages to report feedback during the compilation process.

There are several types of messages, described below. The behavior of the compiler when encountering a message of each type is also listed.

Warning Messages Indicates source code or other situations that can be compiled, but is unusual and might lead to runtime failures of the code. The code or situation that triggered the warning should be investigated; however, compilation of the current module will continue, as will compilation of any remaining modules.

Error Messages Indicates source code that is illegal or that compilation of code cannot take place. Compilation will be attempted for the remaining source code in the current module (however the cause of the initial error might trigger further errors) and compilation of the other modules in the project will take place, but the project will not be linked.

Fatal Messages Indicates a situation in which compilation cannot proceed and which forces the compilation process to stop immediately.

3.5.1 Changing Message Behavior

You can change some attributes of compiler-generated messages and can sometimes disable messages entirely. The number of warning messages produced can also be controlled to assist with debugging.

3.5.1.1 Disabling Messages

The compiler will issue warnings to alert you to potential problems in your source code.

All warning messages can be disabled by using `-w` option.

You can turn off explicit warnings by using the `-Wno-`*`message`* option, where *`message`* relates to the warning type, for example, the `-Wno-return-type` option will prevent the warnings associated with functions whose return type defaults in `int`. When a warning is produced by the compiler, it prints in square brackets the associated warning option that controls this warning. For example, if the compiler issues the warning:

```
avr.c:13:1: warning: 'keep' attribute directive ignored [-Wattributes]
```

you can disable this warning using the option `-Wno-attributes`.

Note: Disabling warning messages in no way fixes the condition that triggered the message. Always use extreme caution when exercising these options.

You can enable a more complete set of warning messages about questionable constructions by using `-Wall`. The `-Wextra` option turns on additional messages. Alternatively, you can enable individual messages using the `-W`*`message`* option, for example `-Wunused-function` will ensure that warnings are produced for functions that are never called.

3.5.1.2 Changing Message Types

It is also possible to change the type (and hence behavior) of some messages.

Warnings can be turned into errors by using the `-Werror` option. Errors can be turned into fatal errors by using the `-Wfatal-errors` option.

3.6 Option Descriptions

Most aspects of the compilation process can be controlled using options passed to the command-line driver, `xc8-cc`.

The GCC compiler on which the MPLAB XC8 C Compiler is based provides many options in addition to those discussed in this document. It is recommended that you avoid any option that has not been documented here, especially those that control the generation or optimization of code.

All options are case sensitive and are identified by single or double leading dash character, e.g. `-c` or `--version`.

Use the `--help` option to obtain a brief description of accepted options on the command line.

If you are compiling from within the MPLAB X IDE, it will issue explicit options to the compiler that are based on the selections in the project's **Project Properties** dialog. The default project options might be different to the default options used by the compiler when running on the command line, so you should review these to ensue that they are acceptable.

3.6.1 Options Specific to AVR Devices

The options tabulated below are useful when compiling for 8-bit Microchip AVR devices with the MPLAB XC8 compiler and are discussed in the sections that follow.

Table 3-4. AVR Device-Specific Options

Option	Controls
`-m[no-]accumulate-args`	How arguments are passed between functions.
`-m[no-]call-prologues`	How functions save and restore registers.
`-mconst-data-in-progmem`	Whether `const`-qualified objects are placed in program or data memory.
`-mcpu=`*`device`*	The target device or architecture for which to compile.
`-mdfp=`*`path`*	The source of device-specific information.

...........continued	
Option	**Controls**
`-mno-interrupts`	How the stack pointer is changed.
`-fno-jump-tables`	Whether jump tables are used in `switch()` statements.
`-mrelax`	Optimization of call/jump instructions.
`-mshort-calls`	How function calls are encoded.
`-mstrict-X`	The use of the X register.
`-mtiny-stack`	The width of the stack pointer.

3.6.1.1 Accumulate-args Option

The `-maccumulate-args` option prevents function arguments from being pushed onto and popped off the stack, instead producing code that adjusts the stack pointer once at the beginning of the calling function. This option has no effect when functions that do not use the stack for arguments are called, but for other functions, it can reduce code size if those functions are called several times.

3.6.1.2 Call-prologues Option

The `-mcall-prologues` option changes how functions save registers on entry and how those registers are restored on function exit.

If this option is not specified or the `-mno-call-prologues` options is used, the registers that need to be preserved by each function will be saved and restored by code inside those functions. If the `-mcall-prologues` option is used, this preservation code is extracted as subroutines that are called at the appropriate points in the function.

Use of this option can reduce code size, but can increase the code's execution time.

3.6.1.3 Const-data-in-progmem Option

The `-mconst-data-in-progmem` option changes the location of where objects qualified with `const` are stored.

By default, `const`-qualified objects are located in and accessed from program memory and this location can be explicitly specified using the `-mconst-data-in-progmem` option. The `-mno-const-data-in-progmem` option forces all `const`-qualified data to be located in data memory, where it will be read using different instructions. This option does not affect code built for devices that have flash mapped into data memory, for example the avrxmega3 and avrtiny architectures.

Having `const`-qualified objects in program memory will free up more valuable RAM and does not require the use of non-standard keywords, making it easier to have pointers access such objects. Accessing `const`-qualified objects in data memory is the more efficient in terms of program size.

3.6.1.4 Cpu Option

The `-mcpu=`*`device`* option should be used to specify the target device or at least a target architecture family.

For example:

```
xc8-cc -mcpu=atmega161 main.c
```

To see a list of supported devices that can be used with this option, use the `-mprint-devices` option (see 3.6.2.4 Print-devices). The available architecture families are tabulated below.

Table 3-5. Selectable Architecture Families

Architecture	Architecture Features
`avr1`	Simple core, no data RAM, assembly support only.
`avr2`	Classic core, up to 8kB program memory.
`avr25`	avr2 with `movw` and `lpm Rx,Z[+]` instructions.
`avr3`	Classic core with up to 64kB extended program memory.

...........continued	
Architecture	**Architecture Features**
avr31	Classic core with 128kB of program memory.
avr35	avr3 with movw and lpm Rx,Z[+] instructions.
avr4	Enhanced core up to 8kB program memory.
avr5	Enhanced core up to 64kB program memory.
avr51	Enhanced core 128kB program memory.
avr6	Enhanced core 256kB program memory.
avrxmega2	XMEGA core, up to 64kB program memory, up to 64kB data address space.
avrxmega3	xmega2 devices with program memory mapped in data address space.
avrxmega4	XMEGA core, up to 128kB program memory, up to 64kB data address space.
avrxmega5	XMEGA core, up to 128kB program memory, greater than 64kB data address space.
avrxmega6	XMEGA core, greater than 128 kB program memory, up to 64kB data address space.
avrxmega7	XMEGA core, greater than 128 kB program memory, greater than 64kB data address space.
avrtiny	AVR Tiny core, 16 registers.

3.6.1.5 Dfp Option

The `-mdfp=`*`path`* option indicates that device-support for the target device (indicated by the `-mcpu` option) should be obtained from the contents of a Device Family Pack (DFP), where *`path`* is the path to the `xc8` sub-directory of the DFP.

When this option has not been used, the `xc8-cc` driver will where possible provide the path of the relevant DFP that is installed with the compiler, in *`<installation dir>`*`/dfp`.

The MPLAB X IDE automatically uses this option to inform the compiler of which device-specific information to use. Use this option on the command line if additional DFPs have been obtained for the compiler.

A DFP might contain such items as device-specific header files, configuration bit data and libraries, letting you take advantage of features on new devices without you having to otherwise update the compiler. DFPs never contain executables or provide bug fixes or improvements to any existing tools or standard library functions.

The preprocessor will search for include files in the *`<DFP>`*`/xc8/avr/include/` directory before search the standard search directories. For libraries, startup and specification files, the compiler will search *`<DFP>`*`/xc8/avr/` and *`<DFP>`*`/xc8/avr/lib/` before the standard search paths.

3.6.1.6 No-interrupts Option

The `-mno-interrupts` option controls whether interrupts should be disabled when the stack pointer is changed.

For most devices, the state of the status register, SREG, is saved in a temporary register and interrupts are disabled before the stack pointer is adjusted. The status register is then restored after the stack pointer has been changed.

If a program does not use interrupts, there is no need for the stack adjustments to be protected in this way. Use of this option omits the code that disables and potentially re-enables interrupts around the code the adjusts the stack pointer, thus reducing code size and execution time.

Since the AVR XMEGA devices and devices with an 8-bit stack pointer can change the value held by the stack pointer atomically, this option is not required and has no effect when compiling for one of these devices.

Specifying this option will define the preprocessor macro `__NO_INTERRUPTS__` to the value 1.

3.6.1.7 No-jump-tables Option

The `-fno-jump-tables` option controls how `switch()` statements are encoded. See 4.5.3 Switch Statements for full details regarding this option.

3.6.1.8 Relax Option

The `-mrelax` option controls the optimization of the long form of call and jump instructions, which are always output by the code generator into shorter and/or faster relative calls and jumps at link-time. These changes can only take place if the relative forms of the instructions can be determined to be in range of their destination (for more information see 4.3.6.3 Function Pointers).

3.6.1.9 Short-calls Option

The `-mshort-call` option controls how calls are encoded.

When building for devices which have more then 8kB of program memory, the compiler will automatically use the longer form of the jump and call instructions when program execution is leaving the current function. Doing so allows program execution to reach the entire memory space, but the program will be larger and take longer to execute. The `-mshort-calls` option will force calls to use PC-relative instructions such as the `rjmp` and `rcall` instructions, which have a limited destination range. This option has no effect on indirect jumps or calls made via function pointers.

Use this option with caution, as your code might fail if functions fall out of range of the shorter instructions. See the `-mrelax` option (3.6.1.8 Relax Option) to allow function pointers to be encoded as 16-bits wide, even on large memory device. This option has no effect for the avr2 and avr4 architectures, which have less than 8kB of program memory and which always use the shorter form of the call/jump instructions.

3.6.1.10 Strict-X Option

The `-mstrict-X` option ensures that the X register (r26-r27) is only used in indirect, post-increment or pre-decrement addressing. This restricts the register's usage, which could be beneficial in terms of code size.

3.6.1.11 Tiny-stack Option

The `-mtiny-stack` option controls the width of the stack pointer.

On some devices that have a small amount of data RAM, the stack pointer is only 8-bits wide. For other devices, it is 16-bits wide and occasionally each byte might need to be accessed separately to change where the stack pointer points.

If your devices uses a 16-bit stack pointer and the stack is located in the lower half of memory and does not exceed 256 bytes in size, this option will force the stack pointer to use only a single byte, thus reducing the amount of code necessary to adjust the stack pointer.

The option is automatically applied if the device implements RAM whose size is 256 bytes or less.

3.6.2 Options for Controlling the Kind of Output

The options tabulated below control the kind of output produced by the compiler and are discussed in the sections that follow.

Table 3-6. Kind-of-Output Control Options

Option	Produces
`-c`	An intermediate file.
`-E`	A preprocessed file.
`-o file`	An output file with the specified name.
`-mprint-devices`	Chip information only.
`-S`	An assembly file.
`-v`	Verbose compilation.
`-x language`	Output assuming that source files have the specified content.
`-###`	Command lines but with no execution of the compiler applications.
`--help`	Help information only.
`--version`	Compiler version information.

3.6.2.1 C: Compile To Intermediate File

The `-c` option is used to generate an intermediate file for each source file listed on the command line.

For all source files, compilation will terminate after executing the assembler, leaving behind relocatable object files with a `.o` extension.

This option is often used to facilitate multi-step builds using a make utility.

3.6.2.2 E: Preprocess Only

The `-E` option is used to generate preprocessed C source files (also called modules or translation units).

The preprocessed output is printed to `stdout`, but you can use the `-o` option to redirect this to a file.

You might check the preprocessed source files to ensure that preprocessor macros have expanded to what you think they should. The option can also be used to create C source files that do not require any separate header files. This is useful when sending files to a colleague or to obtain technical support without sending all the header files, which can reside in many directories.

3.6.2.3 O: Specify Output File

The `-o` option specifies the base name and directory of the output file.

The option `-o main.elf`, for example, will place the generated output in a file called `main.elf`. The name of an existing directory can be specified with the file name, for example `-o build/main.elf`, so that the output file will appear in that directory.

You cannot use this option to change the type (format) of the output file.

3.6.2.4 Print-devices

The `-mprint-devices` option displays a list of devices the compiler supports.

The names listed are those devices that can be used with the `-mcpu` option. This option will only show those devices that were officially supported when the compiler was released. Additional devices that might be available via device family packs (DFPs) will not be shown in this list.

The compiler will terminate after the device list has been printed.

3.6.2.5 S: Compile To Assembly

The `-S` option is used to generate an assembly file for each source file listed on the command line.

When this option is used, the compilation sequence will terminate early, leaving behind assembly files with the same basename as the corresponding source file and with a `.s` extension.

For example, the command:

```
xc8-cc -mcpu=atmega3290p -S test.c io.c
```

will produce two assembly file called `test.s` and `io.s`, which contain the assembly code generated from their corresponding source files.

This option might be useful for checking assembly code output by the compiler without the distraction of line number and opcode information that will be present in an assembly list file. The assembly files can also be used as the basis for your own assembly coding.

3.6.2.6 V: Verbose Compilation

The `-v` option specifies verbose compilation.

When this option is used, the name and path of the internal compiler applications will be displayed as they are executed, followed by the command-line arguments that each application was passed.

You might use this option to confirm that your driver options have been processed as you expect, or to see which internal application is issuing a warning or error.

3.6.2.7 X: Specify Source Language Option

The `-x` language option specifies the language for the following sources files.

The compiler usually uses the extension of an input file to determine the file's content. This option allows you to have the language of a file explicitly stated. The option remains in force until the next `-x` option, or the `-x none` option, which turns off the language specification entirely for subsequent files. The allowable languages are tabulated below.

Table 3-7. Source file Language

Language	File language
`assembler`	Assembly source
`assembler-with-cpp`	Assembly with C preprocessor directives
`c`	C source
`cpp-output`	Preprocessed C source
`c-header`	C header file
`none`	Based entirely on the file's extension

The `-x assembler-with-cpp` language option ensures assembly source files are preprocessed before they are assembled, thus allowing the use of preprocessor directives, such as `#include`, and C-style comments with assembly code. By default, assembly files not using a `.S` extension are not preprocessed.

You can create precompiled header files with this option, for example:

```
xc8-cc -mmcu=atxmega32d4 -x c-header init.h
```

will create the precompiled header called `init.h.gch`.

3.6.2.8 ### Option

The `-###` option is similar to `-v`, but the commands are not executed. This option allows you to see the compiler's command lines without executing the compiler.

3.6.2.9 Help

The `--help` option displays information on the `xc8-cc` compiler options, then the driver will terminate.

3.6.2.10 Version

The `--version` option prints compiler version information then exits.

3.6.3 Options for Controlling the C Dialect

The options tabulated below define the type of C dialect used by the compiler and are discussed in the sections that follow.

Table 3-8. C Dialect Control Options

Option	Controls
`-ansi`	Strict ANSI conformance.
`-aux-info` *filename*	The generation of function prototypes.
`-fno-asm`	Keyword recognition.
`-fno-builtin` `-fno-builtin-`*function*	Use of built-in functions.
`-f[no-]signed-char` `-f[no-]unsigned-char`	The signedness of a plain `char` type.
`-f[no-]signed-bitfields` `-f[no-]unsigned-bitfields`	The signedness of a plain `int` bit-field.
`-mext=`*extension*	Which language extensions is in effect.

..........continued	
Option	**Controls**
`-std=`*standard*	The C language standard.

3.6.3.1 Command Line Driver - Ansi Option

The `-ansi` option ensures the C program strictly conforms to the C90 standard.

When specified, this option disables certain GCC language extensions when compiling C source, such as C++ style comments, and keywords such as `asm` and `inline`. The macro `__STRICT_ANSI__` is defined when this option is in use. See also `-Wpedantic` for information on ensuring strict ISO compliance.

3.6.3.2 Aux-info Option

The `-aux-info` option generates function prototypes from a C module.

The `-aux-info main.pro` option, for example, prints to `main.pro` prototyped declarations for all functions declared and/or defined in the module being compiled, including those in header files. Only one source file can be specified on the command line when using this option so that the output file is not overwritten. This option is silently ignored in any language other than C.

The output file also indicates, using comments, the source file and line number of each declaration, whether the declaration was implicit, prototyped or unprototyped. This is done by using the codes `I` or `N` for new-style and `O` for old-style (in the first character after the line number and the colon) and whether it came from a declaration or a definition using the codes `C` or `F` (in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided inside comments after the declaration.

For example, compiling with the command:

```
xc8-cc -mmcu=atmega3290p -aux-info test.pro test.c
```

might produce `test.pro` containing the following declarations, which can then be edited as necessary:

```
/* test.c:2:NC */ extern int add (int, int);
/* test.c:7:NF */ extern int rv (int a); /* (a) int a; */
/* test.c:20:NF */ extern int main (void); /* () */
```

3.6.3.3 Ext Option

The `-mext=`*extension* option controls the language extension used during compilation. The allowed extensions are shown in the following Table.

Table 3-9. Acceptable C Language Extensions

Extension	C Language Description
`xc8`	The native XC8 extensions (default)
`cci`	A common C interface acceptable by all MPLAB XC compilers

Enabling the `cci` extension requests the compiler to check all source code and compiler options for compliance with the Common C Interface (CCI). Code that complies with this interface can be more easily ported across all MPLAB XC compilers. Code or options that do not conform to the CCI will be flagged by compiler warnings.

3.6.3.4 No-asm Option

The `-fno-asm` option restricts the recognition of certain keywords, freeing them to be used as identifiers.

When used, this option ensures that `asm`, `inline` and `typeof` are not recognized as keywords. You can, instead, use the keywords `__asm__`, `__inline__` and `__typeof__`, which have identical meanings.

The `-ansi` option implies `-fno-asm`.

3.6.3.5 No-builtin Option

The `-fno-builtin` option will prevent the compiler from producing special code for built-in functions that are not prefixed with `__builtin_`.

Normally specialized code that avoids a function call is produced for many built-in functions. The resulting code is often smaller and faster, but since calls to these functions no longer appear in the output, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library.

The `-fno-builtin-`*`function`* form of this option allows you to prevent a built-in version of the named function from being used. In this case, *`function`* must not begin with `__builtin_`.

3.6.3.6 Signed-bitfields Option

The `-fsigned-bitfield` option control the signedness of a plain `int` bit-field type.

By default, the plain `int` type, when used as the type of a bit-field, is equivalent to `signed int`. This option specifies the type that will be used by the compiler for plain `int` bit-fields. Using `-fsigned-bitfield` or the `-fno-unsigned-bitfield` option forces a plain `int` bit-field to be signed.

Consider explicitly stating the signedness of bit-fields when they are defined, rather than relying on the type assigned to a plain `int` bit-field type.

3.6.3.7 Signed-char Option

The `-fsigned-char` option forces a plain `char` objects to have a signed type.

By default, the plain `char` type is equivalent to `signed char`, unless the `-mext=cci` option has been used, in which case it is equivalent to `unsigned char`.

The `-fsigned-char` (or `-fno-unsigned-char` option) forces a plain `char` to be signed.

Consider explicitly stating the signedness of `char` objects when they are defined, rather than relying on the type assigned to a plain `char` type by the compiler.

3.6.3.8 Std Option

The `-std=`*`standard`* option specifies the C standard to which the compiler assumes source code will conform. Allowable standards and devices are tabulated below.

Table 3-10. Acceptable C Language Standards

Standard	Supports
`c89` or `c90`	ISO C90 (ANSI) programs
`c99`	ISO C99 programs

3.6.3.9 Unsigned-bitfields Option

The `-funsigned-bitfield` option control the signedness of a plain `int` bit-field type.

By default, the plain `int` type, when used as the type of a bit-field, is equivalent to `signed int`. This option specifies the type that will be used by the compiler for plain `int` bit-fields. Using the `-funsigned-bitfield` or the `-fno-signed-bitfield` option forces a plain `int` to be unsigned.

Consider explicitly stating the signedness of bit-fields when they are defined, rather than relying on the type assigned to a plain `int` bit-field type.

3.6.3.10 Unsigned-char Option

The `-funsigned-char` option forces a plain `char` objects to have an unsigned type.

By default, the plain `char` type is equivalent to `signed char`, unless the `-mext=cci` option has been used, in which case it is equivalent to `unsigned char`. The `-funsigned-char` (or `-fno-signed-char` option) makes this type explicit.

Consider explicitly stating the signedness of `char` objects when they are defined, rather than relying on the type assigned to a plain `char` type by the compiler.

3.6.4 Options for Controlling Warnings and Errors

Warnings are diagnostic messages that report constructions that, while not inherently erroneous, indicate source code or some other situation is unusual and might lead to runtime failures of the code.

You can request specific warnings using options beginning with `-W`; for example, `-Wimplicit`, to request warnings on implicit declarations. Each of these specific warning options also has a negative form, beginning `-Wno-` to turn off warnings; for example, `-Wno-implicit`.

The options shown in Table 3-11 and Table 3-12 are the more commonly used warning options that control the messages produced by the compile. In the sections that follow, the options that affect warnings generally are discussed.

Table 3-11. Warning and Error Options Implied By All Warnings

Option	Controls
`-f[no-]syntax-only`	Checking code for syntax errors only
`-pedantic`	Warnings demanded by strict ANSI C; rejects all programs that use forbidden extensions
`-pedantic-errors`	Warnings implied by -pedantic, except that errors are produced rather than warnings
`-w`	Suppression of all warning messages
`-W[no-]all`	Enablement of all warnings
`-W[no-]address`	Warnings from suspicious use of memory addresses
`-W[no-]char-subscripts`	Warnings from array subscripts with type char
`-W[no-]comment`	Warnings from suspicious comments
`-W[no-]div-by-zero`	Warnings from compile-time integer division by zero.
`-Wformat`	Warnings from inappropriate `printf()` arguments
`-Wimplicit`	Warnings implied by both `-Wimplicit-int` and `-Wimplicit-function-declaration`
`-Wimplicit-function-declaration`	Warnings from use of undeclared function
`-Wimplicit-int`	Warnings from declarations not specifying a type
`-Wmain`	Warnings from unusual main definition
`-Wmissing-braces`	Warnings from missing braces
`-Wno-multichar`	Warnings from multi-character constant
`-Wparentheses`	Warnings from missing precedence
`-Wreturn-type`	Warnings from missing return type
`-Wsequence-point`	Warnings from sequence point violations
`-Wswitch`	Warnings from missing or extraneous case values
`-Wsystem-headers`	Warnings from code within system headers
`-Wtrigraphs`	Warnings from use of trigraphs
`-Wuninitialized`	Warnings from use of uninitialized variables
`-Wunknown-pragmas`	Warnings from use of unknown pragma
`-Wunused`	Warnings from unused objects and constructs
`-Wunused-function`	Warnings from unused static function
`-Wunused-label`	Warnings from unused labels
`-Wunused-parameter`	Warnings from unused parameter

...........continued	
Option	**Controls**
`-Wunused-variable`	Warnings from unused variable
`-Wunused-value`	Warnings from unused value

Table 3-12. Warning Options Not Implied by All Warnings

Option	Controls
`-Wextra`	The generation of additional warning messages
`-Waggregate-return`	Warnings from aggregate objects being returned
`-Wbad-function-cast`	Warnings from functions cast to a non-matching type
`-Wcast-qual`	Warnings from discarded pointer qualifiers
`-Wconversion`	Warnings from implicit conversions that can alter values
`-Werror`	Generation of errors instead of warnings for dubious constructs
`-Winline`	Warnings when functions cannot be in-lined
`-Wlarger-than=`*`len`*	Warnings when defining large objects
`-W[no-]long-long`	Warnings from use of `long long`
`-Wmissing-declarations`	Warnings when functions are not declared
`-Wmissing-format-attribute`	Warnings with missing format attributes
`-Wmissing-noreturn`	Warnings from potential noreturn attribute omissions
`-Wmissing-prototypes`	Warnings when functions are not declared with prototype
`-Wnested-externs`	Warnings from `extern` declarations
`-Wno-deprecated-declarations`	Whether warnings are produced for deprecated declarations
`-Wpointer-arith`	Warnings when taking size of unsized types
`-Wredundant-decls`	Warnings from redundant declarations
`-Wshadow`	Warnings when local objects shadow other objects
`-W[no-]sign-compare`	Warnings from signed comparisons
`-Wstrict-prototypes`	Warnings from K&R function declarations
`-Wtraditional`	Warnings from traditional differences
`-Wundef`	Warnings from undefined identifiers
`-Wunreachable-code`	Warnings from unreachable code
`-Wwrite-strings`	Warnings when using non-const string pointers

3.6.4.1 Pedantic Option

The `-pedantic` option ensures that programs do not use forbidden extensions and that warnings are issued when a program does not follow ISO C.

3.6.4.2 Pedantic-errors Option

The `-pedantic-errors` option works in the same way as the `-pedantic` option, only errors, instead of warnings, are issued when a program is not ISO compliant.

3.6.4.3 Syntax-only Option

The `-fsyntax-only` option checks the C source code for syntax errors, then terminates the compilation.

3.6.4.4 W: Disable all Warnings Option

The `-w` option inhibits all warning messages, and thus should be used with caution.

3.6.4.5 Wall Option

The `-Wall` option enables all the warnings about easily avoidable constructions that some users consider questionable, even in conjunction with macros.

Note that some warning flags are not implied by `-Wall`. Of these warnings, some relate to constructions that users generally do not consider questionable, but which you might occasionally wish to check. Others warn about constructions that are necessary or hard to avoid in some cases and there is no simple way to modify the code to suppress the warning. Some of these warnings can be enabled using the `-Wextra` option, but many of them must be enabled individually.

3.6.4.6 Extra Option

The `-Wextra` option generates extra warnings in the following situations.

- A nonvolatile automatic variable might be changed by a call to longjmp. These warnings are possible only in optimizing compilation. The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called. In fact, a signal handler could call it at any point in the code. As a result, a warning may be generated even when there is in fact no problem, because `longjmp` cannot in fact be called at the place that would cause a problem.
- A function could exit both via `return` *`value`*`;` and `return;`. Completing the function body without passing any return statement is treated as `return;`.
- An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to `void`. For example, an expression such as `x[i,j]` causes a warning, but `x[(void)i,j]` does not.
- An unsigned value is compared against zero with `<` or `<=`.
- A comparison like `x<=y<=z` appears. This is equivalent to `(x<=y ? 1 : 0) <= z`, which is a different interpretation from that of an ordinary mathematical notation.
- Storage-class specifiers like `static` are not the first things in a declaration. According to the C Standard, this usage is obsolescent.
- If `-Wall` or `-Wunused` is also specified, warn about unused arguments.
- A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned (but won't warn if `-Wno-sign-compare` is also specified).
- An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for `x.h`:

```
struct s { int f, g; };
struct t { struct s h; int i; };
struct t x = { 1, 2, 3 };
```

- An aggregate has an initializer that does not initialize all members. For example, the following code would cause such a warning, because `x.h` would be implicitly initialized to zero:

```
struct s { int f, g, h; };
struct s x = { 3, 4 };
```

3.6.5 Options for Debugging

The options tabulated below control the debugging output produced by the compiler and are discussed in the sections that follow.

Table 3-13. Debugging Options

Option	Controls
`-mcodecov`	Instrument the output for code coverage

...........continued	
Option	**Controls**
`-g`	The type of debugging information generated
`-Q`	Printing of diagnostics associated with each function as it is compiled, and statistics about each pass on conclusion.
`-save-temps` `-save-temps=dir`	Whether and where intermediate files should be kept after compilation

3.6.5.1 Codecov Option

The `-mcodecov=suboptions` option embeds diagnostic code into the program's output, allowing analysis of the extent to which the program's source code has been executed. See 4.2.8 Code Coverage for more information.

A suboption must be specified and at this time, the only available suboption is `ram`.

3.6.5.2 G: Produce Debugging Information Option

The `-gformat` option instructs the compiler to produce additional information, which can be used by hardware tools to debug your program.

The support formats are tabulated below.

Table 3-14. Supported Debugging File Formats

Format	Debugging file format
`-gcoff`	COFF
`-gdwarf-3`	ELF/Dwarf release 3

By default, the compiler produces Dwarf release 2 files.

The compiler supports the use of this option with the optimizers enabled, making it possible to debug optimized code; however, the shortcuts taken by optimized code may occasionally produce surprising results, such as variables that do not exist and flow control that changes unexpectedly.

3.6.5.3 Q: Print Function Information Option

The `-Q` option instructs the compiler to print out each function name as it is compiled, and print statistics about each pass when it finishes.

3.6.5.4 Save-temps Option

The `-save-temps` option instructs the compiler to keep temporary files after compilation has finished.

The intermediate files will be placed in the current directory and have a name based on the corresponding source file. Thus, compiling `foo.c` with `-save-temps` would produce `foo.i`, `foo.s` and the `foo.o` object file.

The `-save-temps=cwd` option is equivalent to `-save-temps`.

The `-save-temps=obj` form of this option is similar to `-save-temps`, but if the `-o` option is specified, the temporary files are placed in the same directory as the output object file. If the `-o` option is not specified, the `-save-temps=obj` switch behaves like `-save-temps`.

The following example will create `dir/xbar.i`, `dir/xbar.s`, since the `-o` option was used.

```
xc8-gcc -save-temps=obj -c bar.c -o dir/xbar.o
```

3.6.6 Options for Controlling Optimization

The options tabulated below control compiler optimizations and are described in the sections that follow.

Table 3-15. General Optimization Options

Option	Edition	Builds with
`-O0`	All	No optimizations (default)
`-O` `-O1`	All	Optimization level 1
`-O2`	PRO only	Optimization level 2
`-O3`	PRO only	Speed-orientated Optimizations
`-Og`	All	Better debugging
`-Os`	PRO only	Size-orientated optimizations
`-flto`	PRO only	The standard link-time optimizer
`-fwhole-program`	PRO only	The whole-program optimizations
`-fuse-linker-plugin`	PRO	A linker plugin with link-time optimizations

3.6.6.1 O0: Level 0 Optimizations

The `-O0` option performs only rudimentary optimization. This is the default optimization level if no `-O` option is specified.

With this optimization level selected, the compiler’s goal is to reduce the cost of compilation and to make debugging produce the expected results.

Statements are independent when compiling with this optimization level. If you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

The compiler only allocates variables declared `register` in registers.

3.6.6.2 O1: Level 1 Optimizations

The `-O1` or `-O` options request level 1 optimizations.

The optimizations performed when using `-O1` aims to reduce code size and execution time, but still allows a reasonable level of debugability.

This level is available for unlicensed as well as licensed compilers.

3.6.6.3 O2: Level 2 Optimizations Option

The `-O2` option requests level 2 optimizations.

At this level, the compiler performs nearly all supported optimizations that do not involve a space-speed trade-off.

This level is available for unlicensed as well as licensed compilers.

3.6.6.4 O3: Level 3 Optimizations Option

The `-O3` option requests level 3 optimizations.

This option requests all supported optimizations that reduces execution time but which might increase program size.

This level is available only for licensed compilers.

3.6.6.5 Og: Better Debugging Option

The `-Og` option disables optimizations that severely interfere with debugging, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.

3.6.6.6 Os: Level s Optimizations Option

The `-Os` option requests space-orientated optimizations.

This option requests all supported optimizations that do not typically increase code size.

It also performs further optimizations designed to reduce code size, but which might slow program execution, such as procedural abstraction optimizations.

This level is available only for licensed compilers.

3.6.6.7 Lto Option

This `-flto` option runs the standard link-time optimizer.

When invoked with source code, the compiler adds an internal bytecode representation of the code to special sections in the object file. When the object files are linked together, all the function bodies are read from these sections and instantiated as if they had been part of the same translation unit.

To use the link-timer optimizer, specify `-flto` both at compile time and during the final link. For example

```
xc8-cc -c -O1 -flto -mcpu=atmega3290p foo.c
xc8-cc -c -O1 -flto -mcpu=atmega3290p bar.c
xc8-cc -o myprog.elf -flto -O3 -mcpu=atmega3290p foo.o bar.o
```

Another (simpler) way to enable link-time optimization is,

```
xc8-cc -o myprog.elf -flto -O3 -mcpu=atmega3290p foo.c bar.c
```

3.6.6.8 Whole-program Optimizations Option

The `-fwhole-program` option runs more aggressive interprocedural optimizations.

The option assumes that the current compilation unit represents the whole program being compiled. All public functions and variables, with the exception of `main()` and those merged by attribute `externally_visible`, become `static` functions and in effect are optimized more aggressively by interprocedural optimizers. While this option is equivalent to proper use of the `static` keyword for programs consisting of a single file, in combination with option `-flto`, this flag can be used to compile many smaller scale programs since the functions and variables become local for the whole combined compilation unit, not for the single source file itself.

Whole-program optimizations should not be used with the `-fuse-linker-plugin` link time optimizations option.

3.6.6.9 Use-linker-plugin Option

The `-fuse-linker-plugin` option enables the use of a linker plugin during link-time optimization, improving the quality of optimization by exposing more code to the link-time optimizer. The `-fwhole-program` option should not be used with the `-fuse-linker-plugin` option.

3.6.7 Options for Controlling the Preprocessor

The options tabulated below control the preprocessor and are discussed in the sections that follow.

Table 3-16. Preprocessor Options

Option	Controls
`-C`	Preserve comments
`-dD`	Preserve macro definitions
`-Dmacro` `-Dmacro=`*defn*	Define a macro
`-dM`	Output macro definition list
`-dN`	Preserve macro names
`-fno-show-column`	Omit column numbers in diagnostics
`-H`	Print header file name
`-imacros` *file*	Include file macro definitions only
`-include` *file*	Include file

...........continued	
Option	**Controls**
`-iquote`	Specify quoted include file search path
`-M`	Generate make rule
`-MD`	Write dependency information to file
`-MF` *`file`*	Specify dependency file
`-MG`	Ignore missing header files
`-MM`	Generate make rule for quoted headers
`-MMD`	Generate make rule for user headers
`-MP`	Add phony target for dependency
`-MQ`	Change rule target with quotes
`-MT` *`target`*	Change rule target
`-nostdinc`	System directories omitted from header search
`-P`	Don't generate #line directives
`-trigraphs`	Support trigraphs
`-U`*`macro`*	Undefine macros
`-undef`	Do not predefine nonstandard macros

3.6.7.1 C: Preserve Comments Option

The `-C` option tells the preprocessor not to discard comments from the output. Use this option with the `-E` option to see commented yet preprocessed source code.

3.6.7.2 d: Preprocessor Debugging Dumps Option

The `-d`*`letters`* option has the preprocessor produce debugging dumps during compilation as specified by *`letters`*. This option should be used in conjunction with the `-E` option.

The acceptable letter arguments to `-d` are tabulated below. Other arguments are silently ignored.

Table 3-17. Preprocessor Debugging Information

Letter	Produces
`D`	Full macro definitions
`M`	Only those macro definitions that are in effect at the end of preprocessing
`N`	Macro definitions without arguments and contents

3.6.7.3 D: Define a Macro

The `-D`*`macro`* option allows you to define a preprocessor macro and the `-D`*`macro`*`=`*`text`* form of this option additionally allows a user-define replacement string to be specified with the macro. A space may be present between the option and macro name.

When no replacement text follows the macro name, the `-D`*`macro`* option defines a preprocessor macro called *`macro`* and specifies its replacement text as `1`. Its use is the equivalent of placing `#define` *`macro`* `1` at the top of each module being compiled.

The `-D`*`macro`*`=`*`text`* form of this option defines a preprocessor macro called *`macro`* with the replacement text specified. Its use is the equivalent of placing `#define` *`macro text`* at the top of each module being compiled.

Either form of this option creates an identifier (the macro name) whose definition can be checked by `#ifdef` or `#ifndef` directives. For example, when using the option, `-DMY_MACRO` (or `-D MY_MACRO`) and building the following code:

```
#ifdef MY_MACRO
int input = MY_MACRO;
#endif
```

the definition of the `int` variable `input` will be compiled, and the variable assigned the value `1`.

If the above example code was instead compiled with the option `-DMY_MACRO=0x100`, then the variable definition that would ultimately be compiled would be: `int input = 0x100;`

See 4.13.1.1 Preprocessor Arithmetic for clarification of how the replacement text might be used.

Defining macros as C string literals requires bypassing any interpretation issues in the operating system that is being used. To pass the C string, `"hello world"`, (including the quote characters) in the Windows environment, use: `"-DMY_STRING=\\\"hello world\\\""` (you must include the quote characters around the entire option, as there is a space character in the replacement textu). Under Linux or Mac OS X, use: `-DMY_STRING=\"hello\ world\"`.

All instances of `-D` on the command line are processed before any `-U` options.

3.6.7.4 H: Print Header Files Option

The `-H` option prints to the console the name of each header file used, in addition to other normal activities.

3.6.7.5 Imacros Option

The `-imacros` *`file`* option processes the specified file in the same way the `-include` option would, except that any output produced by scanning the file is thrown away. The macros that the file defines remain defined during processing. Because the output generated from the file is discarded, the only effect of this option is to make the macros defined in file available for use in the main input.

Any `-D` and `-U` options on the command line are always processed before an `-imacros` option, regardless of the order in which they are placed. All the `-include` and `-imacros` options are processed in the order in which they are written.

3.6.7.6 Include Option

The `-include` *`file`* option processes *`file`* as if `#include "file"` appeared as the first line of the primary source file. In effect, the contents of *`file`* are compiled first. Any `-D` and `-U` options on the command line are always processed before the `-include` option, regardless of the order in which they are written. All the `-include` and `-imacros` options are processed in the order in which they are written.

3.6.7.7 Iquote Option

The `-iquote` *`dir`* option adds the directory *`dir`* to the list of directories to be searched for header files during preprocessing. Directories specified with this option apply only to the quoted form of the directive, for example `#include "file"`.

3.6.7.8 M: Generate Make Rule

The `-M` option tells the preprocessor to output a rule suitable for `make` that describes the dependencies of each object file.

For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the header files it includes. This rule may be a single line or may be continued with a backslash-newline sequence if it is lengthy.

The list of rules is printed on standard output instead of the preprocessed C program.

The `-M` option implies `-E`.

3.6.7.9 MD: Write Dependency Information To File Option

The `-MD` option writes dependency information to a file.

This option is similar to `-M` but the dependency information is written to a file and compilation continues. The file containing the dependency information is given the same name as the source file with a `.d` extension.

3.6.7.10 MG: Ignore Missing Header Files Option

The `-MG` option treats missing header files as generated files and adds them to the dependency list without raising an error.

The option assumes the missing files live in the same directory as the source file. If `-MG` is specified, then either `-M` or `-MM` must also be specified. This option is not supported with `-MD` or `-MMD`.

3.6.7.11 MM: Generate Make Rule For Quoted Headers Option

The `-MM` option performs the same tasks as `-M`, but system headers are not included in the output.

3.6.7.12 MMD: Generate Make Rule For User Headers Option

The `-MMD` option performs the same tasks as `-MD`, but only user header files are included in the output.

3.6.7.13 MP: Add Phony Target For Dependency Option

The `-MP` option instructs the preprocessor to add a phony target for each dependency other than the main file, causing each to depend on nothing. These dummy rules work around make errors if you remove header files without updating the make-file to match.

This is typical output:

```
test.o: test.c test.h
test.h:
```

3.6.7.14 MQ: Change Rule Target With Quotes Option

The `-MQ` option is similar to `-MT`, but it quotes any characters which are considered special by `make`.

```
-MQ '$(objpfx)foo.o' gives $$(objpfx)foo.o: foo.c
```

The default target is automatically quoted, as if it were given with `-MQ`.

3.6.7.15 MT: Change Rule Target Option

The `-MT` target option changes the target of the rule emitted by dependency generation.

By default, the preprocessor takes the name of the main input file, including any path, deletes any file suffix such as `.c`, and appends the platform's usual object suffix. The result is the target. An `-MT` option sets the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to `-MT`, or use multiple `-MT` options.

For example:

```
-MT '$(objpfx)foo.o' might give $(objpfx)foo.o: foo.c
```

3.6.7.16 No-show-column Option

The `-fno-show-column` option controls whether column numbers will be printed in diagnostics.

This option may be necessary if diagnostics are being scanned by a program that does not understand the column numbers, such as DejaGnu.

3.6.7.17 Nostdinc Option

The `-nostdinc` option prevents the standard system directories for header files being searched by the preprocessor. Only the directories you have specified with `-I` options (and the current directory, if appropriate) are searched.

By using both `-nostdinc` and `-iquote`, the include-file search path can be limited to only those directories explicitly specified.

3.6.7.18 Trigraphs Option

The `-trigraphs` option turns on support for ANSI C trigraphs. The `-ansi` option also has this effect.

3.6.7.19 U: Undefine Macros

The `-U`*`macro`* option undefines the macro `macro`.

Any builtin macro or macro defined using `-D` will be undefined by the option. All `-U` options are evaluated after all `-D` options, but before any `-include` and `-imacros` options.

3.6.7.20 Undef Option

The `-undef` option prevents any system-specific or GCC-specific macros being predefined (including architecture flags).

3.6.8 Options for Assembling

The options tabulated below control assembler operations and are discussed in the sections that follow.

Table 3-18. Assembly Options

Option	Controls
`-Wa,`*`option`*	Options to passed to the assembler.
`-Xassembler` *`option`*	Options to passed to the assembler.

3.6.8.1 Wa: Pass Option To The Assembler Option

The `-Wa,`*`option`* option passes its option argument directly to the assembler. If `option` contains commas, it is split into multiple options at the commas. For example `-Wa,-a` will request that the assembler produce an assembly list file.

3.6.8.2 Xassembler Option

The `-Xassembler` *`option`* option passes *`option`* to the assembler, where it will be interpreted as an assembler option. You can use this to supply system-specific assembler options that the compiler does not know how to recognize.

3.6.9 Mapped Assembler Options

The option tabulated below is a commonly used assembler option.

Table 3-19. Mapped Assembler Options

Option	Controls
`-Wa,-a`	The generation of an assembly list file

3.6.10 Options for Linking

The options tabulated below control linker operations and are discussed in the sections that follow. If any of the options `-c`, `-S` or `-E` are used, the linker is not run.

Table 3-20. Linking Options

Option	Controls
`-l`*`library`*	Which library files are scanned
`-nodefaultlibs`	Whether library code is linked with the project
`-nostartfiles`	Whether the runtime startup module is linked in
`-nostdlib`	Whether the library and startup code is linked with the project
`-s`	Remove all symbol table and relocation information from the executable.
`-u` *`symbol`*	The linking in of library modules so that symbol can be defined. It is legitimate to use -u multiple times with different symbols to force loading of additional library modules.
`-Wl,`*`option`*	Options passed to the linker.
`-Xlinker` *`option`*	System-specific options to passed to the linker

3.6.10.1 L: Specify Library File Option

The `-llibrary` option scans the library named library for unresolved symbols when linking.

When this option is used, the linker will search a standard list of directories for the library with the name `library.a`. The directories searched include the standard system directories, plus any that you specify with the `-L` option.

The linker processes libraries and object files in the order they are specified, so it makes a difference where you place this option in the command. The options (and in this order), `foo.o -llibz bar.o` search library `libz.a` after file `foo.o` but before `bar.o`. If `bar.o` refers to functions in `libz.a`, those functions may not be loaded.

Normally the files found this way are library files (archive files whose members are object files). The linker handles an archive file by scanning through it for members which define symbols that have been referenced but not defined yet. But if the file found is an ordinary object file, it is linked in the usual fashion.

The only difference between using an `-l` option (e.g., `-lmylib`) and specifying a file name (e.g., `mylib.a`) is that the compiler will search for a library specified using `-l` in several directories, as specified by the `-L` option.

3.6.10.2 Nodefaultlibs Option

The `-nodefaultlibs` option will prevent the standard system libraries being linked into the project. Only the libraries you specify are passed to the linker.

3.6.10.3 Nostartfiles Option

The `-nostartfiles` option will prevent the runtime startup modules from being linked into the project.

3.6.10.4 Nostdlib Option

The `-nostdlib` option will prevent the standard system startup files and libraries being linked into the project. No startup files and only the libraries you specify are passed to the linker.

The compiler may generate calls to `memcmp`, `memset` and `memcpy`. These entries are usually resolved by entries in standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.

3.6.10.5 S: Remove Symbol Information Option

The `-s` option removes all symbol table and relocation information from the output.

3.6.10.6 WL: Linker Option

The `-Wl,option` option passes *option* to the linker where it will be interpreted as a linker option. If *option* contains commas, it is split into multiple options at the commas.

3.6.10.7 Xlinker Option

The `-Xlinker option` option pass *option* to the linker where it will be interpreted as a linker option. You can use this to supply system-specific linker options that the compiler does not know how to recognize.

3.6.11 Mapped Linker Options

The options tabulated below are commonly used linker options.

Table 3-21. Mapped Linker Options

Option	Controls
`-Wl,-Map=mapfile`	The generation of a linker map file

3.6.12 Options for Directory Search

The options tabulated below control directories searched operations and are discussed in the sections that follow.

Table 3-22. Directory Search Options

Option	Controls
`-idirafter dir`	Additional directories searched for headers after searching system paths
`-Idir`	The directories searched for preprocessor include files

..........continued	
Option	**Controls**
`-Ldir`	The directories searched for libraries
`-nostdinc`	The directories searched for headers

3.6.12.1 Idirafter Option

The `-idirafter dir` option adds the directory `dir` to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path, including those specified by `-I`.

3.6.12.2 I: Specify Include File Search Path Option

The `-Idir` option adds the directory `dir` to the head of the list of directories to be searched for header files. A space may be present between the option and directory name.

The option can specify either an absolute or relative path and it can be used more than once if multiple additional directories are to be searched, in which case they are scanned from left to right. The standard system directories are searched after scanning the directories specified with this option.

Under the Windows OS, the use of the directory backslash character may unintentionally form an escape sequence. To specify an include file path that ends with a directory separator character and which is quoted, use `-I "E:\\"`, for example, instead of `-I "E:\"`, to avoid the escape sequence `\"`. Note that MPLAB X IDE will quote any include file path you specify in the project properties and that search paths are relative to the output directory, not the project directory.

3.6.12.3 L: Specify Library Search Path Option

The `-Ldir` option allows you to specify an additional directory to be searched for library files that have been specified by using the `-l` option. The compiler will automatically search standard library locations, so you only need to use this option if you are linking in your own libraries.

3.6.12.4 Nostdinc Option

The `-nostdinc` option prevents the standard system directories for header files being searched by the preprocessor. Only the directories you have specified with `-I` options (and the current directory, if appropriate) are searched.

By using both `-nostdinc` and `-iquote`, the include-file search path can be limited to only those directories explicitly specified.

3.6.13 Options for Code Generation Conventions

The options tabulated below control machine-independent conventions used when generating code and are discussed in the sections that follow.

Table 3-23. Code Generation Convention Options

Option	Controls
`-fshort-enums`	The size of `enum` types

3.6.13.1 Short-enums Option

The `-fshort-enums` option allocates the smallest possible integer type to an `enum` such that the range of possible values can be held. Use of this option generates code that is not binary compatible with code generated without the option.

3.7 MPLAB X IDE Integration

The 8-bit language tools may be integrated into and controlled from the MPLAB X IDE, to provide a GUI-based development of application code for the 8-bit AVR MCU families of devices.

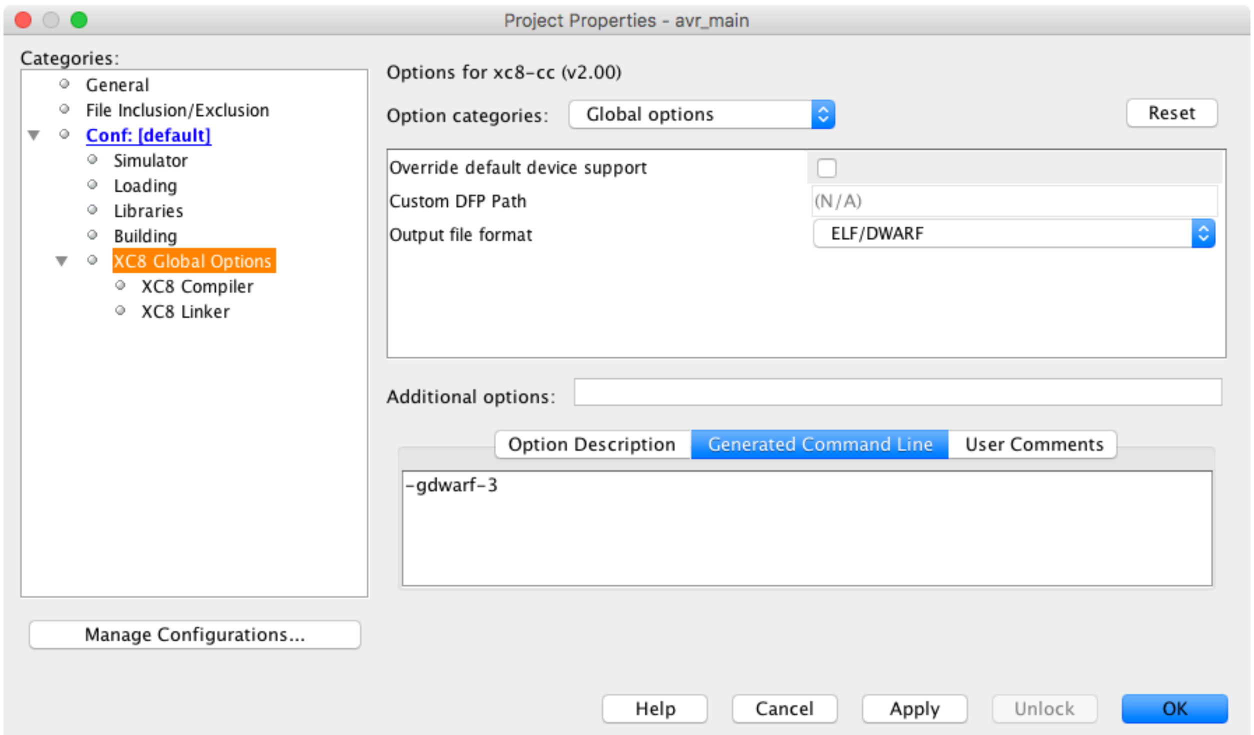
For installation of the IDE, and the creation and setup of projects to use the MPLAB XC8 C Compiler, see the *MPLAB® X IDE User's Guide*.

3.7.1 MPLAB X IDE Option Equivalents

The following descriptions map the MPLAB X IDE's **Project Properties** controls to the MPLAB XC8 command-line driver options. Reference is given to the relevant section in the user's guide to learn more about the option's function. In the IDE, click any option to see online help and examples shown in the **Option Description** field in the lower part of the dialog.

3.7.1.1 XC8 Global Options - Global Options

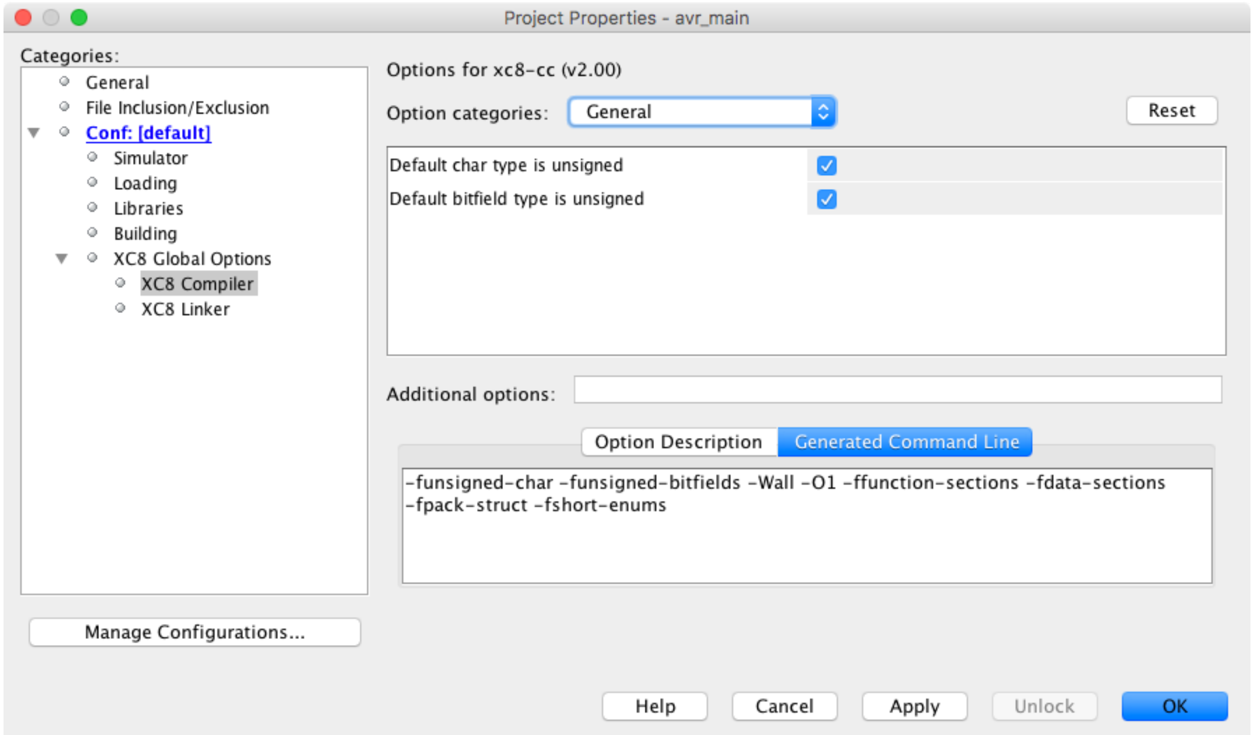
Figure 3-2. XC8 Global Options - Global Options



Override default device support	Selecting this checkbox enables the **Custom DFP Path** field below, which allows you to specify an alternate device family pack location. The DFPs provide device-specific information, such as register names and addresses, which can be used by code in the project.
Output file format	This selector specifies the output file type. See 3.6.5.2 G: Produce Debugging Information Option.

3.7.1.2 XC8 Compiler - General Options

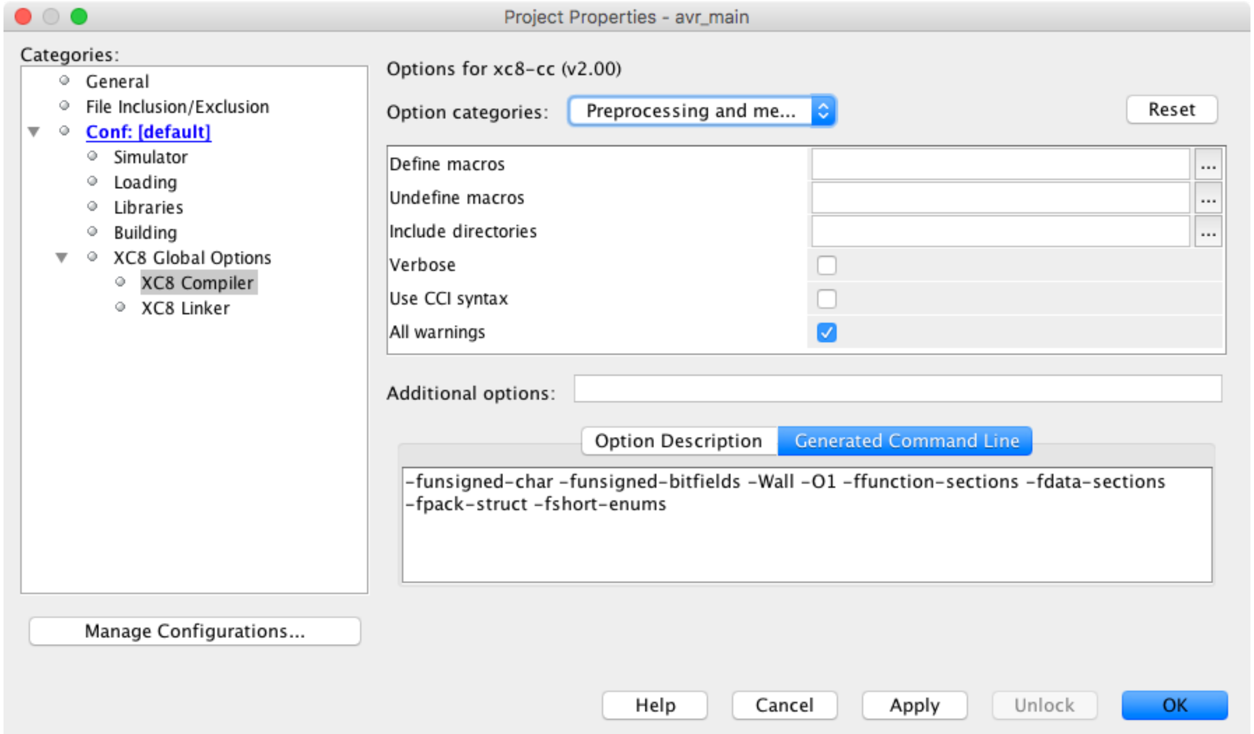
Figure 3-3. XC8 Compiler - General Options



Default char type is unsigned	Enabling this checkbox indicates that a plain `char` type will be treated as an `unsigned char`. See 3.6.3.7 Signed-char Option and 3.6.3.10 Unsigned-char Option.
Default bitfield type is unsigned	Enabling this checkbox indicates that a plain bit-field will be treated as an unsigned object. See 3.6.3.6 Signed-bitfields Option and 3.6.3.9 Unsigned-bitfields Option.

3.7.1.3 XC8 Compiler - Preprocessing and Messaging Options

Figure 3-4. XC8 Compiler - Preprocessing and Messaging Options



Define macros This field allows you to define preprocessor macros. See 3.6.7.3 D: Define a Macro.

Undefine macros This field allows you to undefine preprocessor macros. See 3.6.7.19 U: Undefine Macros.

Include directories This field allows you to specify the directories searched for header files. See 3.6.12.2 I: Specify Include File Search Path Option.

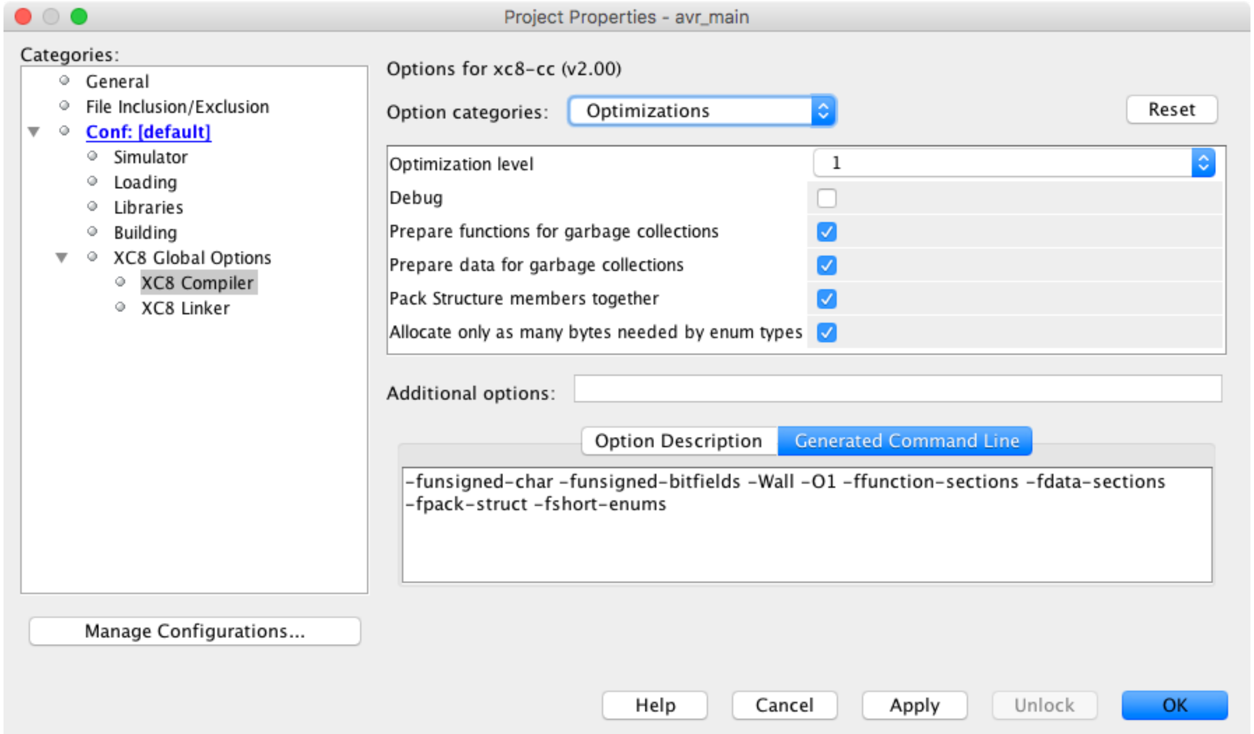
Verbose Enabling this checkbox shows the command lines used to build the project. See 3.6.2.6 V: Verbose Compilation.

Use CCI syntax Enabling this checkbox requests that the CCI language extension be enforced. See 3.6.3.3 Ext Option.

All warnings Enabling this checkbox requests that no compiler warning messages will be suppressed when building. See 3.6.4.5 Wall Option.

3.7.1.4 XC8 Compiler - Optimizations options

Figure 3-5. XC8 Compiler - Optimization Options



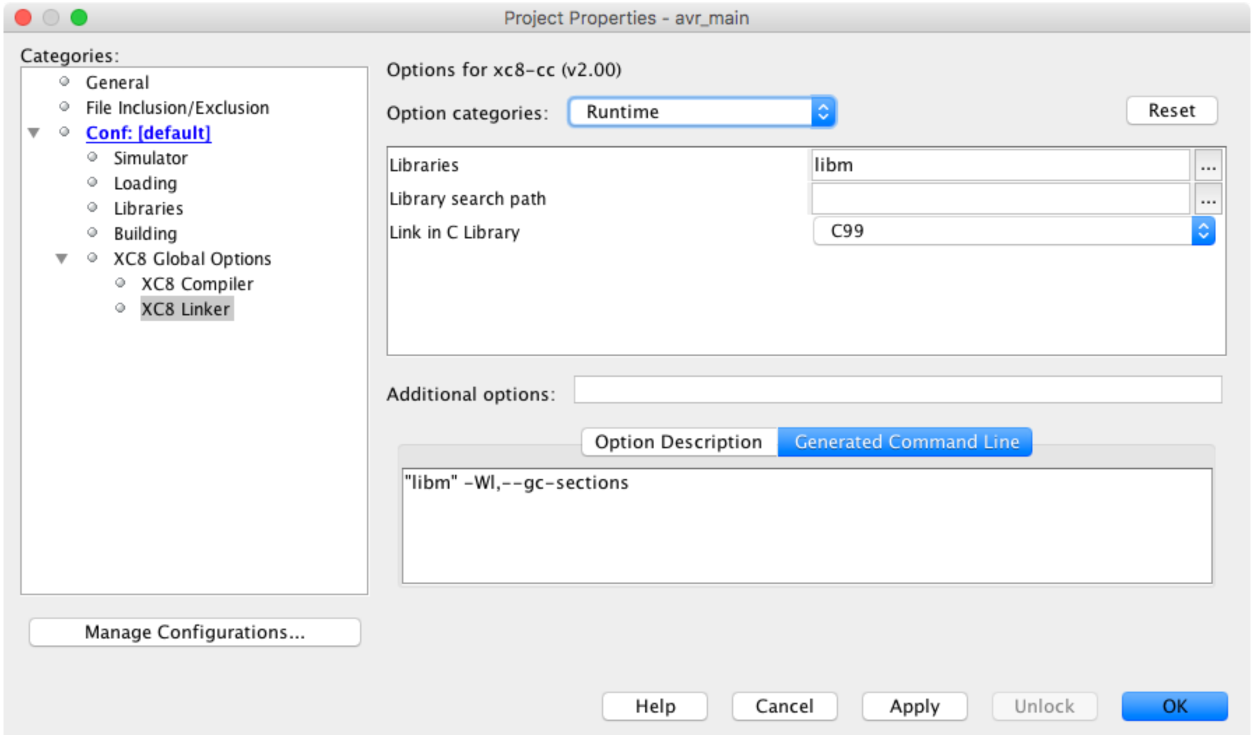
Optimization level This selector controls the level to which programs are optimized. See 3.6.6 Options for Controlling Optimization.

Debug This checkbox inhibits aggressive optimization that can impact on the debugability of code. See 3.6.6.5 Og: Better Debugging Option.

Other optimizations The remainder of the selectors in this dialog control specific code generation features which can optimize the generated code.

3.7.1.5 XC8 Linker - Runtime Options

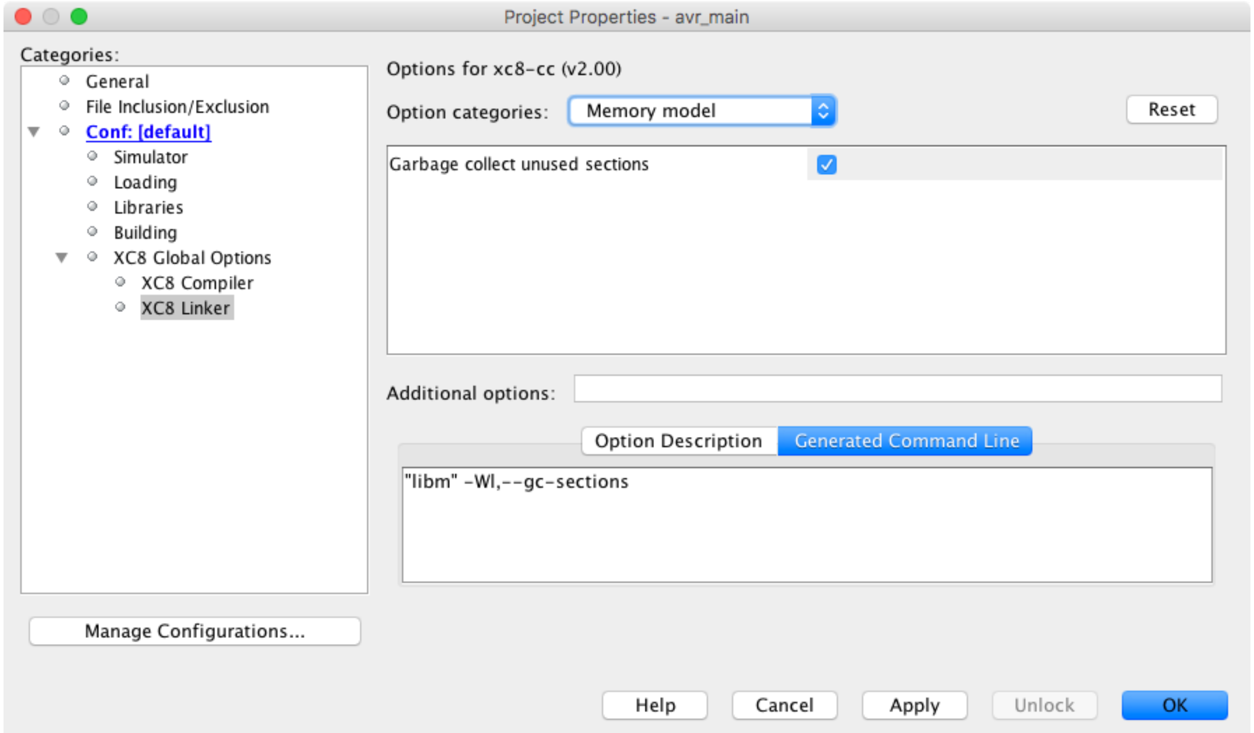
Figure 3-6. XC8 Linker - Runtime Options



Libraries	This field allows you to specify the names of additional libraries to link.
Library search path	This field lets you specify the search paths used by the compiler to find library files. See 3.6.12.3 L: Specify Library Search Path Option.
Link in C library	This selector specifies whether the standard libraries will be linked. See 3.6.10.2 Nodefaultlibs Option.

3.7.1.6 XC8 Linker - Memory Model Options

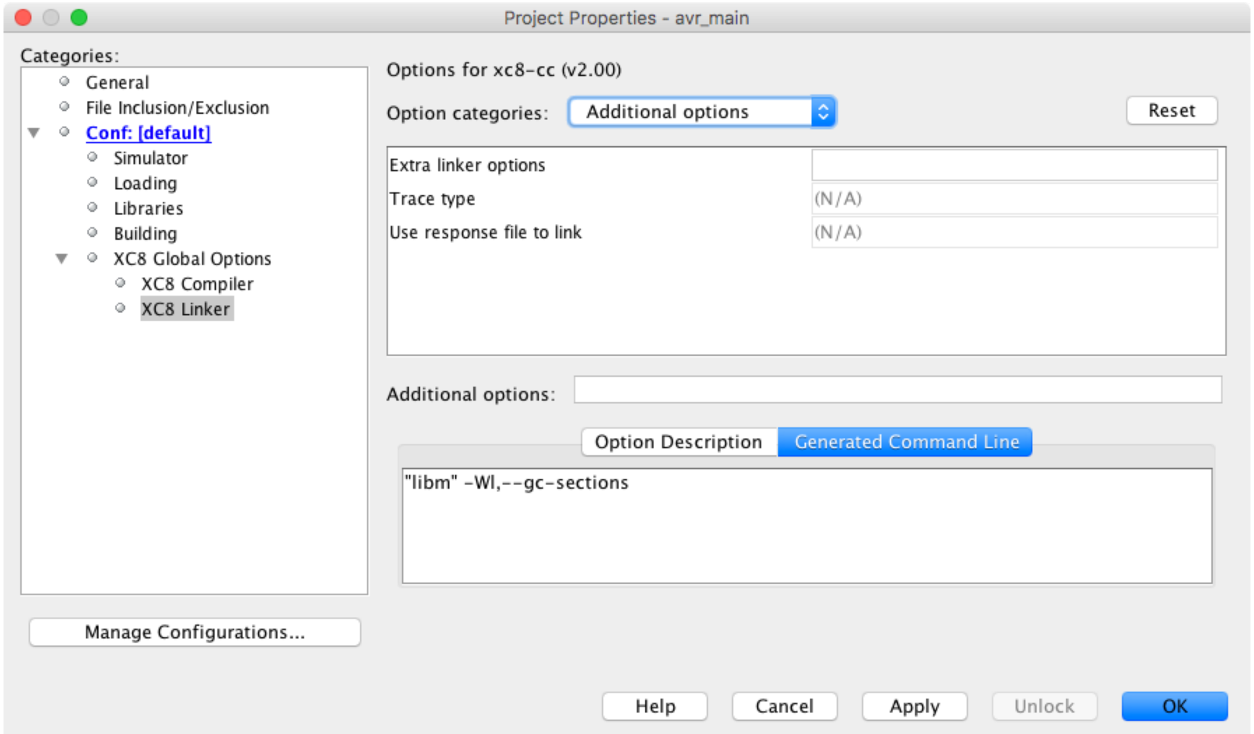
Figure 3-7. XC8 Linker - Memory model options



Garbage collect unused sections	This selector specifies if the linker should remove sections that it considers are not used.

3.7.1.7 XC8 Linker - Additional options

Figure 3-8. XC8 Linker - Additional options



Extra linker options This field allows you to specify additional linker-related options that cannot be otherwise controlled from the IDE. See 3.6.10.6 WL: Linker Option.

Trace type This selector is not yet implemented.

Use response file to link This field allows a command-line options file to be used by the compiler during the link step, in preference to the other link-step settings in the project properties. See 3.1.1.1 Long Command Lines. This option is only relevant when running MPLAB X IDE under Windows.

4. C Language Features

MPLAB XC8 C Compiler supports a number of special features and extensions to the C language that are designed to ease the task of producing ROM-based applications for 8-bit AVR devices. This chapter documents the special language features that are specific to these devices.

4.1 C Standard Compliance

This compiler is a freestanding implementation that conforms to the ISO/IEC 9899:1990 Standard (referred to as the C90 standard) as well the ISO/IEC 9899:1999 Standard (C99) for programming languages. The program standard can be selected using the `-std` option (see 3.6.3.8 Std Option).

This implementation makes no assumptions about any underlying operating system and does not provide support for streams, files, or threads. Aspects of the compiler that diverge from the standards are discussed in this section.

4.1.1 Common C Interface Standard

This compiler conforms to the Microchip XC compiler Common C Interface standard (CCI) and can verify that C source code is compliant with CCI.

CCI is a further refinement of the C standards that attempts to standardize implementation-defined behavior and non-standard extensions across the entire MPLAB XC compiler family.

CCI can be enforced by using the `-mext=cci` option (see 3.6.3.3 Ext Option).

4.1.2 Divergence From the C99 Standard

The C language implemented on MPLAB XC8 C Compiler diverges from the C99 Standard, as described in the following sections.

4.1.2.1 Complex Number Support

The complex type `_Imaginary` is not supported (although the use of `_Complex` is permitted). The `<complex.h>` header is also not supported.

4.1.3 Implementation-Defined Behavior

Certain features of the ISO C standard have implementation-defined behavior. This means that the exact behavior of some C code can vary from compiler to compiler. The exact behavior of the compiler is detailed throughout this manual and is fully summarized in 6. Implementation-Defined Behavior.

4.2 Device-Related Features

MPLAB XC8 has several features which relate directly to the 8-bit PIC architectures and instruction sets. These are detailed in the following sections.

4.2.1 Device Support

The MPLAB XC8 C Compiler aims to support all 8-bit PIC and AVR devices (excluding only the avr1 architecture devices, which must be programmed in assembly). This user's guide should be consulted when you are programming an 8-bit AVR device; when programming a PIC target, see the *MPLAB® XC8 C Compiler User's Guide for PIC® MCU* (DS50002737).

New AVR devices are frequently released. There are several ways you can check whether the compiler you are using supports a particular device.

From the command line, run the compiler you wish to use and pass it the option `-mprint-devices` (See 3.6.2.4 Print-devices). A list of all devices will be printed.

You can also see the supported devices in your favorite web browser. Open the file `avr_chipinfo.html` for a list of all supported devices. This file is located in the docs directory under your compiler's installation directory.

You may expand the list of devices supported by your compiler by downloading and installing Device Family Packs (DFPs) as they are released. This will usually be managed by the MPLAB X IDE.

4.2.2 Instruction Set Support

The compiler support all instruction sets for all 8-bit AVR devices, excluding that for the avr1 architecture.

4.2.3 Stacks

There is one stack implemented by MPLAB XC8. This stack is used for both function return addresses and stack-based objects allocated by functions. The registers r28 and r29 (Y pointer) act as a frame pointer from which stack-based objects can be accessed.

The stack pointer is initialized to the highest valid data memory address. As functions are called, they allocate a chunk of memory for the stack-based objects and the stack grows down in memory, towards smaller addresses. When the function exits, the memory it claimed is made available to other functions.

Note that the compiler cannot detect for overflow of the memory reserved for the stack as a whole. There is no runtime check made for software stack overflows. If the software stack overflows, data corruption and code failure might result.

4.2.4 Configuration Bit Access

Configuration bits or fuses are used to set up fundamental device operation, such as the oscillator mode, watchdog timer, programming mode, and lockbit code protection. These bits must be correctly set to ensure your program executes correctly.

Use the configuration pragma, which has the following form, to set up your device.

```
#pragma config setting = state|value
```

Here, *setting* is a configuration setting descriptor, e.g., `WDT`, and *state* is a textual description of the desired state, e.g., `SET`. Those states other than `SET` and `CLEAR` are prefixed with the setting descriptor, as in `BODLEVEL_4V3`, as shown in the following examples.

```
#pragma config WDTON = SET
#pragma config EESAVE = CLEAR
#pragma config BODLEVEL = BODLEVEL_4V3
#pragma config LB = LB_NO_LOCK
```

The settings and states associated with each device can be determined from an HTML guide. Open the `avr_chipinfo.html` file that is located in the `docs` directory of your compiler installation. Click the link to your target device and the page will show you the settings and values that are appropriate with this pragma. Review your device data sheet for more information.

One pragma can be used to program several settings by separating each setting-value pair with a comma. For example, the above could be specified with one pragma, as in the following.

```
#pragma config WDTON=SET, EESAVE=CLEAR, BODLEVEL=BODLEVEL_4V3, LB=LB_NO_LOCK
```

The *value* field is a constant that can be used in preference to a descriptor, as in the following.

```
#pragma config SUT_CKSEL = 0x10
```

Setting-value pairs are not scanned by the preprocessor and they are not subject to macro substitution. The setting-value pairs must not be placed in quotes.

The `config` pragma does not produce executable code, and ideally it should be placed outside function definitions.

Those bits not specified by a pragma are assigned a default value. Rather than rely on this default value, all the bits in the Configuration Words should be programmed to prevent erratic program behavior.

4.2.5 Signatures

A signature value can be used by programming software to verify the program was built for the intended device before it is programmed.

Signatures are specified with each device and can be added to your program by simply including the `<avr/signature.h>` header file. This header will declare a constant `unsigned char` array in your code and initialize it with the three signature bytes, MSB first, that are defined in the device's I/O header file. This array is then placed in the .signature section in the resulting linked ELF file. This header file should only be included once in an application.

The three signature bytes used to initialize the array are these defined macros in the device I/O header file, from MSB to LSB: `SIGNATURE_2`, `SIGNATURE_1`, `SIGNATURE_0`.

4.2.6 Using SFRs From C Code

The Special Function Registers (SFRs) are memory mapped registers that can be accessed from C programs. Each register can be accessed using a macro that is available once you include `<xc.h>`. For example:

```
#include <xc.h>
if(EEDR == 0x0)
  PORTA = 0x55;
```

Bits within SFRs can be accessed via a special macro, `_BV()`, and other macros which represent the bit you wish to access. For example, to set bit #1 in PORTB, use the following.

```
PORTB |= _BV(PB1);
```

To clear both bits #4 and #5 in EECR, use the following.

```
EECR &= ~(_BV(EEPM4) | _BV(EEPM5));
```

In both these examples, the compiler will use the device's single bit set and clear instructions whenever possible.

4.2.7 Special Register Issues

Some of the timer-related 16-bit registers internally use an 8-bit wide temporary register (called TEMP in the device data sheets) to guarantee atomic access to the timer, since two separate byte transfers are required to move timer values. Typically, this register is used by the device when accessing the current timer/counter value register (TCNTn), the input capture register (ICRn), and when writing the output compare registers (OCRnM). Refer to your device data sheet to determine which peripherals make use of the TEMP register.

This temporary register is not accessible to your program, but it is shared by many peripherals, thus your program needs to ensure that the register is not corrupted by interrupt routines that also uses this register.

Within main-line code, interrupts could be disabled during the execution of the code which utilizes this register. That can be done be encapsulating the code in calls to the `cli()` and `sei()` macros, but if the status of the global interrupt flag is not known, the following example code can be used.

```
unsigned int read_timer1(void)
{
  unsigned char sreg;
  unsigned int val;

  sreg = SREG; // save state of interrupt
  cli(); // disbale interrupts
  val = TCNT1; // read timer value register; TEMP used internally
  SREG = sreg; // restore state of interrupts

  return val;
}
```

4.2.8 Code Coverage

After purchase of a special license (SW006026-COV), the compiler's code coverage feature can be used to facilitate analysis of the extent to which a project's source code has been executed.

This feature is initially available for all 8-bit AVR devices, excluding the attiny families (such as attiny5 and attiny40 etc).

When enabled, this feature instruments the project's program image with small assembly sequences. When the program image is executed, these sequences record the execution of the code that they represent in reserved areas of device RAM. The records stored in the device can be later analyzed to determine which parts of a project's source code have been executed. Compiler-supplied library code is not instrumented.

When code coverage is enabled, the compiler will execute an external tool called `xc-ccov` to determine the most efficient way to instrument the project. The tool considers the program's basic blocks, which can be considered as sequences of one or more instructions with only one entry point, located at the start of the sequence and only one exit located at the end. Not all of these blocks need to be instrumented, with the tool determining the minimum set of blocks that will allow the program to be fully analyzed.

Use the `-mcodecov` option to enable code coverage in the compiler.

All compiler options you use to build the project, when using code coverage, are significant, as these will affect the program image that is ultimately instrumented. To ensure that the analysis accurately reflects the shipped product, the build options should be the same as those that will be used for the final release build.

If code coverage is enabled, there will be 1 bit of RAM allocated per instrumented basic block, which will increase the data memory requirement of the project.

There is a small sequence of assembly instructions inserted into each instrumented basic block to set the corresponding coverage bit.

The instrumented project code must be executed for code coverage data to be generated and this execution will be fractionally slower due to the added assembly sequences. Provide the running program with input and stimulus that should exercise all parts of the program code, so that execution of all parts of the program source can be recorded.

Code coverage data can be analyzed in the MPLAB X IDE. Information in the ELF file produced by the compiler will allow the plugin to locate and read the device memory containing the code coverage results and display this in a usable format. (See https://www.microchip.com/Developmenttools/ProductDetails/SW006026-COV for information on how to use the new code coverage features).

4.3 Supported Data Types and Variables

Values in the C programming language conform to one of several data types, which determine their storage size, format, and range of values. Variables and objects used to store these values are defined using the same set of types.

4.3.1 Identifiers

Identifiers are used to represent C objects and functions and must conform to strict rules.

A C identifier is a sequence of letters and digits where the underscore character "_" counts as a letter. Identifiers cannot start with a digit. Although they can start with an underscore, such identifiers are reserved for the compiler's use and should not be defined by C source code in your programs. Such is not the case for assembly-domain identifiers.

Identifiers are case sensitive, so `main` is different to `Main`.

4.3.2 Integer Data Types

The MPLAB XC8 compiler supports integer data types with 1, 2, 4 and 8 byte sizes. Tabulated below are the supported data types and their corresponding size and arithmetic type.

Table 4-1. Integer Data Types

Type	Size (bits)	Arithmetic Type
`signed char`	8	Signed integer
`unsigned char`	8	Unsigned integer
`signed short`	16	Signed integer
`unsigned short`	16	Unsigned integer
`signed int`	16	Signed integer
`unsigned int`	16	Unsigned integer
`signed long`	32	Signed integer

...........continued		
Type	**Size (bits)**	**Arithmetic Type**
`unsigned long`	32	Unsigned integer
`signed long long`	64	Signed integer
`unsigned long long`	64	Unsigned integer

If no type signedness is specified (including `char` types), then that type is `signed`.

All integer values are represented in little endian format with the Least Significant Byte (LSB) at the lower address in the device memory.

Signed values are stored as a two's complement integer value.

The range of values capable of being held by these types is summarized in 7.10 <limits.h> Implementation-Defined Limits The symbols in this table are preprocessor macros which are available after including `<limits.h>` in your source code. As the size of data types are not fully specified by the C Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

Macros are also available in `<stdint.h>` which define values associated with fixed-width types, such as `int8_t`, `uint32_t` etc.

4.3.3 Boolean Types

The compiler supports `_Bool`, a type used for holding true and false values.

The values held by variables of this type are not integers and behave differently in expressions compared to similar expressions involving integers types. Values converted to a `_Bool` type result in 0 (false) if the value is 0; otherwise, they result in 1 (true).

The `<stdbool.h>` header defines `true` and `false` macros that can be used with `_Bool` types and the `bool` macro, which expands to the `_Bool` type. For example:

```
#include <stdbool.h>
_Bool motorOn;
motorOn = false;
```

4.3.4 Floating-Point Data Types

The MPLAB XC8 compiler supports 32-bit floating-point types. Floating point is implemented using a IEEE 754 32-bit format. Tabulated below are the floating-point data types and their size.

Table 4-2. Floating-Point Data Types

Type	Size (bits)	Arithmetic Type
`float`	32	Real
`double`	32	Real
`long double`	32	Real

Floating-point types are always signed and the `unsigned` keyword is illegal when specifying a floating-point type. All floating-point values are represented in little endian format with the LSB at the lower address.

Infinities are legal arguments for all operations and behave as the largest representable number with that sign. For example, the expression `+inf + -inf` yields the value 0.

The format for both floating-point types is described in the table below, where:

- *Sign* is the sign bit, which indicates whether the number is positive or negative.
- The *Biased Exponent* is 8 bits wide and is stored as excess 127 (i.e., an exponent of 0 is stored as 127).
- *Mantissa*, is located to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number is $(-1)^{sign} \times 2^{(exponent-127)} \times 1.\mathit{mantissa}$.

Table 4-3. Floating-Point Formats

Format	Sign	Biased Exponent	Mantissa
IEEE 754 32-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx

An example of the IEEE 754 32-bit format shown in the following table. Note that the Most Significant Bit (MSb) of the mantissa column (i.e., the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero.

Table 4-4. Floating-Point Format Example IEEE 754

Format	Value	Biased Exponent	1.mantissa	Decimal
32-bit	7DA6B69Bh	11111011b	1.01001101011011010011011b	2.77000e+37
		(251)	(1.302447676659)	—

The sign bit is zero; the biased exponent is 251, so the exponent is 251-127=124. Take the binary number to the right of the decimal point in the mantissa. Convert this to decimal and divide it by 223 where 23 is the size of the mantissa, to give 0.302447676659. Add 1 to this fraction. The floating-point number is then given by:

$-1^{0} \times 2^{124} \times 1.302447676659$

which is approximately equal to:

2.77000e+37

Binary floating-point values are sometimes misunderstood. It is important to remember that not every floating-point value can be represented by a finite sized floating-point object. The size of the exponent in the number dictates the range of values that the number can hold and the size of the mantissa relates to the spacing of each value that can be represented exactly.

For example, if you are using a 32-bit wide floating-point type, it can store the value 95000.0 exactly. However, the next highest value which can be represented is (approximately) 95000.00781.

The characteristics of the floating-point formats are summarized in 7.7 <float.h> Floating-Point Characteristics.

The symbols in this table are preprocessor macros that are available after including `<float.h>` in your source code. As the size and format of floating-point data types are not fully specified by the C Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

4.3.5 Structures and Unions

MPLAB XC8 C Compiler supports `struct` and `union` types. Structures and unions only differ in the memory offset applied to each member.

These types will be at least 1 byte wide. Bit-fields and `_Bool` objects are fully supported.

Structures and unions can be passed freely as function arguments and function return values. Pointers to structures and unions are fully supported.

4.3.5.1 Structure and Union Qualifiers

The compiler supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will inherit this qualification. In the following example the structure is qualified `const`.

```
const struct {
  int number;
  int *ptr;
} record = { 0x55, &i };
```

In this case, each structure member will be read-only. Remember that all members should be initialized if a structure is `const`, as they cannot be initialized at runtime.

4.3.5.2 Bit-fields In Structures

MPLAB XC8 C Compiler fully supports bit-fields in structures.

Bit-fields are always allocated within 8-bit words, even though it is usual to use the type `unsigned int` in the definition. The first bit defined will be the LSb of the word in which it will be stored. When a bit-field is declared, it is allocated within the current 8-bit unit if it will fit; otherwise, a new byte is allocated within the structure.

Bit-fields can span the boundary between 8-bit allocation units; however, the code to access bit-fields that do so is extremely inefficient.

Consider the following definition:

```
struct {
    unsigned        lo : 1;
    unsigned        dummy : 6;
    unsigned        hi : 1;
} foo;
```

This will produce a structure occupying 1 byte.

If `foo` was ultimately linked at address 0x10, the field `lo` will be bit 0 of address 0x10 and field `hi` will be bit 7 of address 0x10. The LSb of `dummy` will be bit 1 of address 0x10.

Note: Accessing bit-fields larger than a single bit can be very inefficient. If code size and execution speed are critical, consider using a `char` type or a `char` structure member, instead. Be aware that some SFRs are defined as bit-fields. Most are single bits, but some can be multi-bit objects.

Unnamed bit-fields can be declared to pad out unused space between active bits in control registers. For example, if `dummy` is never referenced, the structure above could have been declared as:

```
struct {
    unsigned        lo : 1;
    unsigned           : 6;
    unsigned        hi : 1;
} foo;
```

A structure with bit-fields can be initialized by supplying a comma-separated list of initial values for each field. For example:

```
struct {
    unsigned        lo  : 1;
    unsigned        mid : 6;
    unsigned        hi  : 1;
} foo = {1, 8, 0};
```

Structures with unnamed bit-fields can be initialized. No initial value should be supplied for the unnamed members, for example:

```
struct {
    unsigned        lo  : 1;
    unsigned            : 6;
    unsigned        hi  : 1;
} foo = {1, 0};
```

will initialize the members `lo` and `hi` correctly.

A bit-field that has a size of 0 is a special case. The Standard indicates that no further bit-field is to be packed into the allocation unit in which the previous bit-field, if any, was placed.

4.3.5.3 Anonymous Structures And Unions

The MPLAB XC8 compiler supports anonymous structures and unions. These are C11 constructs with no identifier and whose members can be accessed without referencing the identifier of the construct. Anonymous structures and unions must be placed inside other structures or unions. For example:

```
struct {
    union {
        int x;
```

```
        double y;
    };
} aaa;
aaa.x = 99;
```

Here, the `union` is not named and its members are accessed as if they are part of the structure.

4.3.6 Pointer Types

There are two basic pointer types supported by the MPLAB XC8 C Compiler:

Data pointers	These hold the addresses of objects which can be read (and possibly written) by the program.
Function pointers	These hold the address of an executable function which can be called via the pointer.

These pointer types cannot be used interchangeably. Data pointers (even generic `void *` pointers) should never be used to hold the address of functions and function pointers should never be used to hold the address of objects.

4.3.6.1 Combining Type Qualifiers And Pointers

It is helpful to first review the C conventions for definitions of pointer types.

Pointers can be qualified like any other C object, but care must be taken when doing so as there are two quantities associated with pointers. The first is the actual pointer itself, which is treated like any ordinary C object and has memory reserved for it. The second is the target, or targets, that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following:

```
target_type_&_qualifiers * pointer's_qualifiers pointer's_name;
```

Any qualifiers to the right of the `*` (i.e., next to the pointer's name) relate to the pointer variable itself. The type and any qualifiers to the left of the `*` relate to the pointer's targets. This makes sense since it is also the `*` operator that dereferences a pointer and that allows you to get from the pointer variable to its current target.

Here are three examples of pointer definitions using the `volatile` qualifier. The fields in the definitions have been highlighted with spacing:

```
volatile int *            vip ;
int          * volatile  ivp ;
volatile int * volatile  vivp ;
```

The first example is a pointer called `vip`. The pointer itself – the variable that holds the address – is not `volatile`; however, the objects that are accessed when the pointer is dereferenced are treated as being `volatile`. In other words, the target objects accessible via the pointer could be externally modified.

In the second example, the pointer called `ivp` is `volatile`, that is, the address the pointer contains could be externally modified; however, the objects that can be accessed when dereferencing the pointer are not `volatile`.

The last example is of a pointer called `vivp` which is itself qualified `volatile`, and which also holds the address of `volatile` objects.

Bear in mind that one pointer can be assigned the addresses of many objects; for example, a pointer that is a parameter to a function is assigned a new object address every time the function is called. The definition of the pointer must be valid for every target address assigned.

Note: Care must be taken when describing pointers. Is a "const pointer" a pointer that points to `const` objects, or a pointer that is `const` itself? You can talk about "pointers to const" and "const pointers" to help clarify the definition, but such terms might not be universally understood.

4.3.6.2 Data Pointers

Pointers to objects that are located in only the data space are all 2 bytes wide.

A pointer qualified with `__flash` or `__flashn`, where *n* can range from 1 thru 5, can access objects data objects stored in program memory. Both these pointer types are 16 bits wide. Pointers to `__flash` use the `lpm` instruction to access data; pointers to `__flashn` use the RAMPZ register and the `elpm` instruction. Neither pointer can be used to access objects in RAM.

These pointers must be assigned the addresses of objects that are defined using the same qualifiers, so for example, a pointer to `__flash1` must only be assigned the addresses of objects that also use the `__flash1` qualifier.

4.3.6.2.1 Pointers to Both Memory Spaces

Any pointer to `const` has the ability to read objects in both the data and program memory space. Such pointers are said to be mixed memory space pointers and they may be larger than regular data pointers defined without using `const` on the target type.

For example, the following function can read and return an `int` from either memory space, depending on the address passed to it.

```
int read(const int * mip) {
  return *mip;
}
```

For any device in the avrtiny or avrxmega3 families, these pointers are 16-bits wide and can target objects in data or in program memory, since the program memory is mapped into the data space. For other devices, the pointers are 24 bits wide.

Addresses of type `const *` that are 24-bits wide linearize flash and RAM, using the high bit of the address to determine which memory space is being accessed. If the MSb is set, the object is accessed from data memory using the lower two bytes as address. If the MSb of the address is clear, data is accessed from program memory, with the RAMPZ segment register set according to the high byte of the address.

Note that the pointer type `const int *` is similar to the `const __memx int *` pointer type in terms of how it accesses objects, but it does not require the use of non-standard keywords, and access is controllable using the `-mconst-data-in-progmem` option (see 3.6.1.3 Const-data-in-progmem Option).

4.3.6.3 Function Pointers

The MPLAB XC8 compiler fully supports pointers to functions. These are often used to call one of several function addresses stored in a user-defined C array, which acts like a lookup table.

Function pointers are 2 bytes in size. As the address is word aligned, such pointers can reach program memory addresses up to 128kB. If the device you are using supports more than this amount of program memory and you wish to indirectly access routines above this address, then you need to use the `-mrelax` option (see 3.6.1.8 Relax Option), which maintain the size of the pointer, but will instruct the linker to have calls reach their final destination via lookups.

In order to facilitate indirect jump on devices with more than 128 Ki bytes of program memory space, there is a special function register called EIND that serves as most significant part of the target address when `eicall` or `eijmp` instructions are executed. The compiler might also use this register in order to emulate an indirect call or jump by means of a `ret` instruction.

The compiler never sets the EIND register and assumes that it never changes during the startup code or program execution, and this implies that the EIND register is not saved or restored in function or interrupt service routine prologues or epilogues.

To accommodate indirect calls to functions and computed gotos, the linker generates function stubs, or trampolines, that contain direct jumps to the desired addresses. Indirect calls and jumps are made to the stub, which then redirects execution to the desired function or location.

For the stubs to work correctly, the `-mrelax` option must be used. This option ensures that the linker will use a 16-bit function pointer and stub combination, even though the destination address might be above 128 kB.

The default linker script assumes code requires the EIND register contain zero. If this is not the case, a customized linker script must be used in order to place the sections whose name begin with `.trampolines` into the segment appropriate to the value held by the EIND register.

The startup code from the `libgcc.a` library never sets the EIND register.

It is legitimate for user-specific startup code to set up EIND early, for example by means of initialization code located in section `.init3`. Such code runs prior to general startup code that initializes RAM and calls constructors, but after the AVR-LibC startup code that sets the EIND register to a value appropriate for the location of the vector table.

```
#include <avr/io.h>
static void
__attribute__((section(".init3"),naked,used,no_instrument_function))
init3_set_eind (void)
{
  __asm volatile ("ldi r24,pm_hh8(__trampolines_start)\n\t"
  "out %i0,r24" :: "n" (&EIND) : "r24","memory");
}
```

The `__trampolines_start` symbol is defined in the linker script.

Stubs are generated automatically by the linker, if the following two conditions are met:

- The address of a label is taken by means of the `gs` assembler modifier (short for generate stubs) like so:
  `LDI r24, lo8(gs(func))`

  `LDI r25, hi8(gs(func))`
- The final location of that label is in a code segment outside the segment where the stubs are located.

The compiler emits `gs` modifiers for code labels in the following situations:

- When taking the address of a function or code label
- When generating a computed goto
- If the prologue-save function is used (see 3.6.1.2 Call-prologues Option)
- When generating switch/case dispatch tables (these can be inhibited by specifying the `-fno-jump-tables` option, 3.6.1.7 No-jump-tables Option)
- C and C++ constructors/destructors called during startup/shutdown

Jumping to absolute addresses is not supported, as shown in the following example:

```
int main (void)
{
  /* Call function at word address 0x2 */
  return ((int(*)(void)) 0x2)();
}
```

Instead, the function has to be called through a symbol (`func_4` in the following example) so that a stub can be set up:

```
int main (void)
{
  extern int func_4 (void);
  /* Call function at byte address 0x4 */
  return func_4();
}
```

The project should be linked with `-Wl,--defsym,func_4=0x4`. Alternatively, `func_4` can be defined in the linker script.

4.3.7 Constant Types and Formats

Constant in C are an immediate value that can be specified in several formats and that are assigned a type.

4.3.7.1 Integral Constants

The format of integral constants specifies their radix. MPLAB XC8 supports the standard radix specifiers, as well as ones which enables binary constants to be specified in C code.

The formats used to specify the radices are tabulated below. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

Table 4-5. Radix Formats

Radix	Format	Example
binary	`0b`*number* or `0B`*number*	`0b10011010`
octal	`0`*number*	`0763`
decimal	*number*	`129`
hexadecimal	`0x`*number* or `0X`*number*	`0x2F`

Any integral constant will have a type of `int`, `long int` or `long long int`, so that the type can hold the value without overflow. Constants specified in octal or hexadecimal can also be assigned a type of `unsigned int`, `unsigned long int` or `unsigned long long int` if their signed counterparts are too small to hold the value.

The default types of constants can be changed by the addition of a suffix after the digits; e.g., `23U`, where `U` is the suffix. The table below shows the possible combination of suffixes and the types that are considered when assigning a type. For example, if the suffix `l` is specified and the value is a decimal constant, the compiler will assign the type `long int`, if that type will hold the constant; otherwise, it will assigned `long long int`. If the constant was specified as an octal or hexadecimal constant, then unsigned types are also considered.

Table 4-6. Suffixes And Assigned Types

Suffix	Decimal	Octal or Hexadecimal

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`, and \ or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
`u` or `U`, and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

Here is an example of code that can fail because the default type assigned to a constant is not appropriate:

```
unsigned long int result;
unsigned char shifter;
shifter = 20;
result = 1 << shifter;   // oops!
```

The constant 1 (one) will be assigned an `int` type. If the size of an int type on the compiler being used is hence the value 1 shifted left 20 bits will yield the result 0, not 0x100000.

The following uses a suffix to change the type of the constant, hence ensure the shift result has an unsigned long type.

```
result = 1UL << shifter;
```

4.3.7.2 Floating-point Constant

Floating-point constants have `double` type unless suffixed by `f` or `F`, in which case it is a `float` constant. The suffixes `l` or `L` specify a `long double` type which is considered an identical type to `double` by MPLAB XC8.

Floating point constants can be specified as decimal digits with a decimal point and/or an exponent. If you are using C99, they can be expressed as hexadecimal digits and a binary exponent, initiated with either `p` or `P`. For example:

```
myFloat = -123.98E12;
myFloat = 0xFFEp-22;  // C99 float representation
```

4.3.7.3 Character And String Constants

Character constants are enclosed by single quote characters, `'`, for example `'a'`. A character constant has `int` type, although this can be later optimized to a `char` type by the compiler.

To comply with the C standard, the compiler does not support the extended character set in characters or character arrays. Instead, they need to be escaped using the backslash character, as in the following example.

```
const char name[] = "Bj\370rk";
printf("%s's Resum\351", name);     \\ prints "Bjørk's Resumé"
```

Multi-byte character constants are not supported by this implementation.

String constants, or string literals, are enclosed by double quote characters `"`, for example `"hello world"`. The type of string constants is `const char *` and the characters that make up the string are stored in program memory, as are all objects qualified `const`.

A common warning relates to assigning a string literal, which cannot be modified, to a pointer that does not specify a `const` target, for example:

```
char * cp = "hello world\n";
```

See 4.3.6.1 Combining Type Qualifiers And Pointers and qualify the pointer as follows.

```
const char * cp = "hello world\n";
```

Defining and initializing an array (i.e., not a pointer) with a string is an exception. For example:

```
char ca[]= "hello world\n";
```

will actually copy the string characters into the RAM array, rather than assign the address of the characters to a pointer, as in the previous examples. The string literal remains read-only, but the array is both readable and writable.

The MPLAB XC8 compiler will use the same storage location and label for strings that have identical character sequences, except where the strings are used to initialize an array residing in the data space. For example, in the code snippet

```
if(strncmp(scp, "hello", 6) == 0)
    fred = 0;
if(strcmp(scp, "world") == 0)
    fred--;
if(strcmp(scp, "hello world") == 0)
    fred++;
```

the characters in the string `"world"` and the last 6 characters of the string `"hello world"` (the last character is the null terminator character) would be represented by the same characters in memory. The string `"hello"` would not overlap with the same characters in the string `"hello world"` as they differ in terms of the placement of the null character.

4.3.8 Standard Type Qualifiers

The compiler supports the standard qualifiers `const` and `volatile`, which are important for embedded application development.

4.3.8.1 Const Type Qualifier

The `const` type qualifier is used to tell the compiler that an object is read only and should not be modified. If any attempt is made to modify an object declared `const`, the compiler will issue a warning or error.

Provided that no other storage qualifiers are used, `const`-qualified objects (excluding autos and parameters) are by default linked into the program space, however this can be changed by using the `-mno-const-data-in-progmem` option (see 3.6.1.3 Const-data-in-progmem Option). Such objects can also be made absolute, allowing them to be easily placed at a specific address in the program memory, see 4.4.4.1 Absolute Objects In Program Memory.

Objects qualified with `const` are accessed as if they were qualified with `__memx`, see 4.3.9.1 Memx Address Space Qualifier. If the definition of `const`-qualified objects also uses storage qualifiers, such as `__flash` or `__flashn` (see 4.3.9.2 Flash Qualifier and 4.3.9.3 Flashn Qualifiers) the read access strategy implied by these qualifiers take precedence over that implied by `const`.

Usually a `const` object must be initialized when it is declared, as it cannot be assigned a value at any point at runtime. For example:

```
const int version = 3;
```

will define `version` as being a read-only `int` variable, holding the value 3.

4.3.8.2 Volatile Type Qualifier

The `volatile` type qualifier indicates to the compiler that an object cannot be guaranteed to retain its value between successive accesses. This information prevents the optimizer from eliminating apparently redundant references to objects declared `volatile` because these references might alter the behavior of the program.

Any SFR which can be modified by hardware or which drives hardware is qualified as `volatile`, and any variables which can be modified by interrupt routines should use this qualifier as well. For example:

```
#include <xc.h>
volatile static unsigned int TACTL __at(0x800160);
```

The `volatile` qualifier does not guarantee that any access will be atomic, which is often not the case, since the 8-bit AVR architecture can typically access only 1 byte of data per instruction.

The code produced by the compiler to access `volatile` objects can be different of that to access ordinary variables and typically the code will be longer and slower for `volatile` objects, so only use this qualifier if it is necessary. Failure to use this qualifier when it is required, however, can lead to code failure.

A common use of the `volatile` keyword is to prevent some unused variables being removed. If a non-`volatile` variable is never used, or used in a way that has no effect, then it can be removed before code is generated by the compiler.

A C statement that consists only of a `volatile` variable's identifier will produce code that reads the variable's memory location and discards the result. For example, if `PORTB;` is an entire statement, it will produce assembly code the reads PORTB.

4.3.9 Special Type Qualifiers

The MPLAB XC8 C Compiler supports special type qualifiers to allow the user to control placement of objects with static storage duration into particular address spaces.

4.3.9.1 Memx Address Space Qualifier

Using both the `__memx` and `const` qualifiers indicates that the object is to be placed in the program space. This method of placing objects in flash is now largely redundant, as you can perform the same task using just the `const` qualifier, see 4.3.8.1 Const Type Qualifier. The `__memx` qualifier is not needed when compiling for any device that maps the program memory into their data memory space, such as the avrxmega3 and avrtiny devices.

Pointers that use this qualifier can reference both data and program memory spaces in a similar way to pointers to `const`, see 4.3.6.2.1 Pointers to Both Memory Spaces.

4.3.9.2 Flash Qualifier

Using both the `__flash` and `const` qualifiers indicates that the object should be located in a different program memory section. This section can be repositioned using the project's linker script, if desired, but must be wholly linked in the lowest flash 64k segment. The section associated with the `__flash` qualifier is by default located in the lowest flash segment (segment 0), which occupies the lowest 64kb address range.

For devices that do not have memory-mapped flash, data is read using the `lpm` instruction.

Pointers that use this qualifier can reference program memory, see 4.3.6.2 Data Pointers.

4.3.9.3 Flashn Qualifiers

Using both the `__flash`*n* and `const` qualifiers, where *n* can range from 1 thru 5, indicates that the object should be located in a different program memory section. For correct program operation, these sections must be positioned into the correct flash segment using the project's linker script. Subject to your target device having the flash segments implemented, the `__flash1` qualifier is associated with flash segment 1, which should be located in the flash segment from address 64k to 128kb, with `__flash2` qualified objects placed in a section that is linked to the segment from 128k to 196kb, etc. Clearly, not all the `__flash`*n* qualifiers will available with all devices.

For those devices that do not have memory-mapped flash, data is read using the RAMPZ register and the *elpm* instruction, allowing the full program memory to be read.

Pointers that use these qualifiers can reference program memory, see 4.3.6.2 Data Pointers.

4.3.10 Attributes

The compiler keyword `__attribute__()` allows you to specify special attributes of objects or structure fields. Place inside the parentheses following this keyword a comma-separated list of the relevant attributes, for example:

```
__attribute__((unused))
```

The attribute can be placed anywhere in the object's definition, but is usually placed as in the following examples.

```
char __attribute__((weak)) input;
char input __attribute__((weak));
```

Note: It is important to use variable attributes consistently throughout a project. For example, if a variable is defined in one file with the `aligned` attribute and declared `extern` in another file without the `aligned attribute`, then a link error may result.

4.3.10.1 Absdata Attribute

The `absdata` attribute indicates that the objects can be accessed by the `lds` and `sts` instructions, which take absolute addresses. This attribute is only supported for the reduced AVR Tiny core like ATtiny40.

You must make sure that respective data is located in the address range 0x40-0xbf to prevent out of range errors. One way to achieve this as an appropriate linker description file.

4.3.10.2 Address Attribute

Variables with the `address(`*addr*`)` attribute are used to address memory-mapped peripherals that may lie outside the io address range.

```
volatile int porta __attribute__((address (0x600)));
```

To place objects at a specified address in the ordinary data memory, use the `__at()` specifier (see 4.4.4 Absolute Variables).

4.3.10.3 Aligned Attribute

The `aligned(`*n*`)` attributed aligns the object's address with the next *n*-byte boundary, where *n* is an numerical argument to the attribute. If the CCI is enabled (see 3.6.3.3 Ext Option) a more portable macro, `__align(`*n*`)` (note the different spelling), is available.

This attribute can also be used on a structure member. Such a member will be aligned to the indicated boundary within the structure.

If the alignment argument is omitted, the alignment of the variable is set to 1 (the largest alignment value for a basic data type).

Note that the aligned attribute is used to increase the alignment of a variable, not reduce it. To decrease the alignment value of a variable, use the `packed` attribute.

4.3.10.4 Deprecated Attribute

The `deprecated` attribute generates a warning whenever the specified object is used. If an option string argument is present, it will be printed in the warning. If the CCI is enabled (see 3.6.3.3 Ext Option) a more portable macro, `__deprecate` (note the different spelling), is available.

4.3.10.5 Io Attribute

Objects defined using the `io(address)` attribute represent memory-mapped peripherals in the I/O space and at the address indicated. Example:

```
volatile int porta __attribute__((io(0x22)));
```

When used without an address, the object it is not assigned an address, but the compiler will still use `in` and `out` instructions where applicable, assuming some other module will assign the object an address. For example:

```
extern volatile int porta __attribute__((io));
```

4.3.10.6 Io_low Attribute

The `io_low(address)` attribute is similar the `io(address)` attribute, but additionally it informs the compiler that the object lies in the lower half of the I/O area, allowing the use of `cbi`, `sbi`, `sbic` and `sbis` instructions. This attribute also has an `io_low` form, which does not specify an address.

4.3.10.7 Packed Attribute

The `packed` attribute forces the object or structure member to have the smallest possible alignment, that is, no alignment padding storage will be allocated for the declaration. Used in combination with the `aligned` attribute, `packed` can be used to set an arbitrary alignment restriction greater or lesser than the default alignment for the type of the variable or structure member.

If the CCI is enabled (see 3.6.3.3 Ext Option) a more portable macro, `__pack` (note the different spelling), is available.

4.3.10.8 Persistent Attribute

The `persistent` attribute is used to indicate that objects should not be cleared by the runtime startup code by having them stored in a different area of memory to other objects. If the CCI is enabled (see 3.6.3.3 Ext Option) a more portable macro, `__persistent`, is available.

By default, C objects with static storage duration that are not explicitly initialized are cleared on startup. This is consistent with the definition of the C language. However, there are occasions where it is desired for some data to be preserved across a Reset.

For example, the following CCI-compliant code ensures that the variable, `intvar`, is not cleared at startup:

```
void test(void)
{
 static __persistent int intvar; /* must be static */
  // ...
}
```

4.3.10.9 Progmem Attribute

Using both the `progmem` attribute and the `const` qualifier allows you to have objects placed in the program memory. Alternatively, you can use the more portable `PROGMEM` macro, defined by `<avr/pgmspace.h>`, which maps to this attribute. For example.

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romChar = 0x55;
```

Prior to the AVR GCC compiler supporting names address spaces, this was the only way in which objects could be placed in flash, thus it exists today for compatibility with legacy projects or to improve portability of code migrated from other platforms.

Unlike `const`-qualified objects in program memory, which can be read directly, objects defined using the `progmem` attribute must be read using an appropriate library function, e.g., `pgm_read_byte_near()`. It is up to the programmer to use the correct library function; however, the code to access objects defined in this way is typically efficient, since their location is more accurately known by the compiler.

4.3.10.10 Section Attribute

The `section(section)` attribute allocates the object to a user-nominated section rather than allowing the compiler to place it in a default section. If the CCI is enabled (see 3.6.3.3 Ext Option) a more portable macro, `__section(section)`, is available. See 4.15 Changing and Linking the Allocated Section for full information on the use of this qualifier.

For example, the following CCI-compliant code places `foobar` in a unique section called `myData`:

```
int __section("myData") foobar;
```

4.3.10.11 Unused Attribute

The `unused` attribute indicates to the compiler that the object might not be used and that no warnings should be produced if it is detected as being unused.

4.4 Memory Allocation and Access

Objects you define are automatically allocated to an area of memory. In some instances, it is possible to alter this allocation. Memory areas and allocation are discussed in the following sections.

4.4.1 Address Spaces

Most 8-bit AVR devices have a Harvard architecture, which has a separate data memory (RAM) and program memory space. On some devices, the program memory is mapped into and accessible from the data memory space. Some devices also implement EEPROM, which is memory mapped on some devices.

Both the general purpose RAM and SFRs share the same data space; however, SFRs appear in a range of addresses (called the I/O space in the device data sheets) that can be accessed by instructions that access the I/O space, such as the `in` and `out` instructions. If a device has more SFRs than these instructions can address, the registers are located at a higher address and accessed via the `st` and `ld` group of instructions.

The program memory space is primarily for executable code, but data can also be located here. There are several ways the different device families locate and read data from this memory, but all objects located here will be read-only.

4.4.2 Objects in Data Memory

Most variables are ultimately positioned into the data memory. Due to the fundamentally different way in which automatic and static storage duration objects are allocated memory, they are discussed separately.

4.4.2.1 Static Storage Duration Objects

Objects which are not allocated space on a stack (all objects excluding auto, parameter and `const`-qualified objects) have a static (permanent) storage duration and are located by the compiler into the data memory.

Allocation is performed in two steps. The compiler places each object into a specific section and then the linker places these sections into the relevant memory areas. After placement, the addresses of the objects in those sections can be fully resolved.

The compiler considers two object categories, which relate to the value the object should contain when the program begins execution. Each object category has a corresponding family of sections (see 4.14.1 Compiler-Generated Sections), which are tabulated below.

bss These sections contain any uninitialized objects, which will be cleared by the runtime startup code.

data These sections contain the RAM image of initialized objects, whose non-zero value is copied to them by the runtime startup code.

See 4.9 Main, Runtime Startup and Reset has information on how the runtime startup code operates.

4.4.2.1.1 Static Objects

All `static` objects have static storage duration, even local `static` objects, defined inside a function and which have a scope limited to that function. Even local `static` objects can be referenced by a pointer and are guaranteed to retain their value between calls to the function in which they are defined, unless explicitly modified via a pointer.

Objects which are `static` only have their initial value assigned once during the program's execution. Thus, they can be preferable over initialized auto objects which are assigned a value every time the block in they are defined begins execution.

All `static` variables which are also specified as `const` will be stored in program memory.

4.4.2.1.2 Changing the Default Allocation

You can change the default memory allocation of objects with static storage duration by either:

- Using specifiers
- Making the objects absolute
- Placing objects in their own section and explicitly linking that section

Variables can be placed in a combined flash and data section by using the `__memx` specifier (see 4.3.9.1 Memx Address Space Qualifier).

If only a few objects are to be located at specific addresses in data space memory, then those objects can be made absolute (described in 4.4.4 Absolute Variables). Once variables are made absolute, their address is hard coded in generated output code, they are no longer placed in a section and do not follow the normal memory allocation procedure.

The `.bss` and `.data` sections, in which the different categories of static storage duration objects are allocated, can be shifted as a whole by changing the default linker options. For example, you could move all the persistent variables.

Objects can also be placed at specific positions by using the `__section()` specifier to allocate them to a unique section, then link that section to the required address via an option.

See 4.15 Changing and Linking the Allocated Section for more information on changing the default linker options for sections.

4.4.2.2 Automatic Storage Duration Objects

Objects with automatic storage duration, such as auto, parameter objects, and temporary variables, are allocated space on a stack implemented by the compiler. Temporary objects might be placed on the stack as well. The stack used by MPLAB XC8 and the 8-bit AVR devices is described in 4.2.3 Stacks.

Since objects with automatic storage duration are not in existence for the entire execution of the program, there is the possibility to reclaim memory they use when the objects are not in existence and allocate it to other objects in the program. Typically such objects are stored on some sort of a dynamic data stack where memory can be easily allocated and deallocated by each function. Because this stack is used to create new instances of function objects when the function is called, all functions are reentrant.

The standard `const` qualifier can be used with auto objects and these do not affect how they are positioned in memory. This implies that a local `const`-qualified object is still an auto object and will be allocated memory in the stack of data space memory.

4.4.2.2.1 Object Size Limits

An object with automatic storage duration cannot be made larger than the stack space available at the time at which the object comes into existence. Therefore, the maximum permitted size is not fixed and will depend on the program's execution.

4.4.2.2.2 Changing the Default Allocation

All objects with automatic storage duration are located on a stack, thus there is no means to individually move them. They cannot be made absolute, nor can they be assigned a unique section using the `__section()` specifier.

4.4.3 Objects in Program Space

Objects defined using the `const` qualifier and that have static storage duration are placed in program memory. The same is true for objects defined using `const` and any one of the `__flash`, `__flash`*n*, `__memx` qualifiers, or using `const` and the `progmem` attribute.

The avrtiny and avrxmega3 device families can easily access program-memory objects, since this memory is mapped into the data address space. For other device families, program memory is distinct and is accessed via different code sequences.

The macro `__AVR_CONST_DATA_IN_PROGMEM__` is defined if objects qualified using only `const` will be located in program memory by the compiler.

4.4.3.1 Size Limitations of Program-memory Objects

Objects defined using just `const`, or `const` and `__memx` are limited only by the available program memory, and objects defined using the `progmem` attribute can span multiple flash segments, but functions from the `pgm_read_xxx_far()` family must be used to access them. Objects defined using the `__flash`, or `__flashn` qualifiers are limited to the memory available in one 64kb segment of flash memory and must never cross a segment boundary; however the linker script can be adjusted so that the relevant sections can be located in any one of the available flash segments.

Note that in addition to the data itself, extra code is required to read data in program memory for those devices that do not have program memory mapped into the data space. This code might be library code that is included only once; however the code sequences to read program memory are typically longer that those to read from RAM.

4.4.3.2 Changing the Default Allocation of Program-memory Objects

You can change the default memory allocation of objects in program memory by either:

- Making the objects absolute
- Placing objects in their own section and explicitly linking that section

If only a few program-memory objects are to be located at specific addresses in program space memory, then the objects can be made absolute. Absolute variables are described in 4.4.4 Absolute Variables.

Objects in program memory can also be placed at specific positions by using the `__section()` specifier (see 4.15 Changing and Linking the Allocated Section), to allocate them to a unique section, then link that section to the required address via an option.

4.4.4 Absolute Variables

Objects can be located at a specific address by following their declaration with the construct `__at(`*`address`*`)`, where *`address`* is the memory location at which the object is to be positioned. The CCI must be enabled (see 3.6.3.3 Ext Option) and `<xc.h>` must be included for this construct to compile. Such objects are known as an absolute objects.

Making a variable absolute is the easiest method to place an object at a user-defined location, but it only allows placement at an address which must be known prior to compilation and must be specified with each object to be relocated.

4.4.4.1 Absolute Objects In Program Memory

Any `const`-qualified object which has static storage duration and which has file scope can be placed at an absolute address in program memory.

For example:

```
const int settings[] __at(0x200) = { 1, 5, 10, 50, 100 };
```

will place the array `settings` at address 0x200 in the program memory.

4.4.5 Variables in EEPROM

For devices with on-chip EEPROM, the compiler offers several methods of accessing this memory as described in the following sections.

4.4.5.1 Eeprom Variables

Objects can be placed in the EEPROM by specifying that they be placed in the `.eeprom` section, using the `section` attribute. A warning is produced if the attribute is not supported for the selected device. Check your device data sheet to see the memory available with your device.

The macro `EEMEM` is defined in `<avr/eeprom.h>` and can be alternatively used to simplify the definition of objects in EEPROM. For example, both the following definitions create objects which will be stored in EEPROM.

```
int serial __attribute__((section(.eeprom)));
char EEMEM ID[5] = { 1, 2, 3, 4, 5 };
```

Objects in this section are cleared or initialized, as required, just like ordinary RAM-based objects; however, the initialization process is not carried out by the runtime startup code. Initial values are placed into a HEX file and are burnt into the EEPROM when you program the device. If you modify the EEPROM during program execution and then reset the device, these objects will not contain the initial values specified in your code at startup up.

Note that the objects that are in the `eeprom` section must all use the `const` type qualifier or all not use this qualifier.

4.4.5.2 Eeprom Access Functions

You must access objects in EEPROM using special library routines, such as `eeprom_read_byte()` and `eeprom_write_word()`, accessible once you include `<avr/eeprom.h>`.

Code to access EEPROM based objects will be much longer and slower than code to access RAM-based objects. Consider using these routines to copy values from the EEPROM to regular RAM-based objects if you need to use them many times in complex calculations.

4.4.6 Variables in Registers

You can define a variable and associate it with a specified register; however, it is generally recommended that register allocation be left to the compiler to achieve optimal results and to avoid code failure.

Register variables are defined by using the `register` keyword and indicating the desired register, as in the following example:

```
register int input asm("r12");
```

A valid AVR device register name must be quoted as an argument to the `asm()`. Such a definition can be placed inside or outside a function, but you cannot made the variable `static`. Support for local register variables is to limited to specifying registers for input and output operands when calling extended in-line assembly.

The compiler reserves the allocated register for the duration of the current compilation unit, but library routines may clobber the register allocated, thus it is recommended that you allocate a register that is normally saved and restored by function calls (a call-saved register, described in 4.6 Register Usage).

You cannot take the address of a `register` variable.

4.4.7 Dynamic Memory Allocation

Dynamic memory allocated from a heap at runtime (by functions like `malloc()` etc) is supported by MPLAB XC8. Given the small amounts of data memory available on the AVR architecture, the allocation scheme is relatively robust.

The memory allocated by dynamic memory functions includes an addition two-byte-wide header that is prepended to the requested memory. This header records the size of the allocation and is used by `free()`. The address returned by the memory allocation functions point to the first usable location that has been allocated. The two bytes located before this address contain the header, so your program should take extra care to ensure these locations are not corrupted.

The implementation maintains a simple freelist that accounts for memory regions that have been returned in previous calls to `free()`. Note that all of this memory is considered to have been successfully added to the heap, so no further checks against stack-heap collisions are done when recycling memory from the freelist.

The freelist itself is not maintained as a separate data structure. The contents of the freed memory regions are written with pointers which link the regions. This requires no additional memory to maintain this list, except for a link pointer, which contains the address of the lowest memory region available for reallocation. Since the size of the region (the two-byte header) and a two-byte link pointer to the next free region are recorded in each region, the minimum region size on the freelist is four bytes.

When allocating memory, the linked memory regions in the freelist are first walked to determine if this contains a memory region that satisfies the request. Regions with the same usable size as that requested are allocated first;

otherwise, larger regions are considered, if they are available. If a larger free region has at least four bytes more than that requested, it is split into one region which is returned by the allocation function and the remaining region is left on the free list. If splitting the larger free region would result in a region on the freelist that is less than four bytes in size (i.e. not large enough to hold the header information and link pointer), the larger region is not split and is allocated as a whole.

If no suitable memory region could be found on the freelist, the allocation functions attempt to extend the heap. If the heap is located below the stack, memory will be allocated up to a maximum address of the current stack limit `minus__malloc_margin` bytes, which by default is 32 bytes. If the heap is above the stack, memory will be allocated up to a maximum address of `__malloc_heap_end`.

If no memory can be claimed from the heap, the allocation functions will return `NULL`.

When calling `free()`, a new region will be placed on the freelist. This region will be combined with other contiguous regions, yielding the largest possible entry for further allocations. That way, the heap fragmentation can be minimized. When deallocating the topmost chunk of memory, the size of the heap is reduced.

A call to `realloc()` first determines whether the operation should increase or decrease the size of the existing allocation. If the new request is for a region at least two bytes smaller than the current region, the existing region is split and the region no longer required is passed to the standard `free()` function for insertion into the freelist. If the new request is for a region one byte smaller than the existing region, no operation is performed and the existing region is returned.

When a request to `realloc()` is for a larger memory region, the existing allocation is extended in-place, if possible, without having to copy data to the new region. As a side-effect, this check will also record the size of the largest chunk on the freelist.

If the existing region cannot be extended in-place, but is located at the top of heap with no suitable regions in the freelist, the heap is extended (if possible) without having to copy data to the new region. Otherwise, `malloc()` will be called with the new request size, the data in the existing region will be copied over to the new region, and `free()` will be called on the now defunct region.

The request will fail if the top of the heap would surpass its maximum permissible address.

4.4.7.1 Adjusting Dynamic Allocation Behavior

There are a number of variables that can be tuned to customize the behavior of functions such as `malloc()`. Any changes to these variables should be made before any memory allocation is made, remembering that library functions might use dynamic memory.

The variables `__malloc_heap_start` and `__malloc_heap_end` can be used to restrict the memory allocated by the `malloc()` function. These variables are statically initialized to point to `__heap_start` and `__heap_end`, respectively, where `__heap_start` is set to an address just beyond the `.bss` section, and `__heap_end` is set to 0, which places the heap is below the stack.

If the heap is located in external RAM, `__malloc_heap_end` must be adjusted accordingly. This can be done either at run-time, by writing directly to this variable, or it can be done automatically at link-time, by adjusting the value of the symbol `__heap_end`.

The following example shows an option that can relocate those input sections mapped to the `.data` output section in the linker script to location 0x1100 in external RAM. (The option `-Wl,-Tdata=0x801100` could also be used in this situation). The heap will extend up to address 0xffff.

```
-Wl,--section-start,.data=0x801100,--defsym=__heap_end=0x80ffff
```

Since these are addresses in RAM, the MSb is set in the address.

If the heap should be located in external RAM while keeping the ordinary variables in internal RAM, the following options can be used. Note that in this example, there is a 'hole' in memory between the heap and the stack that remain unaccessible by ordinary variables or dynamic memory allocations.

```
-Wl,--defsym=__heap_start=0x802000,--defsym=__heap_end=0x803fff
```

If `__malloc_heap_end` is 0, the memory allocation routines attempt to detect the bottom of the stack in order to prevent a stack-heap collision when extending the heap. They will not allocated memory beyond the current stack limit with a buffer of `__malloc_margin` bytes. Thus, all possible stack frames of interrupt routines that could interrupt the current function, plus all further nested function calls must not require more stack space, or they will risk colliding with the data segment.

The default value of `__malloc_margin` is set to 32.

4.4.8 Memory Models

MPLAB XC8 C Compiler does not use fixed memory models to alter allocation of variables to memory. Memory allocation is fully automatic and there are no memory model controls.

4.5 Operators and Statements

The MPLAB XC8 C Compiler supports all the ANSI operators, some of which behave in an implementation defined way, see 6. Implementation-Defined Behavior. The following sections illustrate code operations that are often misunderstood as well as additional operations that the compiler is capable of performing.

4.5.1 Integral Promotion

Integral promotion changes the type of some expression values. MPLAB XC8 C Compiler always performs integral promotion in accordance with the C standard, but this action can confuse those who are not expecting such behavior.

When there is more than one operand to an operator, the operands must typically be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they do have the same type. The conversion is to a “larger” type so there is no loss of information; however, the change in type can cause different code behavior to what is sometimes expected. These form the standard type conversions.

Prior to these type conversions, some operands are unconditionally converted to a larger type, even if both operands to an operator have the same type. This conversion is called integral promotion. The compiler performs these integral promotions where required and there are no options that can control or disable this operation.

Integral promotion is the implicit conversion of enumerated types, `signed` or `unsigned` varieties of `char`, `short int` or bit-field types to either `signed int` or `unsigned int`. If the result of the conversion can be represented by an `signed int`, then that is the destination type, otherwise the conversion is to unsigned int.

Consider the following example.

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

The `unsigned char` result of `a - b` is 206 (which is not less than 10), but both `a` and `b` are converted to `signed int` via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 (which is less than 10) and hence the body of the `if` statement is executed.

If the result of the subtraction is to be an unsigned quantity, then apply a cast, as in the following example, which forces the comparison to be done as `unsigned int` types:

```
if((unsigned int)(a - b) < 10)
  count++;
```

Another problem that frequently occurs is with the bitwise complement operator, `~`. This operator toggles each bit within a value. Consider the following code.

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

If `c` contains the value 0x55, it often assumed that `~c` will produce 0xAA; however, the result is instead 0xFFAA for compilers using a 16-bit `int`, or 0xFFFFFFAA for compilers that use a 32-bit `int`, and so the comparison in the above example would fail. The compiler is able to issue a mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behavior.

The consequence of integral promotion as illustrated above is that operations are not performed with `char`-type operands, but with `int`-type operands. However, there are circumstances when the result of an operation is identical regardless of whether the operands are of type `char` or `int`. In these cases, the compiler might not perform the integral promotion so as to increase the code efficiency. Consider this example.

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of `b` and `c` are promoted to `unsigned int`, the addition is performed, the result of the addition is cast to the type of `a` and that result is assigned. In this case, the value assigned to `a` will be the same whether the addition is performed as an `int` or `char`, and so the compiler might encode the former.

If in the above example, the type of `a` was `unsigned int`, then integral promotion would have to be performed to comply with the C Standard.

4.5.2 Rotation

The C language does not specify a rotate operator; however, it does allow shifts. You can follow the C code examples below to perform rotations for 16-bit integers.

```
unsigned char c;
unsigned int u;
c = (c << 1) | (c >> 7); // rotate left, one bit
u = (u >> 2) | (u << 14); // rotate right, two bits
```

4.5.3 Switch Statements

By default, jump tables can be used to optimize `switch()` statements. The `-fno-jump-tables` option prevents these from being used and will use sequences of compare statements instead. Jump tables are usually faster to execute on average, but in particular for `switch()` statements, where most of the jumps would go to the default label, they might waste a bit of flash memory.

The jump tables use the `lpm` assembler instruction for access to jump tables. Always use the `-fno-jump-tables` option when compiling a bootloader for devices with more than 64 kB of program memory.

4.6 Register Usage

The assembly generated from C source code by the compiler will use certain registers in the AVR register set. Some registers are assumed to hold their value over a function call.

The call-used registers, r18-r27 and r30-r31, can be allocated by the compiler for values within a function. Functions do not need to preserve the content of these registers. These registers may be used in hand-written assembler subroutines. Since any C function called by these routines can clobber these registers, the calling routine must ensure they are saved as restored as appropriate.

The call-saved registers, r2-r17 and r28-r29, can also be allocated by the compiler for local data; however, C functions must preserve these registers. Hand-written assembler subroutines are responsible for saving and restoring these registers when necessary. The registers must be saved even when the compiler has assigned them for argument passing.

The temporary register, r0, can be clobbered by C functions, but they are saved by interrupt handlers.

The compiler assumes that the Zero register, r1, always contains the value zero. It can be used in hand-written assembly routine for intermediate values, but must be cleared after use (e.g using `clr r1`). Be aware that multiplication instructions return their result in the r1-r0 register pair. Interrupt handlers save and clear r1 on entry, and restore r1 on exit (in case it was non-zero).

All registers that have been used by an interrupt routine are save and restored by the interrupt routine (see 4.8.4 Context Switching).

The registers that have a dedicated function throughout the program are tabulated below.

Table 4-7. Registers with Dedicated Use

Register name	Applicable devices
r0	Temporary register
r1 (__zero_reg__)	Zero register (holds 0 value)
r28, r29	Frame pointer (Y pointer)

4.7 Functions

Functions are written in the usual way, in accordance with the C language. Implementation-specific features associated with functions are discussed in following sections.

4.7.1 Function Specifiers

Aside from the standard C specifier, static, which affects the linkage of the function, there are several non-standard function specifiers, which are described in the following sections.

4.7.1.1 Inline Specifier

The `inline` function specifier is a recommendation that the compiler replace calls to the specified function with the function's body, if possible.

The following is an example of a function which has been made a candidate for inlining.

```
inline int combine(int x, int y) {
  return 2 * x - y;
}
```

All function calls to a function that was in-lined by the compiler will be encoded as if the call was replaced with the body of the called function. This is performed at the assembly code level. In-lining will only take place if the optimizers are enabled (level 1 or higher), but you can ask that a function always be in-lined by using the `always_inline` attribute.

If inlining takes place, this will increase the program's execution speed, since the call and return sequences associated with the call will be eliminated. Code size can be reduced if the assembly code associated with the body of the in-lined function is very small, but code size can increase if the body of the in-lined function is larger than the call/return sequence it replaces. You should only consider this specifier for functions which generate small amounts of assembly code. Note that the amount of C code in the body of a function is not a good indicator of the size of the assembly code that it generates.

There are several reasons why the compiler might not in-line a function which has been specified, so your code should not make any assumption about whether inlining took place. The `-Winline` option can be used to warn you when a function marked in-line could not be substituted, and gives the reason for the failure.

4.7.1.2 Nopa Specifier

The `__nopa` specifier expands to `__attribute__((nopa,noinline))`, disabling procedural abstraction as well as inlining for the associated function. This ensures that inlined code is then not subject to procedural abstraction.

4.7.1.3 Section Specifier

The `__section(`*`section`*`)` specifier allocates the function to a user-nominated section rather than allowing the compiler to place it in a default section. See 4.15 Changing and Linking the Allocated Section for full information on the use of this specifier.

4.7.2 Function Attributes

The compiler keyword `__attribute__()` allows you to specify special attributes of functions. Place inside the parentheses following this keyword a comma-separated list of the relevant attributes, for example:

```
__attribute__((weak))
```


The attribute can be placed anywhere in the object's definition, but is usually placed as in the following example.

```
char __attribute__((weak)) input(int mode);
char input(int mode) __attribute__((weak));
```

4.7.2.1 Naked Attribute

The `naked` attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue or epilogue sequences generated by the compiler. Only basic `asm()` statements can safely be included in naked functions. Do not use extended `asm()` or a mixture of basic `asm()` and C code, as they are not supported.

4.7.2.2 Nopa Attribute

The `nopa` attribute indicates that proceedural abstraction optimizations should not be applied to the function.

4.7.2.3 Os_main/os_task Attribute

On AVR, functions with the `OS_main` or `OS_task` attribute do not save/restore any call-saved register in their prologue/epilogue.

The `OS_main` attribute can be used when there is guarantee that interrupts are disabled at the time when the function is entered. This saves resources when the stack pointer has to be changed to set up a frame for local variables.

The `OS_task` attribute can be used when there is no guarantee that interrupts are disabled at that time when the function is entered like for, e.g. task functions in a multi-threading operating system. In that case, changing the stack pointer register is guarded by save/clear/restore of the global interrupt enable flag.

The differences to the naked function attribute are:

- naked functions do not have a return instruction whereas `OS_main` and `OS_task` functions have a `ret` or `reti` return instruction.
- naked functions do not set up a frame for local variables or a frame pointer whereas `OS_main` and `OS_task` do this as needed.

4.7.2.4 Weak Attribute

The `weak` attribute causes the declaration to be emitted as a weak symbol. A weak symbol indicates that if a global version of the same symbol is available, that version should be used instead.

When the `weak` attribute is applied to a reference to an external symbol, the symbol is not required for linking. For example:

```
extern int __attribute__((weak)) s;
int foo(void) {
  if (&s)
    return s;
  return 0;   /* possibly some other value */
}
```

In the above program, if `s` is not defined by some other module, the program will still link but `s` will not be given an address. The conditional verifies that s has been defined (and returns its value if it has). Otherwise '0' is returned. There are many uses for this feature, mostly to provide generic code that can link with an optional library.

4.7.3 Allocation of Executable Code

Code associated with C functions is always placed in the `.text` section, which is linked into the program memory of the target device.

4.7.4 Changing the Default Function Allocation

You can change the default memory allocation of functions by either:

- Making functions absolute
- Placing functions in their own section and linking that section

The easiest method to explicitly place individual functions at a known address is to make them absolute by using the `__at()` construct in a similar fashion to that used with absolute variables. The CCI must be enabled for this syntax to be accepted and `<xc.h>` must be included

The compiler will issue a warning if code associated with an absolute function overlaps with code from other absolute functions. The compiler will not locate code associated with ordinary functions over the top of absolute functions.

The following example of an absolute function will place the function at address 400h:

```
int __at(0x400) mach_status(int mode)
{
  /* function body */
}
```

If this construct is used with interrupt functions, it will only affect the position of the code associated with the interrupt function body. The interrupt context switch code associated with the interrupt vector will not be relocated.

Functions can be allocated to a user-defined psect using the `__section()` specifier (see Section 3.15.2 “Changing and Linking the Allocated Section”) so that this new section can then be linked at the required location. This method is the most flexible and allows functions to be placed at a fixed address, after other section, or anywhere in an address range. As with absolute functions, when used with interrupt functions, it will only affect the position of the interrupt function body.

Regardless of how a function is located, take care choosing its address. If possible, avoid fragmenting memory and increasing the possibility of linker errors.

4.7.5 Function Size Limits

For all devices, the code generated for a regular function is limited only by the available program memory. See 4.7.3 Allocation of Executable Code for more details.

4.7.6 Function Parameters

MPLAB XC8 uses a fixed convention to pass arguments to a function. The method used to pass the arguments depends on the size and number of arguments involved.

Note: The names “argument” and “parameter” are often used interchangeably, but typically an argument is the value that is passed to the function and a parameter is the variable defined by the function to store the argument.

Arguments are passed to functions via registers and consume as many register as required to hold the object. However, registers are allocated in pairs, thus there will be at least two registers allocated to each argument, even if the argument is a single byte. This make more efficient use of the AVR instruction set.

The first register pair available is r24-r25 and lower register pairs are considered after that down to register r8. For example, if a function with the prototype:

```
int map(unsigned long a, char b);
```

is called, the argument for the four-byte parameter `a` will be passed in registers r22 thru r25, with r22 holding the least significant byte and r25 holding the most significant byte; and the argument for parameter `b` will be assigned to registers r20 and r21.

If there are further arguments to pass once all the available registers have been assigned, they are passed on the stack.

Arguments to functions with variable argument lists (`printf()` etc.) are all passed on stack.

4.7.7 Function Return Values

A function’s return value is usually returned in a register.

A byte-sized return value is returned in r24. Multi-byte return values are return in as many registers as required, with the highest register being r25. Thus, a 16-bit value is returned in r24-r25, a 32-bit value in r22-r25, etc.

4.7.8 Calling Functions

Functions are called using an `rcall` instruction. If your target device has more than 8kB of program memory, it will use a larger call instruction to be able to reach any function, regardless of where it is located in program memory.

If you can guarantee that all call destinations are within range of the `rcall` instruction, the shorter form of call can be requested by using the `-mshort-calls` option (see 3.6.1.9 Short-calls Option).

4.8 Interrupts

The MPLAB XC8 compiler incorporates features allowing interrupts to be fully handled from C code. Interrupt functions are often called Interrupt Service Routines, or ISRs.

The following are the general steps you need to follow to use interrupts. More detail about these steps is provided in the sections that follow.

- Write as many interrupt functions as required. Consider one or more additional functions to handle accidental triggering of unused interrupt sources.
- At the appropriate point in your code, enable the interrupt sources required.
- At the appropriate point in your code, enable the global interrupt flag.

Interrupt functions must not be called directly from C code (due to the different return instruction used), but can call other functions, such as user-defined and library functions.

Interrupt code is the name given to any code that executes as a result of an interrupt occurring, including functions called from the ISR and library code. Interrupt code completes at the point where the corresponding return from interrupt instruction is executed. This contrasts with main-line code, which, for a freestanding application, is usually the main part of the program that executes after Reset.

4.8.1 Writing an Interrupt Service Routine

Observe the following guidelines when writing an ordinary ISR.

- Write each ISR prototype by enabling the CCI (3.6.3.3 Ext Option) and using the `__interrupt()` specifier. This will create a function with the appropriate name, prototype, and attributes.
- If necessary, clear the relevant interrupt flag once the source has been processed, although typically this is not required.
- Only re-enable interrupts inside the ISR body if absolutely necessary. Interrupt are re-enabled automatically when the ISR returns.
- Keep the ISR as short and as simple as possible. Complex code will typically use more registers that will increase the size of the context switch code.

The compiler will process interrupt functions differently to other functions, generating code to save and restore any registers used by the function and using a special instruction to return.

The hardware globally disables interrupts when an interrupt is executed.

Usually, each interrupt source has a corresponding interrupt flag bit, accessible in a control register. When set, these flags indicate that the specified interrupt condition has been met. Interrupt flags are often cleared in the course of processing the interrupt, either when the handler is invoked or by reading a particular hardware register; however, there are a few instances when the flag must be cleared manually by code. Failure to do so might result in the interrupt triggering again as soon as the current ISR returns.

The flag bits in the SFRs have a unique property whereby they are cleared by writing a logic one to them. To take advantage of this property, you should write directly to the register rather than use any instruction sequence that might perform a read-modify-write. Thus to clear the TOV0 timer overflow flag in the TC0 interrupt flag register, use the following code:

```
TIFR = _BV(TOV0);
```

which is guaranteed to clear the TOV0 bit and leave the remaining bits untouched.

An example of an interrupt function is shown below.

```
void __interrupt(SPI_STC_vect_num) spi_Isr(void) {
  process(SPI_SlaveReceive());
  return;
}
```

Note that the argument to the `__interrupt()` specifier is a vector number, which are available as macros ending with `vect_num` once you have included `<xc.h>` in your program.

More complex interrupt function definitions can be created using macros and attributes defined by `<avr/interrupt.h>` and which are shown in the following examples.

If there is no code to be executed for an interrupt source but you want to ensure that the program will continue normal operation should the interrupt unexpectedly trigger, then you can create an empty ISR using the `EMPTY_INTERRUPT()` macro and an interrupt source argument.

```
#include <avr/interrupt.h>
EMPTY_INTERRUPT(INT2_vect);
```

The special interrupt source symbol, `BADISR_vect`, can be used to define a function that can process any otherwise undefined interrupts. Without this function defined, an undefined interrupt will trigger a device reset. For example:

```
void ISR(BADISR_vect) {
  // place code to process undefined interrupts here
  return;
}
```

If you wish to allow nested interrupts you can manually add an in-line `sei` instruction to your ISR to re-enable the global interrupt flag; however, there is an argument you can use with the `ISR()` macro to have this instruction added by the compiler to the beginning of the interrupt routine. For example:

```
void ISR(IO_PINS_vect, ISR_NOBLOCK)
{ ... }
```

If one ISR is to be used with more than one interrupt vector, then you can define that ISR in the usual way for one vector then reuse that ISR for other vector definitions using the `ISR_ALIASOF()` argument.

```
void __interrupt(PCINT0_vect_num)
{ ... }
void ISR(PCINT1_vect, ISR_ALIASOF(PCINT0_vect));
```

In some circumstances, the compiler-generated context switch code executed by the ISR might not be optimal. In such situations, you can request that the compiler omit this context switch code and supply this yourself. This can be done using the `ISR_NAKED` argument, as shown in this example.

```
void ISR(TIMER1_OVF_vect, ISR_NAKED)
{
  PORTB |= _BV(0); // results in SBI which does not affect SREG
  reti();
}
```

Note that the compiler will not generate any context switch code, including the final return from interrupt instruction, so you must write any relevant switching code and the `reti` instruction. The SREG register must be manually saved if it is modified by the ISR, and the compiler-implied assumption of `__zero_reg__` always being 0 could be wrong, for example when an interrupt occurs right after a `mul` instruction.

4.8.2 Changing the Default Interrupt Function Allocation

You can use the `__at()` specifier (see 4.7.4 Changing the Default Function Allocation) if you want to move the interrupt function itself. This does not alter the position of the vector table, but the appropriate table entry will still point to the correct address of the shifted function.

4.8.3 Specifying the Interrupt Vector

The process of populating the interrupt vector locations is fully automatic, provided you define interrupt functions (as shown in 4.8.1 Writing an Interrupt Service Routine). The compiler will automatically link each ISR entry point to the appropriate fixed vector location.

The location of the interrupt vectors cannot be changed at runtime, nor can you change the code linked to the vector. That is, you cannot have alternate interrupt functions for the same vector and select which will be active during program execution. An error will result if there are more than one interrupt function defined for the same vector.

Interrupt vectors that have not been specified explicitly in the project can be assigned a default function address by defining an interrupt function that uses `BADISR_vect` as its vector.

4.8.4 Context Switching

The compiler will automatically link code into your project which saves the current status when an interrupt occurs, and restores this status when the interrupt returns.

All call-used registers will be saved in interrupt code generated by the compiler. This is the context save or context switch code.

Any objects saved by software are automatically restored by software before the interrupt function returns. The order of restoration is the reverse of that used when context is saved.

4.8.5 Enabling Interrupts

Two macros are available, once you have included `<xc.h>`, that control the masking of all available interrupts. These macros are `ei()`, which enable or unmask all interrupts, and `di()`, which disable or mask all interrupts.

On all devices, they affect the I bit in the status register, SREG. These macros should be used once the appropriate interrupt enable bits for the interrupts that are required in a program have been enabled.

For example:

```
TIMSK = _BV(TOIE1);
ei(); // enable all interrupts
// ...
di(); // disable all interrupts
```

Note: Typically you should not re-enable interrupts inside the interrupt function itself. Interrupts are automatically re-enabled by hardware on execution of the `reti` instruction. Re-enabling interrupts inside an interrupt function can result in code failure if not correctly handled.

In addition to globally enabling interrupts, each device's particular interrupt needs to be enabled separately if interrupts for this device are desired. While some devices maintain their interrupt enable bit inside the device's register set, external and timer interrupts have system-wide configuration registers.

To modify the TIMSK register, use `timer_enable_int(`*`ints`*`)`. The value you pass via *`ints`* should be the bit mask for the interrupt enable bit and is device specific.

To modify the GIMSK register (or EIMSK register if using an AVR Mega device or GICR register for others) use `enable_external_int(`*`mask`*`)`. This macro is unavailable if neither if these registers are defined.

For example:

```
// Enable timer 1 overflow interrupts
timer_enable_int(_BV(TOIE1));
// Do some work...
// Disable all timer interrupts
timer_enable_int(0);
```

4.8.6 Accessing Objects From Interrupt Routines

Reading or writing objects from interrupt routines can be unsafe if other functions access these same objects.

It is recommended that you explicitly mark objects accessed in interrupt and main-line code using the `volatile` specifier (see 4.3.8.2 Volatile Type Qualifier). The compiler will restrict the optimizations performed on `volatile` objects.

Even when objects are marked as `volatile`, the compiler cannot guarantee that they will be accessed atomically. This is particularly true of operations on multi-byte objects.

Interrupts should be disabled around any main-line code that modifies an object that is used by interrupt functions, unless you can guarantee that the access is atomic. Macros are provided in `<avr/atomic.h>` to assist you access these objects.

4.9 Main, Runtime Startup and Reset

Coming out of Reset, your program will first execute runtime startup code added by the compiler, then control is transfered to the function `main()`. This sequence is described in the following sections.

4.9.1 The main Function

The identifier `main` is special. You must always have one, and only one, function called `main()` in your programs. This is the first C function to execute in your program.

Since your program is not called by a host, the compiler inserts special code at the end of `main()`, which is executed if this function ends, i.e., a return statement inside `main()` is executed, or code execution reaches the `main()`'s terminating right brace. This special code causes execution to jump to address 0, the Reset vector for all 8-bit AVR devices. This essentially performs a software Reset. Note that the state of registers after a software Reset can be different to that after a hardware Reset.

It is recommended that the `main()` function does not end. Add a loop construct (such as a `while(1)`) that will never terminate either around your code in `main()` or at the end of your code, so that execution of the function will never terminate. For example,

```
int main(void)
{
    // your code goes here
    // finished that, now just wait for interrupts
    while(1)
        continue;
}
```

4.9.2 Runtime Startup Code

A C program requires certain objects to be initialized and the device to be in a particular state before it can begin execution of its function `main()`. It is the job of the runtime startup code to perform these tasks, specifically (and in no particular order):

- Initialization of static storage duration objects assigned a value when defined
- Clearing of non-initialized static storage duration objects
- General set up of registers or device state
- Calling the `main()` function

One of several runtime startup code object files which provide the runtime startup code is linked into your program.

The runtime startup code assumes that the device has just come out of Reset and that registers will be holding their power-on-reset value. Note that when the watchdog or `RST_SWRST_bm` resets the device, the registers will be reset to their known, default settings; whereas, jumping to the reset vector will not change the registers and they will be left in their previous state.

The sections used to hold the runtime startup code are tabulated below.

Table 4-8. Runtime Startup Code Sections used Before main

Section name	Description
`.init0`	Weakly bound to `__init()`, see 4.9.3 The Powerup Routine. If user defines `__init()`, it will be jumped into immediately after a reset.
`.init1`	Unused. User definable.
`.init2`	In C programs, weakly bound to code which initializes the stack and clears `__zero_reg__` (r1).

...........continued	
Section name	Description
`.init3`	Unused. User definable.
`.init4`	This section contains the code from `libgcc.a` that copies the contents of `.data` from the program to data memory, as well as the code to clear the `.bss` section.
`.init5`	Unused. User definable.
`.init6`	Unused for C programs.
`.init7`	Unused. User definable.
`.init8`	Unused. User definable.
`.init9`	Calls the `main()` function.

The `main()` function returns to code that is also provided by the runtime startup code. You can have code executed after `main()` has returned by placing code in the sections tabulated below.

Table 4-9. Runtime Startup Code Sections used After main

Section name	Description
`.fini9`	Unused. User definable.
`.fini8`	Unused. User definable.
`.fini7`	In C programs, weakly bound to initialize the stack, and to clear `__zero_reg__` (r1).
`.fini6`	Unused for C program.
`.fini5`	Unused. User definable.
`.fini4`	Unused. User definable.
`.fini3`	Unused for C programs.
`.fini2`	Unused. User definable.
`.fini1`	Unused. User definable.
`.fini0`	Goes into an infinite loop after program termination and completion of any `_exit()` code (code in the `.fini9` thru `.fini1` sections).

4.9.2.1 Initialization Of Objects

One task of the runtime startup code is to ensure that any static storage duration objects contain their initial value before the program begins execution. A case in point would be input in the following example.

```
int input = 88;
```

In the code above, the initial value (`0x88`) will be stored as data in program memory and will be copied to the memory reserved for `input` by the runtime startup code. For efficiency, initial values are stored as blocks of data and copied by loops.

Absolute variables are never initialized and must be explicitly assigned a value if that is required for correct program execution.

Since auto objects are dynamically created, they require code to be positioned in the function in which they are defined to perform their initialization and are not considered by the runtime startup code.

Note: Initialized auto variables can impact on code performance, particularly if the objects are large in size. Consider using `static` local objects instead.

Objects whose content should be preserved over a Reset should be marked with the `__persistent` attribute (see 4.3.10.8 Persistent Attribute). Such objects are linked in a different area of memory and are not altered by the runtime startup code.

4.9.2.2 Clearing Objects

Those objects with static storage duration which are not assigned a value must be cleared before the `main()` function begins by the runtime startup code, for example.

```
int output;
```

The runtime startup code will clear all the memory locations occupied by uninitialized objects so they will contain zero before `main()` is executed.

Absolute variables are never cleared and must be explicitly assigned a value of zero if that is required for correct program execution.

Objects whose contents should be preserved over a Reset should be qualified with `__persistent`. Such objects are linked at a different area of memory and are not altered by the runtime startup code.

4.9.3 The Powerup Routine

Some hardware configurations require special initialization, often within the first few instruction cycles after Reset. To achieve this you can have your own powerup routine executed during the runtime startup code.

Provided you write the required code in one of the `.init`*n* sections used by the runtime startup code, the compiler will take care of linking your code to the appropriate location without any need for you to adjust the linker scripts. These sections are listed in 4.9.2 Runtime Startup Code.

For example, the following is a small assembly sequence that is placed in the `.init1` section and is executed soon after reset and before `main()` is called.

```
#include <avr/io.h>
 .section .init1,"ax",@progbits
 ldi r16,_BV(SRE) | _BV(SRW)
 out _SFR_IO_ADDR(MCUCR),r16
```

Place this routine in an assembly source file, assemble it, and link the output with other files in your program.

Remember that code in these sections is executed before all the runtime startup code has been executed, so there is no usable stack and the `__zero_reg__` (r1) might not have been initialized. It is best to leave `__stack` at its default value (at the end of internal SRAM since this is faster and required on some devices, like the ATmega161 to work around known errata), and use the `-Wl,-Tdata,0x801100` option to start the data section above the stack.

4.10 Libraries

The MPLAB XC8 C Compiler provided libraries of standard C functions. In addition, you can create your own libraries from source code you have written.

4.10.1 Standard Libraries

The C standard libraries contain a standardized collection of functions, such as string and math routines. These functions are listed in 7. Library Functions.

4.10.2 User-Defined Libraries

User-defined libraries can be created and linked in with your program. Library files are easier to manage than many source files, and can result in faster compilation times. Libraries must, however, be compatible with the target device and options for a particular project. Several versions of a library might need to be created and maintained to allow it to be used for different projects.

Libraries can be created using the librarian, `avr-ar`, (see 5.1 Archiver/Librarian).

Once built, user-defined libraries can be used on the command line along with the source files or added to the Libraries folder in your MPLAB X IDE project.

Library files specified on the command line are scanned for unresolved symbol before the C standard libraries (but after any project modules), so their content can redefine anything that is defined in the C standard libraries.

4.10.3 Using Library Routines

Library functions and objects that have been referenced will be automatically linked into your program, provided the library file is part of your project. The use of a function from one library file will not include any other functions from that library.

Your program will require declarations for any library functions or symbols it uses. Standard libraries come with standard C headers (`.h` files), which can be included into your source files. See your favorite C text book or 7. Library Functions for the header that corresponds to each library function. Typically you would write library headers if you create your own library files.

Header files are not library files. Library files contain precompiled code, typically functions and variable definitions; header files provide declarations (as opposed to definitions) for those functions, variables and types in the library. Headers can also define preprocessor macros.

4.11 Mixing C and Assembly Code

Assembly language can be mixed with C code using two different techniques:

- Assembly code placed in separate assembly source modules.
- Assembly code placed inline with C code.

Note: The more assembly code a project contains, the more difficult and time consuming will be its maintenance. Assembly code might need revision if the compiler is updated due to differences in the way the updated compiler may work. These factors do not affect code written in C.
If assembly must be added, it is preferable to write this as a self-contained routine in a separate assembly module, rather than in-lining it in C code.

4.11.1 Integrating Assembly Language Modules

Entire functions can be coded in assembly language as separate `.s` (or `.S`) source files included into your project. They will be assembled and combined into the output image by the linker.

The following are guidelines that must be adhered to when writing a C-callable assembly routine.

- Include the `<xc.h>` header file in your code. If this is included using `#include`, ensure the extension used by the source file is `.S` to ensure the file is preprocessed.
- Select or define a suitable section for the executable assembly code (see 4.14.1 Compiler-Generated Sections for an introductory guide).
- Select a name (label) for the routine
- Ensure that the routine's label is accessible from other modules
- Use macros like `_SFR_IO_ADDR` to obtain the correct SFR address to use with instructions that can access the IO memory space.
- Select an appropriate C-equivalent prototype for the routine on which argument passing can be modeled.
- If values need to be passed to or returned from the routine, use the appropriate registers to passed the arguments.

The following example shows an assembly routine for an atmega103 device that takes an `int` parameter, adds this to the content of PORTD, and returns this as an `int`.

```
#include <xc.h>
.section .text
.global plus ; allow routine to be externally used
plus:
  ; int parameter in r24/5
  in r18, _SFR_IO_ADDR(PORTD) ; read PORTD
  add r24, r18 ; add to parameter
  adc r25, r1 ; add zero to MSB
  ; parameter registers are also the return location, so ready to return
```

```
  ret
.end
```

The code has been placed in a .text section, so it will be automatically placed in the area of memory set aside for code without you having to adjust the default linker options.

The _SFR_IO_ADDR macro has been used to ensure that the correct address was specified to instructions that read the IO memory space.

Because the C preprocessor #include directive and preprocessor macros were used, the assembly file must be preprocessed to ensure it uses a .S extension when compiled.

To call an assembly routine from C code, a declaration for the routine must be provided. Here is a C code snippet that declares the operation of the assembler routine, then calls the routine.

```
// declare the assembly routine so it can be correctly called
extern int plus(int);
void main(void) {
  volatile unsigned int result;
  result = plus(0x55); // call the assembly routine
}
```

4.11.2 In-line Assembly

Assembly instructions can be directly embedded in-line into C code using the statement asm();. In-line assembly has two forms: simple and extended.

In the simple form, the assembler instruction is written using the syntax:

```
asm("instruction");
```

where instruction is a valid assembly-language construct, for example:

```
asm("sei");
```

You can write several instructions in the one string, but you should put each instruction on a new line and use linefeed and tab characters to ensure they are properly formatted in the assembly listing file.

```
asm ("nop\n\t"
 "nop\n\t"
 "nop\n\t"
 "nop\n\t");
```

In an extended assembler instruction using asm(), the operands of the instruction are specified using C expressions. The extended syntax, discussed in the following sections, has the general form:

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
[ : [ "constraint"(input-operand) [ , ... ] ]
[ "clobber" [ , ... ] ]
 ] ]);
```

For example,

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
```

The template specifies the instruction mnemonic and optional placeholders for the input and output operands, specified by a percent sign followed by a single digit and which are described in the following section. The compiler replaces these and other tokens in the template that refer to inputs, outputs, and goto labels, then outputs the resulting string to the assembler.

4.11.2.1 Input and Output Operands

Following the template is a comma-separated list of zero or more output operands, which indicate the names of C objects modified by the assembly code and input operands, which make values from C variables and expressions available to the assembly code.

Each operands has several components, described by:

```
[ [asmSymbolicName] ] constraint (Cexpression)
```

where *asmSymbolicName* is an optional symbolic name for the operand, *constraint* is string specifying constraints on the placement of the operand, and *Cexpression* is the C variable or expression to be used by the operand and which is enclosed in parentheses.

The first (left-most) output operand is numbered 0, any subsequent output operands are numbered one higher than the operand before it, with input operands being numbered in the same way.

The supported constraint letters are tabulated below (see table later in this section for operand modifiers).

Table 4-10. Input and Output Operand Constraints

Letter	Constraint	Range
a	Simple upper registers	r16 to r23
b	Base pointer registers pairs	r28 to r32 (Y, Z)
d	Upper register	r16 to r31
e	Pointer register pairs	r26 to r31 (X, Y, Z)
l	Lower registers	r0 to r15
q	Stack pointer register	SPH:SPL
r	Any register	r0 to r31
t	Temporary register	r0
w	Special upper register pairs usable in `adiw` instruction	r24, r26, r28, r30
x	Pointer register pair X	r27:r26 (X)
y	Pointer register pair Y	r29:r28 (Y)
z	Pointer register pair Z	r31:r30 (Z)
G	Floating point constant	0.0
I	6-bit positive integer constant	0 to 63
J	6-bit negative integer constant	-63 to 0
K	Integer constant	2
L	Integer constant	0
M	8-bit integer constant	0 to 255
N	Integer constant	-1
O	Integer constant	8, 16, 24
P	Integer constant	1
Q	Memory address based on Y or Z pointer with displacement	
Cm2	Integer constant	-2
C0*n*	Integer constant, where *n* ranges from 0 to 7	n
Ca*n*	n-byte integer constant that allows AND without clobber register, where *n* ranges from 2 to 4	
Co*n*	n-byte integer constant that allows OR without clobber register, where *n* ranges from 2 to 4	

Letter	Constraint	Range
`Cxn`	n-byte integer constant that allows XOR without clobber register, where *n* ranges from 2 to 4	
`Csp`	Integer constant	-6 to 6
`Cxf`	4-byte integer constant with at least one 0xF nibble	
`C0f`	4-byte integer constant with no 0xF nibbles	
`Ynn`	Fixed-point constant known at compile time	
`Y0n`	Fixed-point or integer constant, where *n* ranges from 0 to 2	n
`Ymn`	Fixed-point or integer constant, where *n* ranges from 1 to 2	-n
`YIJ`	Fixed-point or integer constant	-0x3F to 0x3F

The constraint you choose should match the registers or constants that are appropriate for the AVR instruction operand. The compiler will check the constraint against your C expression; however, if the wrong constraint is used, there is the possibility of code failing at runtime. For example, if you specify the constraint `r` with an `ori` instruction, then the compiler is free to select any register (r0 thru r31) for that operand. This will fail, if the compiler chooses a register in the range r2 to r15. The correct constraint in this case is `d`. On the other hand, if you use the constraint `M`, the compiler will make sure that you only use an 8-bit immediate value operand.

The table below shows all the AVR assembler mnemonics that require operands and the relevant constraints for each of those operands.

Table 4-11. Instructions and Operand Constraints

Mnemonic	Constraints	Mnemonic	Constraints
adc	r,r	add	r,r
adiw	w,I	and	r,r
andi	d,M	asr	r
bclr	I	bld	r,I
brbc	I,label	brbs	I,label
bset	r,I	bst	r,I
cbi	I,I	cbr	d,I
com	r	cp	r,r
cpc	r,r	cpi	d,M
cpse	r,r	dec	r
elpm	t,z	eor	r,r
in	r,I	inc	r
ld	r,e	ldd	r,b
ldi	d,M	lds	r,label
lpm	t,z	lsl	r
lsr	r	mov	r,r
movw	r,r	mul	r,r
neg	r	or	r,r

...........continued

Mnemonic	Constraints	Mnemonic	Constraints
ori	d,M	out	I,r
pop	r	push	r
rol	r	ror	r
sbc	r,r	sbci	d,M
sbi	I,I	sbic	I,I
sbiw	w,I	sbr	d,M
sbrc	r,I	sbrs	r,I
ser	d	st	e,r
std	b,r	sts	label,r
sub	r,r	subi	d,M
swap	r		

Constraint characters may be prepended by a single constraint modifier. Constraints without a modifier specify read-only operands. The constraint modifiers are tabulated below.

Table 4-12. Input and Output Constraint Modifiers

Letter	Constraint
=	Write-only operand, usually used for all output operands.
+	Read-write operand
&	Register should be used for output only

So, in the example:

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
```

the assembler instruction is defined by the template, `"in %0, %1"`. The `%0` token refers to the first output operand, `"=r" (value)`, and `%1` refers to the first input operand, `"I" (_SFR_IO_ADDR(PORTD))`. No clobbered registers were indicated in this example.

The compiler might encode the above in-line assembly as follows:

```
lds r24,value
/* #APP */
in r24, 12
/* #NOAPP */
sts value,r24
```

The comments have been added by the compiler to inform the assembler that the enclosed instruction was hand-written. In this example, the compiler selected register r24 for storage of the value read from PORTD; however, it might not explicitly load or store the value, nor include your assembler code at all, based on the compiler's optimization strategy. For example, if you never use the variable value in the remaining part of the C program, the compiler could remove your in-line assembly code unless you switch off the optimizers. To avoid this, you can add the `volatile` attribute to the `asm()` statement, as shown below:

```
asm volatile(“in %0, %1” : “=r” (value) : “I” (_SFR_IO_ADDR(PORTD)));
```

Operands can be given names, if desired. The name is prepended in brackets to the constraints in the operand list and references to the named operand use the bracketed name instead of a number after the % sign. Thus, the above example could also be written as

```
asm("in %[retval], %[port]" :
 [retval] "=r" (value) :
 [port] "I" (_SFR_IO_ADDR(PORTD)) );
```

The clobber list is primarily used to tell the compiler about modifications done by the assembler code. This section of the statement can be omitted, but all other sections are required. Use the delimiting colons, but leave the operand field empty if there is no input or output used, for example:

```
asm volatile("cli"::);
```

Output operands must be write-only and the C expression result must be an lvalue, i.e., be valid on the left side of an assignment. Note, that the compiler will not check if the operands are of a reasonable type for the kind of operation used in the assembler instructions. Input operands are read-only.

In cases where you need the same operand for input and output, read-write operands are not supported, but it is possible to indicate which operand's register to use as the input register by a single digit in the constraint string. Here is an example:

```
asm volatile("swap %0" : "=r" (value) : "0" (value));
```

This statement will swap the nibbles of an 8-bit variable named value. Constraint `"0"` tells the compiler, to use the same input register used by the first operand. Note, however, that this doesn't automatically imply the reverse case.

The compiler may choose the same registers for input and output, even if not told to do so. This can be an issue if the output operand is modified by the assembler code before the input operand is used. In the situation where your code depends on different registers used for input and output operands, you must use the constraint modifier, `&`, with the output operand, as shown in the following example.

```
asm volatile("in %0,%1" "\n\t"
 "out %1,%2" "\n\t"
 : "=&r" (result)
 : "I" (_SFR_IO_ADDR(port)), "r" (source)
 );
```

Here, a value is read from a port and then a value is written to the same port. If the compiler chooses the same register for input and output, then the output value will be clobbered by the first assembler instruction; however, the use of the `&` constraint modifier prevents the compiler from selecting any register for the output value that is also used for any of the input operands.

Here is another example that swaps the high and low byte of a 16-bit value:

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
 "mov %A0, %B0" "\n\t"
 "mov %B0, __tmp_reg__" "\n\t"
 : "=r" (value)
 : "0" (value)
 );
```

Notice the usage of register `__tmp_reg__`, which you can use without having to save its content. The letters `A` and `B`, used in the tokens representing the instruction operands refer to byte components of a multi-byte register, `A` referring to the least significant byte, `B` the next most significant byte, etc.

The following example, which swaps bytes of a 32-bit value, uses the `C` and `D` components of a 4 byte quantity, and rather than list the same operand as both input and output operand (via `"0"` as the input operand constraint), it can also be declared as a read-write operand by using `"+r"` as the output constraint.

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
 "mov %A0, %D0" "\n\t"
 "mov %D0, __tmp_reg__" "\n\t"
 "mov __tmp_reg__, %B0" "\n\t"
 "mov %B0, %C0" "\n\t"
```

```
 "mov %C0, __tmp_reg__" "\n\t"
 : "+r" (value)
 );
```

If operands do not fit into a single register, the compiler will automatically assign enough registers to hold the entire operand. This also implies, that it is often necessary to cast the type of an input operand to the desired size.

If an input operand constraint indicates a pointer register pair, such as `"e" (ptr)`, and the compiler selects register Z (r30:r31), then you must use `%a0` (lower case `a`) to refer to the Z register, when used in a context like:

```
ld r24,Z
```

4.11.2.2 Clobber Operand

The list of clobbered registers is optional; however, if the instruction modifies registers that are not specified as operands, you need to inform the compiler of these changes.

Typically you can arrange the assembly so that you do not need to specify what has been clobbered. Indicating that a register has been clobbered will force the compiler to store their values before and reload them after your assembly instructions and will limit the ability of the compiler to optimize your code.

The following example will perform an atomic increment. It disables the interrupts then increments an 8-bit value pointed to by a pointer variable. Note, that a pointer is used because the incremented value needs to be stored before the interrupts are enabled.

```
asm volatile(
 "cli" "\n\t"
 "ld r24, %a0" "\n\t"
 "inc r24" "\n\t"
 "st %a0, r24" "\n\t"
 "sei" "\n\t"
 :
 : "e" (ptr)
 : "r24"
);
```

The compiler might produce the following code for the above:

```
cli
ld r24, Z
inc r24
st Z, r24
sei
```

To have this sequence avoid clobbering register r24, make use of the special temporary register `__tmp_reg__` defined by the compiler.

```
asm volatile(
 "cli" "\n\t"
 "ld __tmp_reg__, %a0" "\n\t"
 "inc __tmp_reg__" "\n\t"
 "st %a0, __tmp_reg__" "\n\t"
 "sei" "\n\t"
 :
 : "e" (ptr)
);
```

The compiler will always reload the temporary register when it is needed.

The above code unconditionally re-enables the interrupts, which may not be desirable. To make the code more versatile, the current status can be stored in a register selected by the compiler.

```
{
  uint8_t s;
  asm volatile(
  "in %0, __SREG__" "\n\t"
  "cli" "\n\t"
  "ld __tmp_reg__, %a1" "\n\t"
  "inc __tmp_reg__" "\n\t"
```

```
  "st %a1, __tmp_reg__" "\n\t"
  "out __SREG__, %0" "\n\t"
  : "=&r" (s)
  : "e" (ptr)
  );
}
```

The assembler code here modifies the variable, that `ptr` points to, so the definition of `ptr` should indicate that its target can change unexpected, using the `volatile` specifier, for example:

```
volatile uint8_t *ptr;
```

The special clobber memory informs the compiler that the assembler code may modify any memory location. It forces the compiler to update all variables for which the contents are currently held in a register before executing the assembler code.

When you use a memory clobber with an assembly instruction, it ensures that all prior accesses to `volatile` objects are complete before the instruction executes, and that execution of `volatile` accesses after the instruction have not commenced. However, it does not prevent the compiler from moving non-`volatile`-related instructions across the barrier created by the memory clobber instruction, as such instructions might be those that enable or disable interrupts.

4.11.3 Interaction between Assembly and C Code

MPLAB XC8 C Compiler incorporates several features designed to allow C code to obey requirements of user-defined assembly code. There are also precautions that must be followed to ensure that assembly code does not interfere with the assembly generated from C code.

4.11.3.1 Equivalent Assembly Symbols

By default AVR-GCC uses the same symbolic names of functions or objects in C and assembler code. There is no leading underscore character prepended to a C language's symbol in assembly code.

You can specify a different name for the assembler code by using a special form of the `asm()` statement:

```
unsigned long value asm(“clock”) = 3686400;
```

This statement instructs the compiler to use the symbol name `clock` rather than `value`. This makes sense only for objects with static storage duration, because stack-based objects do not have symbolic names in the assembler code and these can be cached in registers.

With the compiler you can specify the use of a specific register:

```
void Count(void)
{
 register unsigned char counter asm("r3");
 // ... some code...
 asm volatile("clr r3");
 // ... more code...
}
```

The assembler instruction, `clr r3`, will clear the variable counter. The compiler will not completely reserve the specified register, and it might be re-used. The compiler is unable to check whether the use of the specified register conflicts with any other predefined register. It is recommended that you do not reserve too many registers in this way.

In order to change the assembly name of a function, you need a prototype declaration, because the compiler will not accept the `asm()` keyword in a function definition. For example:

```
extern long calc(void) asm ("CALCULATE");
```

Calling the function `calc()` in C code will generate assembler instructions which call the function called `CALCULATE`.

4.11.3.2 Accessing Registers From Assembly Code

In assembly code, SFR definitions are not automatically accessible. The header file `<xc.h>` can be included to gain access to these register definitions.

The symbols for registers in this header file are the same as those used in the C domain; however, you should use the appropriate I/O macros to ensure the correct address is encoded into instructions which accesses memory in the I/O space, for example, the following writes to the TCNT0 register:

```
out _SFR_IO_ADDR(TCNT0), r19
```

Bits within registers have macros associated with them and can be used directly with instructions that expect a bit number (0 thru 7), or with the `_BV()` macro if you need a bit mask based on that bit's position in the SFR, for example:

```
sbic _SFR_IO_ADDR(PORTD), PD4
ldi r16, _BV(TOIE0)
```

4.12 Optimizations

The MPLAB XC8 compiler can perform a variety of optimizations. Optimizations can be controlled using the `-O` option (described in 3.6.6 Options for Controlling Optimization). In Free mode, some of these optimizations are disabled. Even if they are enabled, optimizations might only be applied if very specific conditions are met. As a result, you might see that some lines of code are optimized, but other similar lines are not. When debugging code, you may wish to reduce the optimization level to ensure expected program flow.

4.13 Preprocessing

All C source files are preprocessed before compilation. The preprocessed file is deleted after compilation, but you can examine this file by using the `-E` option (see 3.6.2.2 E: Preprocess Only).

Assembler source files are preprocessed if the file uses a `.S` extension.

4.13.1 Preprocessor Directives

MPLAB XC8 accepts several specialized preprocessor directives, in addition to the standard directives. All of these are tabulated below.

Table 4-13. Preprocessor Directives

Directive	Meaning	Example
`#`	Preprocessor null directive, do nothing.	`#`
`#assert`	Generate error if condition false.	`#assert SIZE > 10`

`#define`	Define preprocessor macro.	`#define SIZE (5)`
		`#define FLAG`
		`#define add(a,b) ((a)+(b))`

`#elif`	Short for `#else #if`.	see `#ifdef`
`#else`	Conditionally include source lines.	see `#if`
`#endif`	Terminate conditional source inclusion.	see `#if`
`#error`	Generate an error message.	`#error Size too big`

`#if`	Include source lines if constant expression true.	`#if SIZE < 10`
		`    c = process(10)`
		`#else`
		`    skip();`
		`#endif`

...........continued

Directive	Meaning	Example

#ifdef	Include source lines if preprocessor symbol defined.	#ifdef FLAG
		    do_loop();
		#elif SIZE == 5
		    skip_loop();
		#endif

#ifndef	Include source lines if preprocessor symbol not defined.	#ifndef FLAG
		    jump();
		#endif

#include	Include text file into source.	#include <stdio.h> #include “project.h”
#line	Specify line number and filename for listing	#line 3 final
#*nn* filename	(where *nn* is a number, and *filename* is the name of the source file) the following content originated from the specified file and line number.	#20 init.c
#pragma	Compiler specific options.	See the Pragma Directives section in this guide.
#undef	Undefines preprocessor symbol.	#undef FLAG
#warning	Generate a warning message.	#warning Length not set

Macro expansion using arguments can use the # character to convert an argument to a string and the ## sequence to concatenate arguments. If two expressions are being concatenated, consider using two macros in case either expression requires substitution itself; for example

```
#define __paste1(a,b)   a##b
#define __paste(a,b)    __paste1(a,b)
```

lets you use the paste macro to concatenate two expressions that themselves can require further expansion. Remember, that once a macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

4.13.1.1 Preprocessor Arithmetic

Preprocessor macro replacement expressions are textual and do not utilize types. Unless they are part of the controlling expression to the inclusion directives (discussed below), macros are not evaluated by the preprocessor. Once macros have been textually expanded and preprocessing is complete, the expansion forms a C expression which is evaluated by the code generator along with other C code. Tokens within the expanded C expression inherit a type, with values then subject to integral promotion and type conversion in the usual way.

If a macro is part of the controlling expression to a conditional inclusion directive (#if or #elif), the macro must be evaluated by the preprocessor. The result of this evaluation is often different to the C-domain result for the same sequence. The preprocessor assigns sizes to literal values in the controlling expression that are equal to the largest integer size accepted by the compiler, as specified by the size of intmax_t defined in <stdint.h>.

For the MPLAB XC8 C Compiler, this size is 64 bits.

4.13.2 Predefined Macros

The compiler drivers define certain symbols to the preprocessor, allowing conditional compilation based on chip type, etc. The symbols tabulated below show the more common symbols defined by the drivers. Each symbol, if defined, is equated to 1. (unless otherwise stated).

Table 4-14. Predefined Macros

Symbol	Description
`__AVR_Device__`	Set when the `-mcpu` option specifies a device rather than an architecture. It indicates the device, for example when compiling for an atmega8, the macro `__AVR_ATmega8__` will be set.
`__AVR_DEVICE_NAME__`	Set when the `-mcpu` option specifies a device rather than an architecture. It indicates the device, for example when compiling for an atmega8 the macro is defined to atmega8.
`__AVR_ARCH__`	Indicates the device architecture. Possible values are: 2, 25, 3, 31, 35, 4, 5, 51, 6 for the avr2, avr25, avr3, avr31, avr35, avr4, avr5, avr51, avr6, architectures respectively and 100, 102, 103, 104, 105, 106, 107 for the avrtiny, avrxmega2, avrxmega3, avrxmega4, avrxmega5, avrxmega6, avrxmega7, architectures respectively.
`__AVR_ASM_ONLY__`	Indicates that the selected device can only be programmed in assembly.
`__AVR_CONST_DATA_IN_PROGMEM__`	Indicates that `const`-qualified objects will be placed in program memory.
`__AVR_ERRATA_SKIP__` `__AVR_ERRATA_SKIP_JMP_CALL__`	Indicates the selected device (AT90S8515, ATmega103) must not skip (SBRS, SBRC, SBIS, SBIC, and CPSE instructions) 32-bit instructions because of a hardware erratum. The second macro is only defined if `__AVR_HAVE_JMP_CALL__` is also set.
`__AVR_HAVE_EIJMP_EICALL__`	Indicates the selected device has more then 128 kB of program memory, a 3-byte wide program counter, and the `eijmp` and `eicall` instructions.
`__AVR_HAVE_ELPM__`	Indicates the selected device has the `elpm` instruction.
`__AVR_HAVE_ELPMX__`	Indicates the device has the `elpm Rn,Z` and `elpm Rn,Z+` instructions.
`__AVR_HAVE_JMP_CALL__`	Indicates the selected device has the `jmp` and `call` instructions and has more than 8kB of program memory.
`__AVR_HAVE_LPMX__`	Indicates the selected device has the `lpm Rn,Z` and `lpm Rn,Z+` instructions.
`__AVR_HAVE_MOVW__`	Indicates the selected device has the `movw` instruction, to perform 16-bit register-register moves.
`__AVR_HAVE_MUL__` `__AVR_HAVE_MUL__`	Indicates the selected device has a hardware multiplier.

`__AVR_HAVE_RAMPD__`
`__AVR_HAVE_RAMPX__`
`__AVR_HAVE_RAMPY__`
`__AVR_HAVE_RAMPZ__`
	Indicates the device has the RAMPD, RAMPX, RAMPY, or RAMPZ special function register, respectively.

`__AVR_HAVE_SPH__` `__AVR_SP8__`	Indicates the device has a 16- or 8-bit stack pointer, respectively. The definition of these macros is affected by the selected device, and for avr2 and avr25 architectures.
`__AVR_HAVE_8BIT_SP__` `__AVR_HAVE_16BIT_SP__`	Indicates the whether 8- or 16-bits of the stack pointer is used, respectively, by the compiler. The `-mtiny-stack` option will affect which macros are defined

...........continued

Symbol	Description
`__AVR_ISA_RMW__`	Indicates the selected device has Read-Modify-Write instructions (`xch`, `lac`, `las` and `lat`).
`__AVR_MEGA__`	Indicates the selected devices `jmp` and `call` instructions.
`__AVR_PM_BASE_ADDRESS__=`*`addr`*	Indicates the address space is linear and program memory is mapped into data memory. The value assigned to this macro is the starting address of the mapped memory.
`__AVR_SFR_OFFSET__=`*`offset`*	Indicates the offset to subtract from the data memory address for those instructions (e.g. `in`, `out`, and `sbi`) that can access SFRs directly.
`__AVR_SHORT_CALLS__`	Indicates the use of the `-mshort-calls` option, which affects the call instruction used and which can be set automatically.
`__AVR_TINY__`	Indicates that the selected device or architecture belongs to the TINY family.
`__AVR_TINY_PM_BASE_ADDRESS__=`*`addr`*	Deprecated; use `__AVR_PM_BASE_ADDRESS__`. Indicates the TINY device address space is linear and program memory is mapped into data memory.
`__AVR_XMEGA__`	Indicates that the selected device or architecture belongs to the XMEGA family.
`__AVR_2_BYTE_PC__`	Indicates the selected device has up to 128 kB of program memory and the program counter is 2 bytes wide.
`__AVR_3_BYTE_PC__`	Indicates the selected device has at least 128 kB of program memory and the program counter is 3 bytes wide.
`__BUILTIN_AVR_`*`name`*	Indicates the names built-in feature is available for the selected device
`__DEBUG`	When performing a debug build and you are using the MPLAB X IDE
`__FLASH`*`n`*	Defines `__FLASH`, `__FLASH1`, `__FLASH2` etc, based on the number of flash segments on the selected device.
`__MEMX`	Indicates the `__memx` specifier is available for the selected device.
`__NO_INTERRUPTS__`	Indicates the use of the `-mno-interrupts` option, which affects how the stack pointer is changed.
`__DATE__`	Indicate the current date, e.g., May 21 2004
`__FILE__`	Indicate this source file being preprocessed
`__TIME__`	Indicate the current time, e.g., 08:06:31
`__XC`	Indicates MPLAB XC compiler for Microchip is in use
`__XC8`	Indicates MPLAB XC compiler for Microchip 8-bit devices is in use
`__XC8_VERSION`	Indicates the compiler's version number multiplied by 1000, e.g., v1.00 will be represented by 1000

4.13.3 Pragma Directives

There is only one AVR-specific pragma that is accepted by MPLAB XC8, that being the `config` pragma, which is discussed in 4.2.4 Configuration Bit Access.

4.14 Linking Programs

The compiler will automatically invoke the linker unless the compiler has been requested to stop earlier in the compilation sequence.

The linker will run with options that are obtained from the command-line driver and use linker scripts, which specify memory areas and where sections are to be placed.

The linker options passed to the linker can be adjusted by the user, but this is only required in special circumstances (see 3.6.10.6 WL: Linker Option for more information).

The linker creates a map file which details the memory assigned to sections and some objects within the code. The map file is the best place to look for memory information.

4.14.1 Compiler-Generated Sections

The code generator places code and data into sections with standard names, which are subsequently positioned by the default linker scripts. A section can be created in assembly code by using the `.section` assembler directive. If you are unsure which section holds an object or code in your project, produce and check the relevant assembly list file.

4.14.1.1 Program Space Sections

The contents of common sections in program memory are described below.

`.text` These sections contain all executable code that does not require a special link location.

`.initn` These sections are used to define the runtime startup code, executed from the moment of reset right through to the invocation of `main()`. The code in these sections are executed in order from `init0` to `init9`.

`.finin` These sections are used to define the exit code, executed after `main()` terminates, either by returning or by calling to `exit()`. The code in the `.fini`*`n`* sections are executed in descending order from `.fini9` to `.fini0`.

4.14.1.2 Data Space Sections

The contents of common sections in data memory are described below.

`.bss` This section contains any objects with static storage duration that have not been initialized.

`.data` This section contains the RAM image of any objects with static storage duration that have been initialized with values.

`.rodata` These sections hold read-only data.

4.15 Changing and Linking the Allocated Section

The location of the default sections in which functions and objects are placed can be changed via driver options. The default sections the compiler uses to hold objects and code are listed in 4.14.1 Compiler-Generated Sections.

The `__section()` specifier allows you to have a object or function redirected into a user-define section. This allows you to relocate individual objects or functions.

Objects that use the `__section()` specifier will be cleared or initialized in the usual way by the runtime startup code.

The following are examples of a object and function allocated to a non-default section.

```
int __section("myBss") foobar;
int __section("myText") helper(int mode) { /* ... */ }
```

You can link these sections by using the `-Wl,--section-start=`*`section`*`=`*`addr`* option when building (linking) your program, provided that the linker script has already defined an output section with the same name. Note that you need to use an offset of 0x800000 for any address that is in the data space. For example, suppose you wish to place the new `myBss` section, created above, at SRAM address 0x300:

```
-Wl,--section-start=myBss=0x800300
```

For standard sections, like the `.text`, `.data` and `.bss` sections, they can be positioned using the `-Wl,-Tsection,addr` option when building (linking) your program. Thus, if you want the `.data` section to start at 0x1100, you can use the following option:

```
-Wl,-Tdata=0x801100
```

4.16 Linker Scripts

Linker scripts are used to instruct the linker how to position sections in memory. There are five different variants of these scripts, tabulated below. These are selected based on the options passed to the linker.

Table 4-15. Linker script variants

Script Extension	Controlling linker option	Linker operation
`.x`	default	
`.xr`	`-r`	perform no relocation
`.xu`	`-Ur`	resolve references to constructors
`.xn`	`-n`	set text to be read-only
`.xbn`	`-N`	Set the text and data sections to be readable and writable.

4.17 Replacing Library Modules

For library functions that are weak (see 4.7.2.4 Weak Attribute), you can have your own version of a routine replace a library routine with the same name without having to using the librarian, `avr-ar`. Simply include the definition of that routine as part of your project.

5. Utilities

This chapter discusses some of the utility applications that are bundled with the compiler.

The applications discussed in this chapter are those more commonly used, but you do not typically need to execute them directly. Some of their features are invoked indirectly by the command line driver that is based on the command-line arguments or MPLAB X IDE project property selections.

5.1 Archiver/Librarian

The archiver/librarian program has the function of combining several intermediate files into a single file, known as a library archive file. Library archives are easier to manage and might consume less disk space than the individual files contained in them.

The archiver can build all library archive types needed by the compiler and can detect the format of existing archives.

5.1.1 Using the Archiver/Librarian

The archiver program is called `avr-ar` and is used to create and edit library archive files. It has the following basic command format:

```
avr-ar [options] file.a [file1.p1 file2.o...]
```

where *`file`*`.a` represents the library archive being created or edited.

The files following the archive file, if required, are the object (`.o`) modules that are required by the command specified.

The *`options`* is zero or more options, tabulated below, that control the program.

Table 5-1. Archiver Command-line Options

Option	Effect
`-d`	Delete module
`-m`	Re-order modules
`-p`	List modules
`-r`	Replace modules
`-t`	List modules with symbols
`-x`	Extract modules
`--target`	Specify target device

When replacing or extracting modules, the names of the modules to be replaced or extracted must be specified. If no names are supplied, all the modules in the archive will be replaced or extracted respectively.

Creating an archive file or adding a file to an existing archive is performed by requesting the archiver to replace the module in the archive. Since the module is not present, it will be appended to the archive. Object and p-code modules can be added to the same archive. The archiver creates library archives with the modules in the order in which they were given on the command line. When updating an archive, the order of the modules is preserved. Any modules added to an archive will be appended to the end.

The ordering of the modules in an archive is significant to the linker. If an archive contains a module that references a symbol defined in another module in the same archive, the module defining the symbol should come after the module referencing the symbol.

When using the `-d` option, the specified modules will be deleted from the archive. In this instance, it is an error not to supply any module names.

The `-p` option will list the modules within the archive file.

The `-m` option takes a list of module names and re-orders the matching modules in the archive file so that they have the same order as the one listed on the command line. Modules that are not listed are left in their existing order, and will appear after the re-ordered modules.

The `avr-ar` archiver will not work for object files built with only LTO data (i.e built with the `-fno-fat-lto-objects` option). For such object files, use the `avr-gcc-ar` archiver instead

5.1.1.1 Archiver Examples

Here are some examples using the archiver. The following command:

```
xc8-ar -r myAvrLib.a ctime.o init.o
```

creates a library archive called `myAvrLib.a` that contains the modules `ctime.o` and `init.o`. The following command deletes the object module `a.o` from the library archive `lcd.a`:

```
xc8-ar -d lcd.a a.o
```

5.2 Hexmate

The Hexmate utility is a program designed to manipulate Intel HEX files. Hexmate is a post-link stage utility that is automatically invoked by the compiler driver and that provides the facility to:

- Calculate and store variable-length hash values.
- Fill unused memory locations with known data sequences.
- Merge multiple Intel HEX files into one output file.
- Convert INHX32 files to other INHX formats (e.g., INHX8M).
- Detect specific or partial opcode sequences within a HEX file.
- Find/replace specific or partial opcode sequences.
- Provide a map of addresses used in a HEX file.
- Change or fix the length of data records in a HEX file.
- Validate checksums within Intel HEX files.

Typical applications for Hexmate might include:

- Merging a bootloader or debug module into a main application at build time.
- Calculating a checksum or CRC value over a range of program memory and storing its value in program memory or EEPROM.
- Filling unused memory locations with an instruction to send the program counter to a known location if it gets lost.
- Storage of a serial number at a fixed address.
- Storage of a string (e.g., time stamp) at a fixed address.
- Store initial values at a particular memory address (e.g., initialize EEPROM).
- Detecting usage of a buggy/restricted instruction.
- Adjusting HEX file to meet requirements of particular bootloaders.

5.2.1 Hexmate Command Line Options

Hexmate is automatically called by the command line driver, `xc8-cc` to merge any HEX files specified on the command line with the output generated by the source files. Some other hexmate functions can be requested from the driver without running Hexmate explicitly, but full control you may run hexmate on the command line and use the options detailed here.

If Hexmate is to be run directly, its usage is:

```
hexmate [specs,]file1.hex [... [specs,]fileN.hex] [options]
```

where *`file1.hex`* through to *`fileN.hex`* form a list of input Intel HEX files to merge using Hexmate.

If only one HEX file is specified, no merging takes place, but other functionality is specified by additional options. Tabulated below are the command line options that Hexmate accepts.

Table 5-2. Hexmate Command-line Options

Option	Effect
`--edf`	Specify the message description file.
`--emax`	Set the maximum number of permitted errors before terminating.
`--msgdisable`	Disable messages with the numbers specified.
`--sla`	Set the start linear address for type 5 records.
`--ver`	Display version and build information then quit.
`-addressing`	Set address fields in all hexmate options to use word addressing or other.
`-break`	Break continuous data so that a new record begins at a set address.
`-ck`	Calculate and store a value.
`-fill`	Program unused locations with a known value.
`-find`	Search and notify if a particular code sequence is detected.
`-find...,delete`	Remove the code sequence if it is detected (use with caution).
`-find...,replace`	Replace the code sequence with a new code sequence.
`-format`	Specify maximum data record length or select INHX variant.
`-help`	Show all options or display help message for specific option.
`-logfile`	Save hexmate analysis of output and various results to a file.
`-mask`	Logically AND a memory range with a bitmask.
`-o`*`file`*	Specify the name of the output file.
`-serial`	Store a serial number or code sequence at a fixed address.
`-size`	Report the number of bytes of data contained in the resultant HEX image.
`-string`	Store an ASCII string at a fixed address.
`-strpack`	Store an ASCII string at a fixed address using string packing.
`-w`	Adjust warning sensitivity.
`+`	Prefix to any option to overwrite other data in its address range, if necessary.

If you are using the driver, `xc8-cc`, to compile your project (or the IDE), a log file is produced by default. It will have the project's name and the extension `.hxl`.

The input parameters to Hexmate are now discussed in detail. The format or assumed radix of values associated with options are described with each option. Note that any address fields specified in these options are to be entered as byte addresses, unless specified otherwise in the `-addressing` option.

5.2.1.1 Specifications And Filename

Hexmate can process Intel HEX files that use either INHX32 or INHX8M format. Additional specifications can be applied to each HEX file to place restrictions or conditions on how this file should be processed.

If any specifications are used, they must precede the filename. The list of specifications will then be separated from the filename by a comma.

A range restriction can be applied with the specification `rStart-End`, where `Start` and `End` are both assumed to be hexadecimal values. A range restriction will cause only the address data falling within this range to be used. For example:

```
r100-1FF,myfile.hex
```

will use `myfile.hex` as input, but only process data which is addressed within the range 100h-1FFh (inclusive) from that file.

An address shift can be applied with the specification `sOffset`. If an address shift is used, data read from this HEX file will be shifted (by the offset specified) to a new address when generating the output. The offset can be either positive or negative. For example:

```
r100-1FFs2000,myfile.HEX
```

will shift the block of data from 100h-1FFh to the new address range 2100h-21FFh.

Be careful when shifting sections of executable code. Program code should only be shifted if it is position independent.

5.2.1.2 Override Prefix

When the `+` operator precedes an argument or input file, the data obtained from that source will be forced into the output file and will overwrite another other data existing at that address range. For example:

```
input.HEX +-string@1000=”My string”
```

will have the data specified by the `-STRING` option placed at address 1000; however:

```
+input.HEX -string@1000=”My string”
```

will copy the data contained in the hex file at address 1000 into the final output.

Ordinarily, Hexmate will issue an error if two sources try to store differing data at the same location. Using the `+` operator informs Hexmate that if more than one data source tries to store data to the same address, the one specified with a `+` prefix will take priority.

5.2.1.3 Edf

The `--edf=file` specifies the message description file to use when displaying warning or error messages. The argument should be the full path to the message file. Hexmate contains an internal copy of this file, so the use of this option is not necessary, but you may wish to specify a file with updated contents.

The message files are located in the `pic/dat` directory in the installation (e.g., the English language file is called `en_msgs.txt`).

5.2.1.4 Emax

The `--emax=num` option sets the maximum number of errors Hexmate will display before execution is terminated, e.g., `--emax=25`. By default, up to 20 error messages will be displayed.

5.2.1.5 Msgdisable

The `--msgdisable=number` option allows error, warning or advisory messages to be disabled during execution of Hexmate.

The option is passed a comma-separated list of message numbers that are to be disabled. Any error message numbers in this list are ignored unless they are followed by an `:off` argument. If the message list is specified as 0, then all warnings are disabled.

5.2.1.6 Sla

The `--sla=address` option allows you to specify the linear start address for type 5 records in the HEX output file, e.g., `--sla=0x10000`.

5.2.1.7 Ver

The `--ver` option will ask Hexmate to print version and build information and then quit.

5.2.1.8 Addressing

This option allows the addressing units of any addresses in Hexmate's command line options to be changed.

By default, all address arguments in Hexmate options expect that values will be entered as byte addresses, as specified in Intel HEX files. In some device architectures, the native addressing format can be something other than byte addressing. In these cases, this option can be used to allow device-orientated addresses to be used with Hexmate's command-line options.

This option takes one parameter that configures the number of bytes contained per address location. For Baseline, Mid-range, and 24-bit PIC devices, an addressing unit of 2 can be used, if desired. For all other devices, you would typically use the default addressing unit of 1 byte.

5.2.1.9 Break

The `-break` option takes a comma-separated list of addresses. If any of these addresses are encountered in the HEX file, the current data record will conclude and a new data record will recommence from the nominated address. This can be useful to use new data records to force a distinction between functionally different areas of program space. Some HEX file readers depend on this.

5.2.1.10 Ck

The `-ck` option is for calculating a hash value. The usage of this option is:

```
-ck=start-end@dest [+offset][wWidth][tCode][gAlgorithm][pPolynomial][rwidth]
```

where:

- *`start`* and *`end`* specify the address range over which the hash will be calculated. If these addresses are not a multiple of the algorithm width, the value zero will be padded into the relevant input word locations that are missing.
- *`dest`* is the address where the hash result will be stored. This value cannot be within the range of calculation.
- *`offset`* is an optional initial value to be used in the calculations.
- *`Width`* is optional and specifies the width of the result. Results can be calculated for byte-widths of 1 to 4 bytes for most algorithms, but it represents the bit width for SHA algorithms. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order. This width argument is not used if you have selected any Fletcher algorithm.
- *`Code`* is a hexadecimal code that will trail each byte in the result. This can allow each byte of the hash result to be embedded within an instruction, for example `code=34` will embed the result in a `retlw` instruction on Mid-range devices.
- *`Algorithm`* is an integer to select which Hexmate hash algorithm to use to calculate the result. A list of selectable algorithms is provided in Table 5-3. If unspecified, the default algorithm used is 8-bit checksum addition (algorithm 1).
- *`Polynomial`* is a hexadecimal value which is the polynomial to be used if you have selected a CRC algorithm.
- *`r`* is a decimal word width. If this is non-zero, then bytes within each word are read in reverse order when calculating a hash value. At present, the width must be 0 or 2. A zero width disables the reverse-byte feature, as if the `r` suboption was not present. This suboption should be used when using Hexmate to match a CRC produced by a PIC hardware CRC module that use the Scanner module to stream data to it.

All numerical arguments are assumed to be hexadecimal values, except for the algorithm selector and result width, which are assumed to be decimal values.

A typical example of the use of the checksum option is:

```
-ck=0-1FFF@2FFE+2100w2g2
```

This will calculate a checksum over the range 0 to 0x1FFF and program the checksum result at address 0x2FFE. The checksum value will be offset by 0x2100. The result will be two bytes wide.

Table 5-3. Hexmate Hash Algorithm Selection

Selector	Algorithm Description
-5	Reflected cyclic redundancy check (CRC).
-4	Subtraction of 32 bit values from initial value.
-3	Subtraction of 24 bit values from initial value.
-2	Subtraction of 16 bit values from initial value.
-1	Subtraction of 8 bit values from initial value.
1	Addition of 8 bit values from initial value.
2	Addition of 16 bit values from initial value.
3	Addition of 24 bit values from initial value.
4	Addition of 32 bit values from initial value.
5	Cyclic redundancy check (CRC).
7	Fletcher's checksum (8 bit calculation, 2-byte result width).
8	Fletcher's checksum (16 bit calculation, 4-byte result width).
10	SHA-2 (currently only SHA256 is supported)

See 5.2.2 Hash Functions for more details about the algorithms that are used to calculate checksums.

5.2.1.11 Fill

The `-fill` option is used for filling unused memory locations with a known value. The usage of this option is:

```
-fill=[const_width:]fill_expr@address[:end_address]
```

where:

- `const_width` has the form `wn` and signifies the width (*n* bytes) of each constant in `fill_expr`. If `const_width` is not specified, the default value is two bytes. For example, `-fill=w1:1` with fill every unused byte with the value 0x01.
- `fill_expr` can use the syntax (where const and increment are n-byte constants):
  - *`const`* fill memory with a repeating constant; i.e., `-fill=0xBEEF` becomes 0xBEEF, 0xBEEF, 0xBEEF, 0xBEEF.
  - *`const+=increment`* fill memory with an incrementing constant; i.e., `-fill=0xBEEF+=1` becomes 0xBEEF, 0xBEF0, 0xBEF1, 0xBEF2.
  - *`const-=increment`* fill memory with a decrementing constant; i.e., `-fill=0xBEEF-=0x10` becomes 0xBEEF, 0xBEDF, 0xBECF, 0xBEBF.
  - *`const,const,...,const`* fill memory with a list of repeating constants; i.e., `-fill=0xDEAD,0xBEEF` becomes 0xDEAD,0xBEEF,0xDEAD,0xBEEF.
- The options following `fill_expr` result in the following behavior:
  - `@`*`address`* fill a specific address with `fill_expr`; i.e., `-fill=0xBEEF@0x1000` puts 0xBEEF at address 1000h. If the fill value is wider than the addressing value specified with `-addressing`, then only part of the fill value is placed in the output. For example, if the addressing is set to 1, the option above will place 0xEF at address 0x1000 and a warning will be issued.
  - `@`*`address`*`:`*`end_address`* fill a range of memory with `fill_expr`; i.e., `-fill=0xBEEF@0:0xFF` puts 0xBEEF in unused addresses between 0 and 255. If the address range (multiplied by the `-ADDRESSING` value) is not a multiple of the fill value width, the final location will only use part of the fill value, and a warning will be issued.

The fill values are word-aligned so they start on an address that is a multiple of the fill width. Should the fill value be an instruction opcode, this alignment ensures that the instruction can be executed correctly. Similarly, if the total length of the fill sequence is larger than 1 (and even if the specified width is 1), the fill sequence is aligned to that total length. For example the following fill option, which specifies 2 bytes of fill sequence and a starting address that is not a multiple of 2:

```
-fill=w1:0x11,0x22@0x11001:0x1100c
```

will result in the following hex record, where the starting address was filled with the second byte of the fill sequence due to this alignment.

```
:0C100100221122112211221122112211B1
```

All constants can be expressed in (unsigned) binary, octal, decimal or hexadecimal, as per normal C syntax, for example, 1234 is a decimal value, 0xFF00 is hexadecimal and FF00 is illegal.

5.2.1.12 Find

The `-find=`*`opcode`* option is used to detect and log occurrences of an opcode or partial code sequence. The usage of this option is:

```
-find=Findcode [mMask]@Start-End [/Align][w][t"Title"]
```

where:

- *`Findcode`* is the hexadecimal code sequence to search for. For example, to find a `clrf` instruction with the opcode 0x01F1, use 01F1 as the sequence. In the HEX file, this will appear as the byte sequence F1 01, that is F1 at hex address 0 and 01 at hex address 1.
- *`Mask`* is optional. It specifies a bit mask applied over the *`Findcode`* value to allow a less restrictive search. It is entered in little endian byte order.
- `Start` and *`End`* limit the address range to search.
- *`Align`* is optional. It specifies that a code sequence can only match if it begins on an address that is a multiple of this value.
- `w`, if present, will cause Hexmate to issue a warning whenever the code sequence is detected.
- *`Title`* is optional. It allows a title to be given to this code sequence. Defining a title will make log-reports and messages more descriptive and more readable. A title will not affect the actual search results.

All numerical arguments are assumed to be hexadecimal values.

Here are some examples.

The option `-find=1234@0-7FFF/2w` will detect the code sequence 1234h when aligned on a 2 (two) byte address boundary, between 0h and 7FFFh. w indicates that a warning will be issued each time this sequence is found.

In this next example, `-find=1234M0F00@0-7FFF/2wt"ADDXY"`, the option is the same as in last example but the code sequence being matched is masked with 000Fh, so Hexmate will search for any of the opcodes `123`*`x`*h, where *`x`* is any digit. If a byte-mask is used, is must be of equal byte-width to the opcode it is applied to. Any messaging or reports generated by Hexmate will refer to this opcode by the name, `ADDXY`, as this was the title defined for this search.

If Hexmate is generating a log file, it will contain the results of all searches. `-find` accepts whole bytes of HEX data from 1 to 8 bytes in length. Optionally, `-find` can be used in conjunction with `replace` or `delete` (as described below).

5.2.1.13 Find And Delete

If the `delete` form of the `-find` option is used, any matching sequences will be removed. This function should be used with extreme caution and is not normally recommended for removal of executable code.

5.2.1.14 Find and Replace

If the `replace` form of the `-find` option is used, any matching sequences will be replaced, or partially replaced, with new codes. The usage for this sub-option is:

```
-find...,replace=Code [mMask]
```

where:

- *Code* is a hexadecimal code sequence to replace the sequences that match the `-find` criteria.
- *Mask* is an optional bit mask to specify which bits within *Code* will replace the code sequence that has been matched. This can be useful if, for example, it is only necessary to modify 4 bits within a 16-bit instruction. The remaining 12 bits can masked and left unchanged.

5.2.1.15 Format

The `-format` option can be used to specify a particular variant of INHX format or adjust maximum record length. The usage of this option is:

```
-format=Type [,Length]
```

where:

- *Type* specifies a particular INHX format to generate.
- *Length* is optional and sets the maximum number of bytes per data record. A valid length is between 1 and 16 decimal, with 16 being the default.

Consider the case of a bootloader trying to download an INHX32 file, which fails because it cannot process the extended address records that are part of the INHX32 standard. This bootloader can only program data addressed within the range 0 to 64k and any data in the HEX file outside of this range can be safely disregarded. In this case, by generating the HEX file in INHX8M format the operation might succeed. The Hexmate option to do this would be `-FORMAT=INHX8M`.

Now consider if the same bootloader also required every data record to contain exactly 8 bytes of data. This is possible by combining the `-format` with `-fill` options. Appropriate use of `-fill` can ensure that there are no gaps in the data for the address range being programmed. This will satisfy the minimum data length requirement. To set the maximum length of data records to 8 bytes, just modify the previous option to become `-format=INHX8M,8`.

The possible types that are supported by this option are listed in Table 5-4. Note that `INHX032` is not an actual INHX format. Selection of this type generates an INHX32 file, but will also initialize the upper address information to zero. This is a requirement of some device programmers.

Table 5-4. Inhx Types

Type	Description
`INHX8M`	Cannot program addresses beyond 64K.
`INHX32`	Can program addresses beyond 64K with extended linear address records.
`INHX032`	INHX32 with initialization of upper address to zero.

5.2.1.16 Help

Using `-help` will list all Hexmate options. Entering another Hexmate option as a parameter of `-help` will show a detailed help message for the given option. For example:

```
-help=string
```

will show additional help for the `-string` Hexmate option.

5.2.1.17 Logfile

The `-logfile` option saves HEX file statistics to the named file. For example:

```
-logfile=output.hxl
```

will analyze the HEX file that Hexmate is generating and save a report to a file named `output.hxl`.

5.2.1.18 Mask

Use the `-mask=`*`spec`* option to logically AND a memory range with a particular bitmask. This is used to ensure that the unimplemented bits in program words (if any) are left blank. The usage of this option is as follows:

```
-mask=hexcode@start-end
```

where *`hexcode`* is a value that will be ANDed with data within the *`start`* to *`end`* address range. All values are assumed to be hexadecimal. Multibyte mask values can be entered in little endian byte order.

5.2.1.19 O: Specify Output File

When using the `-o`*`file`* option, the generated Intel HEX output will be created in this file. For example:

```
-oprogram.hex
```

will save the resultant output to `program.hex`. The output file can take the same name as one of its input files; but, by doing so, it will replace the input file entirely.

5.2.1.20 Serial

The `-serial=`*`specs`* option will store a particular HEX value sequence at a fixed address. The usage of this option is:

```
-serial=Code[+/-Increment]@Address[+/-Interval][rRepetitions]
```

where:

- *`Code`* is a hexadecimal sequence to store. The first byte specified is stored at the lowest address.
- *`Increment`* is optional and allows the value of Code to change by this value with each repetition (if requested).
- *`Address`* is the location to store this code, or the first repetition thereof.
- *`Interval`* is optional and specifies the address shift per repetition of this code.
- *`Repetitions`* is optional and specifies the number of times to repeat this code.

All numerical arguments are assumed to be hexadecimal values, except for the *`Repetitions`* argument, which is decimal value by default.

For example:

```
-serial=000001@EFFE
```

will store HEX code 00001h to address EFFEh.

Another example:

```
-serial=0000+2@1000+10r5
```

will store 5 codes, beginning with value 0000 at address 1000h. Subsequent codes will appear at address intervals of +10h and the code value will change in increments of +2h.

5.2.1.21 Size

Using the `-size` option will report the number of bytes of data within the resultant HEX image to standard output. The size will also be recorded in the log file if one has been requested.

5.2.1.22 String

The `-string` option will embed an ASCII string at a fixed address. The usage of this option is:

```
-string@Address[tCode]="Text"
```

where:

- *`Address`* is assumed to be a hexadecimal value representing the address at which the string will be stored.
- *`Code`* is optional and allows a byte sequence to trail each byte in the string. This can allow the bytes of the string to be encoded within an instruction.

- *`Text`* is the string to convert to ASCII and embed.

For example:

```
-string@1000="My favorite string"
```

will store the ASCII data for the string, My favorite string (including the null character terminator), at address 1000h.

And again:

```
-string@1000t34="My favorite string"
```

will store the same string, trailing every byte in the string with the HEX code 34h.

5.2.1.23 Strpack

The `-strpack=spec` option performs the same function as `-string`, but with two important differences. First, only the lower seven bits from each character are stored. Pairs of 7-bit characters are then concatenated and stored as a 14-bit word rather than in separate bytes. This is known as string packing. This is usually only useful for devices where program space is addressed as 14-bit words (AVR 8 devices). The second difference is that `-string`'s `t` specifier is not applicable with the `-strpack` option.

5.2.2 Hash Functions

A hash value is a small fixed-size value that is calculated from, and used to represent, all the values in an arbitrary-sized block of data. If that data block is copied, a hash recalculated from the new block can be compared to the original hash. Agreement between the two hashes provides a high level of certainty that the copy is valid. There are many hash algorithms. More complex algorithms provide a more robust verification, but could use too many resources when used in an embedded environment.

Hexmate can be used to calculate the hash of a program image that is contained in a HEX file built by the MPLAB XC8 C Compiler. This hash can be embedded into that HEX file and burned into the target device along with the program image. At runtime, the target device might be able to run a similar hash algorithm over the program image, now stored in its memory. If the stored and calculated hashes are the same, the embedded program can assume that it has a valid program image to execute.

Hexmate implements several checksum and cyclic redundancy check algorithms to calculate the hash. If you are using the `xc8-cc` driver to perform project builds, the driver's `-mchecksum` option will instruct the driver to invoke Hexmate and pass it the appropriate Hexmate options. That same option is available in the MPLAB X IDE. If you are driving Hexmate explicitly, the option to select the algorithm is described in 5.2.1.10 Ck. In the discussion of the algorithms below, it is assumed you are using the compiler driver to request a checksum or CRC.

Some consideration is required when program images contain unused memory locations. The driver's `-mchecksum` option automatically requests that Hexmate fill unused memory locations to match unprogrammed device memory. You might need to mimic this action if invoking Hexmate explicitly.

Although Hexmate will work with any device, not all devices can read the entire width of their program memory. Mid-range PIC devices also use a 14-bit wide program memory, thus you cannot store a hash larger than a byte directly. For these devices, you would typically use the `code=nn` argument to the `-mcodeoffset` option, to have each byte of the hash value encapsulated in an instruction.

The following sections provide examples of the algorithms that can be used to calculate the hash at runtime, but note that these examples are not directly usable with all devices.

5.2.3 Addition Algorithms

Hexmate has several simple checksum algorithms that sum data values over a range in the program image. These algorithms correspond to the selector values 1, 2, 3 and 4 in the algorithm suboption and read the data in the program image as 1, 2, 3 or 4 byte quantities, respectively. This summation is added to an initial value (offset) that is supplied to the algorithm via the same option. The width to which the final checksum is truncated is also specified by this option and can be 1, 2, 3 or 4 bytes. Hexmate will automatically store the checksum in the HEX file at the address specified in the checksum option.

The function shown below can be customized to work with any combination of data size (`readType`) and checksum width (`resultType`).

```
typedef unsigned char readType; // size of data values read and summed
typedef unsigned int  resultType; // size of checksum result
// add to offset n additions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to sum
// n:    the number of sums to perform
// offset: the intial value to which the sum is added
resultType ck_add(const readType *data, unsigned n, resultType offset)
{
    resultType chksum;
    chksum = offset;
    while(n--) {
        chksum += *data;
        data++;
    }
    return chksum;
}
```

The `readType` and `resultType` type definitions should be adjusted to suit the data read/sum width and checksum result width, respectively. When using MPLAB XC8 and for a size of 1, use a `char` type; for a size of 4, use a `long` type, etc., or consider using the exact-width types provided by `<stdint.h>`. If you never use an offset, that parameter can be removed and `chksum` assigned 0 before the loop.

Here is how this function might be used when, for example, a 2-byte-wide checksum is to be calculated from the addition of 1-byte-wide values over the address range 0x100 to 0x7fd, starting with an offset of 0x20. The checksum is to be stored at 0x7fe and 0x7ff in little endian format. The following option is specified when building the project. In MPLAB X IDE, only enter the information to the right of the first = in the **Checksum** field in the **Additional options** Option category in the **XC8 Linker** category.

```
-mchecksum=100-7fd@7fe,offset=20,algorithm=1,width=-2
```

In your project, add the following code snippet which calls `ck_add()` and compare the runtime checksum with that stored by Hexmate at compile time.

```
extern const readType ck_range[0x6fe/sizeof(readType)] __at(0x100);
extern const resultType hexmate __at(0x7fe);
resultType result;
result = ck_add(ck_range, sizeof(ck_range)/sizeof(readType), 0x20);
if(result != hexmate)
    ck_failure();  // take appropriate action
```

This code uses the placeholder array, `ck_range`, to represent the memory over which the checksum is calculated and the variable `hexmate` is mapped over the locations where Hexmate will have stored its checksum result. Being `extern` and absolute, neither of these objects consume additional device memory. Adjust the addresses and sizes of these objects to match the option you pass to Hexmate.

Hexmate can calculate a checksum over any address range; however, the test function, `ck_add()`, assumes that the start and end address of the range being summed are a multiple of the `readType` width. This is a non-issue if the size of `readType` is 1. It is recommended that your checksum specification adheres to this assumption, otherwise you will need to modify the test code to perform partial reads of the starting and/or ending data values. This will significantly increase the code complexity.

5.2.4 Subtraction Algorithms

Hexmate has several checksum algorithms that subtract data values over a range in the program image. These algorithms correspond to the selector values -1, -2, -3, and -4 in the algorithm suboption and read the data in the program image as 1-, 2-, 3- or 4-byte quantities, respectively. In other respects, these algorithms are identical to the addition algorithms described in 5.2.3 Addition Algorithms.

The function shown below can be customized to work with any combination of data size (`readType`) and checksum width (`resultType`).

```
typedef unsigned char readType; // size of data values read and summed
typedef unsigned int  resultType; // size of checksum result
// add to offset n subtractions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to subtract
// n:    the number of subtractions to perform
// offset: the intial value to which the subtraction is added
resultType ck_sub(const readType *data, unsigned n, resultType offset)
{
    resultType chksum;
    chksum = offset;
    while(n--) {
        chksum -= *data;
        data++;
    }
    return chksum;
}
```

Here is how this function might be used when, for example, a 4-byte-wide checksum is to be calculated from the addition of 2-byte-wide values over the address range 0x0 to 0x7fd, starting with an offset of 0x0. The checksum is to be stored at 0x7fe and 0x7ff in little endian format. The following option is specified when building the project. In MPLAB X IDE, only enter the information to the right of the first = in the **Checksum** field in the **Additional options** Option category in the **XC8 Linker** category.

```
-mchecksum=0-7fd@7fe,offset=0,algorithm=-2,width=-4
```

In your project, add the following code snippet which calls `ck_sub()` and compare the runtime checksum with that stored by Hexmate at compile time.

```
extern const readType ck_range[0x7fe/sizeof(readType)] __at(0x0);
extern const resultType hexmate __at(0x7fe);
resultType result;
result = ck_sub(ck_range, sizeof(ck_range)/sizeof(readType), 0x0);
if(result != hexmate)
    ck_failure();  // take appropriate action
```

5.2.5 Fletcher Algorithms

Hexmate has several algorithms that implement Fletcher's checksum. These algorithms are more complex, providing a robustness approaching that of a cyclic redundancy check, but with less computational effort. There are two forms of this algorithm which correspond to the selector values 7 and 8 in the algorithm suboption and which implement a 1-byte calculation and 2-byte result, with a 2-byte calculation and 4-byte result, respectively. Hexmate will automatically store the checksum in the HEX file at the address specified in the checksum option.

The function shown below performs a 1-byte-wide addition and produces a 2-byte result.

```
unsigned int
fletcher8(const unsigned char * data, unsigned int n )
{
    unsigned int sum = 0xff, sumB = 0xff;
    unsigned char tlen;
    while (n) {
        tlen = n > 20 ? 20 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xff) + (sum >> 8);
        sumB = (sumB & 0xff) + (sumB >> 8);
    }
    sum = (sum & 0xff) + (sum >> 8);
    sumB = (sumB & 0xff) + (sumB >> 8);
    return sumB << 8 | sum;
}
```

This code can be called in a manner similar to that shown for the addition algorithms (see 5.2.3 Addition Algorithms).

The code for the 2-byte-addition Fletcher algorithm, producing a 4-byte result is shown below.

```
unsigned long
fletcher16(const unsigned int * data, unsigned n)
{
    unsigned long sum = 0xffff, sumB = 0xffff;
    unsigned tlen;
    while (n) {
        tlen = n > 359 ? 359 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xffff) + (sum >> 16);
        sumB = (sumB & 0xffff) + (sumB >> 16);
    }
    sum = (sum & 0xffff) + (sum >> 16);
    sumB = (sumB & 0xffff) + (sumB >> 16);
    return sumB << 16 | sum;
}
```

5.2.6 CRC Algorithms

Hexmate has several algorithms that implement the robust cyclic redundancy checks (CRC). There is a choice of two algorithms that correspond to the selector values 5 and -5 in the algorithm suboption to `-mchecksum` and that implement a CRC calculation and reflected CRC calculation, respectively. The reflected algorithm works on the least significant bit of the data first.

The polynomial to be used and the initial value can be specified in the option. Hexmate will automatically store the CRC result in the HEX file at the address specified in the checksum option.

Some devices implement a CRC module in hardware that can be used to calculate a CRC at runtime. These modules can stream data read from program memory using a Scanner module. To ensure that the order of the bytes processed by Hexmate and the CRC/Scanner module are identical, you must specify a reserve word width of 2 using the suboption `revword=2`. This will read each 2-byte word in the HEX file in order, but process the bytes within those words in reverse order.

The function shown below can be customized to work with any result width (`resultType`). It calculates a CRC hash value using the polynomial specified by the `POLYNOMIAL` macro.

```
typedef unsigned int resultType;
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(resultType))
#define MSb     ((resultType)1 << (WIDTH - 1))

resultType
crc(const unsigned char * data, unsigned n, resultType remainder) {
    unsigned pos;
    unsigned char bitp;
    for (pos = 0; pos != n; pos++) {
        remainder ^= ((resultType)data[pos] << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    return remainder;
}
```

The `resultType` type definition should be adjusted to suit the result width. When using MPLAB XC8 and for a size of 1, use a `char` type; for a size of 4, use a `long` type, etc., or consider using the exact-width types provided by `<stdint.h>`.

Here is how this function might be used when, for example, a 2-byte-wide CRC hash value is to be calculated values over the address range 0x0 to 0xFF, starting with an initial value of 0xFFFF. The result is to be stored at 0x100 and 0x101 in little endian format. The following option is specified when building the project. In MPLAB X IDE, only enter the information to the right of the first = in the **Checksum** field in the **Additional options** Option category in the **XC8 Linker** category.

```
-mchecksum=0-FF@100,offset=0xFFFF,algorithm=5,width=-2,polynomial=0x1021
```

In your project, add the following code snippet which calls `crc()` and compares the runtime hash result with that stored by Hexmate at compile time.

```
extern const unsigned char ck_range[0x100] __at(0x0);
extern const resultType hexmate __at(0x100);
resultType   result;

result = crc(ck_range, sizeof(ck_range), 0xFFFF);
if(result != hexmate){
    // something's not right, take appropriate action
    ck_failure();
}
// data verifies okay, continue with the program
```

The reflected CRC result can be calculated by reflecting the input data and final result, or by reflecting the polynomial. The functions shown below can be customized to work with any result width (`resultType`). The `crc_reflected_IO()` function calculates a reflected CRC hash value by reflecting the data stream bit positions. Alternatively, the `crc_reflected_poly()` function does not adjust the data stream but reflects instead the polynomial, which in both functions is specified by the `POLYNOMIAL` macro. Both functions use the `reflect()` function to perform bit reflection.

```
typedef unsigned int resultType;
typedef unsigned char readType;
typedef unsigned int reflectWidth;
// This is the polynomial used by the CRC-16 algorithm we are using.
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(resultType))
#define MSb     ((resultType)1 << (WIDTH - 1))
#define LSb     (1)
#define REFLECT_DATA(X)       ((readType) reflect((X), 8))
#define REFLECT_REMAINDER(X)  (reflect((X), WIDTH))

reflectWidth
reflect(reflectWidth data, unsigned char nBits)
{
    reflectWidth reflection = 0;
    reflectWidth reflectMask = (reflectWidth)1 << nBits - 1;
    unsigned char bitp;
    for (bitp = 0; bitp != nBits; bitp++) {
        if (data & 0x01) {
            reflection |= reflectMask;
        }
        data >>= 1;
        reflectMask >>= 1;
    }
    return reflection;
}

resultType
crc_reflected_IO(const unsigned char * data, unsigned n, resultType remainder) {
    unsigned pos;
    unsigned char reflected;
    unsigned char bitp;
    for (pos = 0; pos != n; pos++) {
        reflected = REFLECT_DATA(data[pos]);
        remainder ^= ((resultType)reflected << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    remainder = REFLECT_REMAINDER(remainder);
    return remainder;
```

```
}

resultType
crc_reflected_poly(const unsigned char * data, unsigned n, resultType remainder) {
    unsigned pos;
    unsigned char bitp;
    resultType rpoly;
    rpoly = reflect(POLYNOMIAL, WIDTH);
    for (pos = 0; pos != n; pos++) {
        remainder ^= data[pos];
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & LSb) {
                remainder = (remainder >> 1) ^ rpoly;
            } else {
                remainder >>= 1;
            }
        }
    }
    return remainder;
}
```

Here is how this function might be used when, for example, a 2-byte-wide reflected CRC result is to be calculated over the address range 0x0 to 0xFF, starting with an initial value of 0xFFFF. The result is to be stored at 0x100 and 0x101 in little endian format. The following option is specified when building the project (Note the algorithm selected is negative 5 in this case).

```
-mchecksum=0-FF@100,offset=0xFFFF,algorithm=-5,width=-2,polynomial=0x1021
```

In your project, call either the `crc_reflected_IO()` or `crc_reflected_poly()` functions, as shown previously.

5.3 Objdump

The `avr-objdump` application can display various information about object files.

The general form of the tool's command line is as follows:

```
avr-objdump [options] objfiles
```

where *`objfiles`* can be any object file, including an archive or output file. The tool is able to determine the format of the file specified.

The `--help` option shows all the command available for this application.

For AVR ELF files only, the `-Pmem-usage` option will show program and data memory usage in bytes and as a percentage of the memory space available.

A common usage of this tool is to obtain a full list file for the entire program. To do this, use the compiler's `-g` option when you build the project, then call the `avr-objdump` application with a command similar to the following.

```
avr-objdump -S -l a.out > avr.lst
```

This will create an `avr.lst` listing file from the default compiler output file, showing the original C source code and line number information in the listing.

6. Implementation-Defined Behavior

This section indicates the compiler's choice of behavior where the C standard indicates that the behavior is implementation defined.

6.1 Overview

ISO C requires a conforming implementation to document the choices for behaviors defined in the standard as "implementation-defined." The following sections list all such areas, the choices made for the compiler, and the corresponding section number from the ISO/IEC 9899:1999 (aka C99) standard (or ISO/IEC 9899:1990 (aka C90)).

6.2 Translation

ISO Standard:	"How a diagnostic is identified (3.10, 5.1.1.3)."
Implementation:	By default, when compiling on the command-line the following formats are used. The string (warning) is only displayed for warning messages. `filename:line:column:{error/warning}: message`
ISO Standard:	"Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2)."
Implementation:	The compiler will replace each leading or interleaved whitespace character sequences with a space. A trailing sequence of whitespace characters is replaced with a new-line.

6.3 Environment

ISO Standard:	"The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2)."
Implementation:	Multi-byte characters are not supported in source files.
ISO Standard:	"The name and type of the function called at program start-up in a freestanding environment (5.1.2.1)."
Implementation:	`int main (void);`
ISO Standard:	"The effect of program termination in a freestanding environment (5.1.2.1)."
Implementation:	Interrupts are disabled and the programs loops indefinitely
ISO Standard:	"An alternative manner in which the `main` function may be defined (5.1.2.2.1)."
Implementation:	`void main (void);`
ISO Standard:	"The values given to the strings pointed to by the argv argument to main (5.1.2.2.1)."
Implementation:	No arguments are passed to main. Reference to `argc` or `argv` is undefined.
ISO Standard:	"What constitutes an interactive device (5.1.2.3)."
Implementation:	Application defined.
ISO Standard:	"The set of signals, their semantics, and their default handling (7.14)."
Implementation:	Signals are not implemented.
ISO Standard:	"Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1)."

Implementation:	Signals are not implemented.
ISO Standard:	“Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program start-up (7.14.1.1).”
Implementation:	Signals are not implemented.
ISO Standard:	“The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5).”
Implementation:	The host environment is application defined.
ISO Standard:	“The manner of execution of the string by the system function (7.20.4.6).”
Implementation:	The host environment is application defined.

6.4 Identifiers

ISO Standard:	“Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).”
Implementation:	None.
ISO Standard:	“The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).”
Implementation:	All characters are significant.

6.5 Characters

ISO Standard:	“The number of bits in a byte (C90 3.4, C99 3.6).”
Implementation:	8.
ISO Standard:	“The values of the members of the execution character set (C90 and C99 5.2.1).”
Implementation:	The execution character set is ASCII.
ISO Standard:	“The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (C90 and C99 5.2.2).”
Implementation:	The execution character set is ASCII.
ISO Standard:	“The value of a `char` object into which has been stored any character other than a member of the basic execution character set (C90 6.1.2.5, C99 6.2.5).”
Implementation:	The value of the `char` object is the 8-bit binary representation of the character in the source character set. That is, no translation is done.
ISO Standard:	“Which of `signed char` or `unsigned char` has the same range, representation, and behavior as “plain” `char` (C90 6.1.2.5, C90 6.2.1.1, C99 6.2.5, C99 6.3.1.1).”
Implementation:	By default, `signed char` is functionally equivalent to plain char. If the CCI is specified, then the default is `unsigned char` The options `-funsigned-char` and `-fsigned-char` can be used to explicitly specify the type.
ISO Standard:	“The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (C90 6.1.3.4, C99 6.4.4.4, C90 and C99 5.1.1.2).”
Implementation:	The binary representation of the source character set is preserved to the execution character set.

ISO Standard:	"The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (C90 6.1.3.4, C99 6.4.4.4)."
Implementation:	The previous value is shifted left by eight, and the bit pattern of the next character is masked in. The final result is of type `int`. If the result is larger than can be represented by an `int`, a warning diagnostic is issued and the value truncated to `int` size.
ISO Standard:	"The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (C90 6.1.3.4, C99 6.4.4.4)."
Implementation:	Multi-byte characters are not supported in source files.
ISO Standard:	"The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (C90 6.1.3.4, C99 6.4.4.4)."
Implementation:	Multi-byte characters are not supported in source files.
ISO Standard:	"The current locale used to convert a wide string literal into corresponding wide character codes (C90 6.1.4, C99 6.4.5)."
Implementation:	Wide strings are not supported.
ISO Standard:	"The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (C90 6.1.4, C99 6.4.5)."
Implementation:	Multi-byte characters are not supported in source files.

6.6 Integers

ISO Standard:	"Any extended integer types that exist in the implementation (C99 6.2.5)."
Implementation:	The `__int24` and `__uint24` keywords designate a signed and unsigned, respectively, 24-bit integer type.
ISO Standard:	"Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (C99 6.2.6.2)."
Implementation:	All integer types are represented as two's complement, and all bit patterns are ordinary values.
ISO Standard:	"The rank of any extended integer type relative to another extended integer type with the same precision (C99 6.3.1.1)."
Implementation:	There are no extended integer types with the same precision.
ISO Standard:	"The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (C90 6.2.1.2, C99 6.3.1.3)."
Implementation:	When converting value X to a type of width N, the value of the result is the Least Significant N bits of the 2's complement representation of X. That is, X is truncated to N bits. No signal is raised.
ISO Standard:	"The results of some bitwise operations on signed integers (C90 6.3, C99 6.5)."

Implementation:	The right shift operator (>>) sign extends signed values. Thus, an object with the `signed int` value 0x0124 shifted right one bit will yield the value 0x0092 and the value 0x8024 shifted right one bit will yield the value 0xC012. Right shifts of unsigned integral values always clear the MSb of the result. Left shifts (<< operator), signed or unsigned, always clear the LSb of the result. Other bitwise operations act as if the operand was unsigned.

6.7 Floating-Point

ISO Standard:	"The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (C90 and C99 5.2.4.2.2)."
Implementation:	The accuracy is unknown.
ISO Standard:	"The rounding behaviors characterized by non-standard values of `FLT_ROUNDS` (C90 and C99 5.2.4.2.2)."
Implementation:	No such values are used.
ISO Standard:	"The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD` (C90 and C99 5.2.4.2.2)."
Implementation:	No such values are used.
ISO Standard:	"The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (C90 6.2.1.3, C99 6.3.1.4)."
Implementation:	The integer is rounded to the nearest floating point representation.
ISO Standard:	"The direction of rounding when a floating-point number is converted to a narrower floating-point number (C90 6.2.1.4, 6.3.1.5)."
Implementation:	A floating-point number is rounded down when converted to a narrow floating-point value.
ISO Standard:	"How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (C90 6.1.3.1, C99 6.4.4.2)."
Implementation:	Not applicable; `FLT_RADIX` is a power of 2.
ISO Standard:	"Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (C99 6.5)."
Implementation:	The pragma is not implemented.
ISO Standard:	"The default state for the `FENV_ACCESS` pragma (C99 7.6.1)."
Implementation:	This pragma is not implemented.
ISO Standard:	"Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (C99 7.6, 7.12)."
Implementation:	None supported.
ISO Standard:	"The default state for the `FP_CONTRACT` pragma (C99 7.12.2)."
Implementation:	This pragma is not implemented.
ISO Standard:	"Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (C99 F.9)."
Implementation:	The exception is not raised.

ISO Standard:	"Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (C99 F.9)."
Implementation:	The exception is not raised.

6.8 Arrays and Pointers

ISO Standard:	"The result of converting a pointer to an integer or vice versa (C90 6.3.4, C99 6.3.2.3)."
Implementation:	A cast from an integer to a pointer or vice versa results uses the binary representation of the source type, reinterpreted as appropriate for the destination type. If the source type is larger than the destination type, the most significant bits are discarded. When casting from a pointer to an integer, if the source type is smaller than the destination type, the result is sign extended. When casting from an integer to a pointer, if the source type is smaller than the destination type, the result is extended based on the signedness of the source type.
ISO Standard:	"The size of the result of subtracting two pointers to elements of the same array (C90 6.3.6, C99 6.5.6)."
Implementation:	The signed integer result will have the same size as the pointer operands in the subtraction.

6.9 Hints

ISO Standard:	"The extent to which suggestions made by using the `register` storage-class specifier are effective (C90 6.5.1, C99 6.7.1)."
Implementation:	The `register` storage class can be used to locate certain objects in a register (see 4.4.6 Variables in Registers).
ISO Standard:	"The extent to which suggestions made by using the `inline` function specifier are effective (C99 6.7.4)."
Implementation:	A function might be inlined if a PRO-licensed compiler has the optimizers set to level 2 or higher. In other situations, the function will not be inlined.

6.10 Structures, Unions, Enumerations, and Bit-Fields

ISO Standard:	"Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field (C90 6.5.2, C90 6.5.2.1, C99 6.7.2, C99 6.7.2.1)."
Implementation:	A plain `int` bit-field is treated as an unsigned integer. The `-fsigned-bitfields` option can be used to treat bit-fields as signed.
ISO Standard:	"Allowable bit-field types other than `_Bool`, signed int, and unsigned int (C99 6.7.2.1)."
Implementation:	All integer types are allowed.
ISO Standard:	"Whether a bit-field can straddle a storage unit boundary (C90 6.5.2.1, C99 6.7.2.1)."
Implementation:	A bit-field can straddle a storage unit.
ISO Standard:	"The order of allocation of bit-fields within a unit (C90 6.5.2.1, C99 6.7.2.1)."
Implementation:	The first bit-field defined in a structure is allocated the LSb position in the storage unit. Subsequent bit-fields are allocated higher-order bits.

ISO Standard:	“The alignment of non-bit-field members of structures (C90 6.5.2.1, C99 6.7.2.1).”
Implementation:	No alignment is performed.
ISO Standard:	“The integer type compatible with each enumerated type (C90 6.5.2.2, C99 6.7.2.2).”
Implementation:	A `signed int` or `unsigned int` can be chosen to represent an enumerated type.

6.11 Qualifiers

ISO Standard:	“What constitutes an access to an object that has `volatile`-qualified type (C90 6.5.3, C99 6.7.3).”
Implementation:	Each reference to the identifier of a `volatile`-qualified object constitutes one access to the object.

6.12 Pre-Processing Directives

ISO Standard:	“How sequences in both forms of header names are mapped to headers or external source file names (C90 6.1.7, C99 6.4.7).”
Implementation:	The character sequence between the delimiters is considered to be a string which is a file name for the host environment.
ISO Standard:	“Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (C90 6.8.1, C99 6.10.1).”
Implementation:	Yes.
ISO Standard:	“Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (C90 6.8.1, C99 6.10.1).”
Implementation:	Yes.
ISO Standard:	“The places that are searched for an included `< >` delimited header, and how the places are specified or the header is identified (C90 6.8.2, C99 6.10.2).”
Implementation:	The preprocessor searches any directory specified using the `-I` option, then, provided the `-nostdinc` option has not been used, the standard compiler include directory, `<install directory>/avr/avr/ include`.
ISO Standard:	“How the named source file is searched for in an included `" "` delimited header (C90 6.8.2, C99 6.10.2).”
Implementation:	The compiler first searches for the named file in the directory containing the including file, then the directories which are searched for a `< >` delimited header.
ISO Standard:	“The method by which preprocessing tokens are combined into a header name (C90 6.8.2, C99 6.10.2).”
Implementation:	All tokens, including whitespace, are considered part of the header file name. Macro expansion is not performed on tokens inside the delimiters.
ISO Standard:	“The nesting limit for `#include` processing (C90 6.8.2, C99 6.10.2).”
Implementation:	No limit.
ISO Standard:	“Whether the `#` operator inserts a `\` character before the `\` character that begins a universal character name in a character constant or string literal (6.10.3.2).”

Implementation:	No.
ISO Standard:	“The behavior on each recognized non-STDC `#pragma` directive (C90 6.8.6, C99 6.10.6).”
Implementation:	See 4.13.3 Pragma Directives
ISO Standard:	“The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (C90 6.8.8, C99 6.10.8).”
Implementation:	The date and time of translation are always available.

6.13 Library Functions

ISO Standard:	“Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).”
Implementation:	See 7. Library Functions.
ISO Standard:	“The format of the diagnostic printed by the `assert` macro (7.2.1.1).”
Implementation:	`Assertion failed: (`*`message`*`), function `*`function`*`, file `*`file`*`, line `*`line`*`.\n`. The `function `*`function`* component is skipped if `__func__` is unavailable.
ISO Standard:	“The representation of floating-point exception flags stored by the `fegetexceptflag` function (7.6.2.2).”
IImplementation:	Unimplemented.
ISO Standard:	“Whether the `feraiseexcept` function raises the inexact exception in addition to the overflow or underflow exception (7.6.2.3).”
Implementation:	Unimplemented.
ISO Standard:	“Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).”
Implementation:	None.
ISO Standard:	“The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12).”
Implementation:	Unimplemented.
ISO Standard:	“Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1).”
Implementation:	None.
ISO Standard:	“The values returned by the mathematics functions on domain errors (7.12.1).”
Implementation:	`errno` is set to `EDOM` on domain errors.
ISO Standard:	“Whether the mathematics functions set `errno` to the value of the macro `ERANGE` on overflow and/or underflow range errors (7.12.1).”
Implementation:	Yes
ISO Standard:	“Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero (7.12.10.1).”
Implementation:	The first argument is returned.
ISO Standard:	“The base-2 logarithm of the modulus used by the remquo function in reducing the quotient (7.12.10.3).”

Implementation:	Unimplemented.
ISO Standard:	Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).
Implementation:	Signals are not implemented.
ISO Standard:	The null pointer constant to which the macro `NULL` expands (7.17).
Implementation:	`((void *)0)`
ISO Standard:	“Whether the last line of a text stream requires a terminating new-line character (7.19.2).”
Implementation:	Streams are not implemented.
ISO Standard:	“Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).”
Implementation:	Streams are not implemented.
ISO Standard:	“The number of null characters that may be appended to data written to a binary stream (7.19.2).”
Implementation:	Streams are not implemented.
ISO Standard:	“Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).”
Implementation:	Streams are not implemented.
ISO Standard:	“Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).”
Implementation:	Streams are not implemented.
ISO Standard:	“The characteristics of file buffering (7.19.3).”
Implementation:	File handling is not implemented.
ISO Standard:	“Whether a zero-length file actually exists (7.19.3).”
Implementation:	File handling is not implemented.
ISO Standard:	“The rules for composing valid file names (7.19.3).”
Implementation:	File handling is not implemented.
ISO Standard:	“Whether the same file can be open multiple times (7.19.3).”
Implementation:	File handling is not implemented.
ISO Standard:	“The nature and choice of encodings used for multibyte characters in files (7.19.3).”
Implementation:	File handling is not implemented.
ISO Standard:	“The effect of the remove function on an open file (7.19.4.1).”
Implementation:	File handling is not implemented.
ISO Standard:	“The effect if a file with the new name exists prior to a call to the rename function (7.19.4.2).”
Implementation:	File handling is not implemented.
ISO Standard:	“Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).”
Implementation:	File handling is not implemented.
ISO Standard:	“What happens when the `tmpnam` function is called more than `TMP_MAX` times (7.19.4.4).”

Implementation:	File handling is not implemented.
ISO Standard:	“Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).”
Implementation:	File handling is not implemented.
ISO Standard:	“The style used to print an infinity or NaN and the meaning of the n-char-sequence, if that style is printed for a NaN (7.19.6.1, 7.24.2.1).”
Implementation:	The values are printed as the nearest number.
ISO Standard:	“The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1).”
Implementation:	Functionally equivalent to `%lx`.
ISO Standard:	“The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2).”
Implementation:	Streams are not implemented.
ISO Standard:	“The set of sequences matched by the `%p` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2).”
Implementation:	Streams are not implemented.
ISO Standard:	“The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4).”
Implementation:	Streams are not implemented.
ISO Standard:	“The meaning of the n-char-sequence in a string converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.20.1.3, 7.24.4.1.1).”
Implementation:	No meaning is attached to the sequence.
ISO Standard:	“Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1).”
Implementation:	No.
ISO Standard:	“Whether the `calloc`, `malloc`, and `realloc` functions return a Null Pointer or a pointer to an allocated object when the size requested is zero (7.20.3).”
Implementation:	The requested size is bumped to the lowest allowable, two bytes. If this can be successfully allocated, a pointer to the space is returned; otherwise `NULL` is returned.
ISO Standard:	“Whether open output streams are flushed, open streams are closed, or temporary files are removed when the abort function is called (7.20.4.1).”
Implementation:	Streams are not implemented.
ISO Standard:	“The termination status returned to the host environment by the abort function (7.20.4.1).”
Implementation:	The host environment is application defined.
ISO Standard:	“The value returned by the system function when its argument is not a Null Pointer (7.20.4.5).”
Implementation:	The host environment is application defined.
ISO Standard:	“The local time zone and Daylight Saving Time (7.23.1).”
Implementation:	Application defined.
ISO Standard:	“The range and precision of times representable in `clock_t` and `time_t` (7.23)”

Implementation:	the `time_t` type is used to hold a number of seconds and is defined as a `long` type; `clock_t` is not defined.
ISO Standard:	"The era for the clock function (7.23.2.1)."
Implementation:	Application defined.
ISO Standard:	"The replacement string for the `%Z` specifier to the `strftime`, `strfxtime`, `wcsftime`, and `wcsfxtime` functions in the "C" locale (7.23.3.5, 7.23.3.6, 7.24.5.1, 7.24.5.2)."
Implementation:	These functions are unimplemented.
ISO Standard:	"Whether or when the trigonometric, hyperbolic, base-e exponential, base-e logarithmic, error, and log gamma functions raise the inexact exception in an IEC 60559 conformant implementation (F.9)."
Implementation:	No.
ISO Standard:	"Whether the functions in `<math.h>` honor the Rounding Direction mode (F.9)."
Implementation:	The rounding mode is not forced.

6.14 Architecture

ISO Standard:	"The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (C90 and C99 5.2.4.2, C99 7.18.2, 7.18.3)."
Implementation:	See 7.7 <float.h> Floating-Point Characteristics, 7.10 <limits.h> Implementation-Defined Limits and the header files in `<install directory>/avr/avr/include/c99`.
ISO Standard:	"The number, order, and encoding of bytes in any object, when not explicitly specified in the standard (C99 6.2.6.1)."
Implementation:	Little endian, populated from Least Significant Byte first.
ISO Standard:	"The value of the result of the `sizeof` operator (C90 6.3.3.4, C99 6.5.3.4)."
Implementation:	The type of the result is equivalent to `unsigned int`.

7. Library Functions

The functions, variables, types, and preprocessor macros defined by the standard compiler library are summarized in this chapter, listed under the header file which declares them.

7.1 <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.

7.1.1 assert Macro

If the argument is false, an assertion failure message is printed to stderr 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 and provided that __ASSERT_USE_STDERR has been defined before the inclusion of <assert.h>, a message is printed to stderr.

The message includes the source file name (__FILE__), the source line number (__LINE__), the expression being evaluated and the message. The macro then calls the function abort().

Assertion testing may be turned off without removing the code by defining NDEBUG before including <assert.h>. If the macro NDEBUG is defined, assert() is ignored and no code is generated.

Example

```
#include <assert.h>

int main(void)
{
  int a;

  a = 2 * 2;
  assert(a == 4); /* if true-nothing prints */
  assert(a == 6); /* if false-print message */
  /* and abort */
}
```

Example Output

```
sampassert.c:9 a == 6 -- assertion failed
ABRT
```

with _VERBOSE_DEBUGGING defined:

```
sampassert.c:8 a == 4 -- OK
sampassert.c:9 a == 6 -- assertion failed
ABRT
```

7.2 <boot.h> Bootloader Functions

The macros in this module provide a C language interface to the bootloader support functionality available on some AVR devices. These macros are designed to work with all sizes of flash memory.

Global interrupts are not automatically disabled for these macros. It is left up to the programmer to do this. Also see the processor data sheet for caveats on having global interrupts enabled during writing of flash memory.

7.2.1 boot_is_spm_interrupt Macro

Check if the store program memory interrupt is enabled.

Include

`<avr/boot.h>`

Prototype

```
int boot_is_spm_interrupt(void);
```

Remarks

This macro returns 1 if the SPM interrupt enable bit is set; 0 otherwise.

Example

```
#include <avr/boot.h>
int main(void)
{
  if(boot_is_spm_interrupt())
    disableMode();
}
```

7.2.2 boot_lock_bits_set Macro

Set the bootloader lock bits.

Include

`<avr/boot.h>`

Prototype

```
int boot_lock_bits_set(unsigned char mask);
```

Remarks

This macro sets those bits specified by the bit mask in the SPM control register. The bootloader lock bits, once set, can only be cleared by a chip erase, which in turn will also erase the boot loader itself.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_lock_bits_set(_BV(BLB11)|_BV(BLB12));
}
```

7.2.3 boot_lock_bits_set_safe Macro

Set the bootloader lock bits.

Include

`<avr/boot.h>`

Prototype

```
int boot_lock_bits_set_safe(unsigned char mask);
```

Remarks

This macro sets those bits specified by the bit mask in the SPM control register after ensuring that EEPROM and PSM operations are complete. The bootloader lock bits, once set, can only be cleared by a chip erase, which in turn will also erase the boot loader itself.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_lock_bits_set_safe(_BV(BLB11)|_BV(BLB12));
}
```

7.2.4 boot_lock_fuse_bits_get Macro

Read the lock or fuse bits.

Include

`<avr/boot.h>`

Prototype

```
int boot_lock_fuse_bits_get(unsigned int address);
```

Remarks

This macro returns the value of the lock or fuse bits at the specified address. The *address* argument can be one of `GET_LOW_FUSE_BITS`, `GET_LOCK_BITS`, `GET_EXTENDED_FUSE_BITS`, or `GET_HIGH_FUSE_BITS`. The bits returned are the physical values, this if a bit position was 0, it impiles that the corresponding location was programmed.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_lock_bits_set(_BV(BLB11)|_BV(BLB12));
}
```

7.2.5 boot_page_erase Macro

Erase the flash page that contains address.

Include

`<avr/boot.h>`

Prototype

```
void boot_page_erase(unsigned int address);
```

Remarks

Erase the flash page that contains byte address specified.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_page_erase(0x100);
}
```

7.2.6 boot_page_erase_safe Macro

Erase the flash page that contains address.

Include

<avr/boot.h>

Prototype

```
void boot_page_erase_safe(unsigned int address);
```

Remarks

Erase the flash page that contains byte address specified ater ensuring that EEPROM and SPM operations are complete.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_page_erase_safe(0x100);
}
```

7.2.7 boot_page_fill Macro

Place a word in the bootloader temporary page buffer corresponding to the flash address.

Include

<avr/boot.h>

Prototype

```
void boot_page_fill(unsigned int address, unsign int value);
```

Remarks

This macro places the value specified in the bootloader temporary page buffer corresponding to the flash address. The address is a byte address; however, AVR devices writes words to the buffer, so for each 16-bit word to be written, increment the address by 2. The LSB of the data is written to the lower address; the MSB of the data is written to the higher address.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_page_fill(0x100, 0x55);
}
```

7.2.8 boot_page_fill_safe Macro

Place a word in the bootloader temporary page buffer corresponding to the flash address.

Include

<avr/boot.h>

Prototype

```
void boot_page_fill_safe(unsigned int address, unsign int value);
```

Remarks

This macro places the value specified in the bootloader temporary page buffer corresponding to the flash address after ensuring that EEPROM and SPM operations have been completed. The address is a byte address; however, AVR devices writes words to the buffer, so for each 16-bit word to be written, increment the address by 2. The LSB of the data is written to the lower address; the MSB of the data is written to the higher address.

Example

```
#include <avr/boot.h>
int main(void)
```



```
{
  boot_page_fill_safe(0x100, 0x55);
}
```

7.2.9 boot_page_write Macro

Write the bootloader temporary buffer to the flash page.

Include

`<avr/boot.h>`

Prototype

```
void boot_page_write(unsigned int address);
```

Remarks

This macro writes the bootloader temporary buffer to the flash page that corresponding to the specified address. The address is a byte address.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_page_write(0x55);
}
```

7.2.10 boot_page_write_safe Macro

Write the bootloader temporary buffer to the flash page.

Include

`<avr/boot.h>`

Prototype

```
void boot_page_write_safe(unsigned int address);
```

Remarks

This macro writes the bootloader temporary buffer to the flash page that corresponding to the specified address after ensuring that EEPROM and SPM operations have completed. The address is a byte address.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_page_write_safe(0x55);
}
```

7.2.11 boot_rww_busy Macro

Check if the read-while-write (RWW) section is busy.

Include

`<avr/boot.h>`

Prototype

```
int boot_rww_busy(void);
```

Remarks

This macro returns 1 if the read-while-write section busy bit is set; 0 otherwise.

Example

```
#include <avr/boot.h>
int main(void)
{
  while(boot_rww_busy())
    waitRWW();
}
```

7.2.12 boot_rww_enable Macro

Enable the read-while-write (RWW) section.

Include

`<avr/boot.h>`

Prototype

```
void boot_rww_enable(void);
```

Remarks

This macro enables the read-while-write section so that while it is being erased or written, code may still be executed from the no-read-while-write (NRWW) section.

Example

```
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>

void boot_program_page (uint32_t page, uint8_t *buf) {
  uint16_t i;
  uint8_t sreg;

  sreg = SREG;
  cli();
  eeprom_busy_wait ();
  boot_page_erase (page);
  boot_spm_busy_wait ();      // Wait until the memory is erased.
  for (i=0; i<SPM_PAGESIZE; i+=2) {
    // Set up little-endian word.
    uint16_t w = *buf++;
    w += (*buf++) << 8;
    boot_page_fill (page + i, w);
  }
  boot_page_write (page);     // Store buffer in flash page.
  boot_spm_busy_wait();       // Wait until the memory is written.
  // Reenable RWW-section again. We need this if we want to jump back
  // to the application after bootloading.
  boot_rww_enable ();
  // Re-enable interrupts (if they were ever enabled).
  SREG = sreg;
}
```

7.2.13 boot_rww_enable_safe Macro

Enable the read-while-write (RWW) section.

Include

`<avr/boot.h>`

Prototype

```
void boot_rww_enable_safe(void);
```

Remarks

This macro enables the read-while-write section after ensuring that EEPROM and PSM operations are complete. When enabled and the read-while-write section is erased or written, code may still be executed from the no-read-while-write (NRWW) section.

Example

```
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>

void boot_program_page (uint32_t page, uint8_t *buf) {
  uint16_t i;
  uint8_t sreg;

  sreg = SREG;
  cli();
  eeprom_busy_wait ();
  boot_page_erase (page);
  boot_spm_busy_wait ();      // Wait until the memory is erased.
  for (i=0; i<SPM_PAGESIZE; i+=2) {
    // Set up little-endian word.
    uint16_t w = *buf++;
    w += (*buf++) << 8;
    boot_page_fill (page + i, w);
  }
  boot_page_write (page);     // Store buffer in flash page.
  boot_spm_busy_wait();       // Wait until the memory is written.
  // Reenable RWW-section again. We need this if we want to jump back
  // to the application after bootloading.
  boot_rww_enable_safe();
  // Re-enable interrupts (if they were ever enabled).
  SREG = sreg;
}
```

7.2.14 boot_signature_byte_get Macro

Read the signature row byte at address.

Include

`<avr/boot.h>`

Prototype

```
unsigned char boot_signature_byte_get(unsigned int address);
```

Remarks

This macro returns the signature row byte at the specified address. For some devices, this macro can obtain the factory-stored oscillator calibration bytes. The address argument can be 0-0x1f, as documented by the datasheet.

Example

```
#include <avr/boot.h>
int main(void)
{
  unsigned char signature;

  signature = boot_signature_byte_get(0x1f);
}
```

7.2.15 boot_spm_busy Macro

Check if the SPM is busy.

Include

`<avr/boot.h>`

Prototype

```
int boot_spm_busy(void);
```

Remarks

This macro returns 1 if the SPM enable is set; 0 otherwise.

Example

```
#include <avr/boot.h>
int main(void)
{
  if(boot_spm_busy())
    altMode();
}
```

7.2.16 boot_spm_busy_wait Macro

Wait while the SPM is busy.

Include

`<avr/boot.h>`

Prototype

```
int boot_spm_busy_wait(void);
```

Remarks

This waits until the SPM is not enabled.

Example

```
#include <avr/boot.h>
int main(void)
{
  boot_spm_busy_wait();
}
```

7.2.17 boot_spm_interrupt_disable Macro

Disable SPM interrupts.

Include

`<avr/boot.h>`

Prototype

```
void boot_spm_interrupt_disable(void);
```

Remarks

This macro disables interrupts associated with SPM.

Example

```

#include <avr/boot.h>

void main(void) {
  boot_spm_interrupt_disable();
}
```

7.2.18 boot_spm_interrupt_enable Macro

Disable SPM interrupts.

Include

`<avr/boot.h>`

Prototype

```
void boot_spm_interrupt_enable(void);
```

Remarks

This macro enables interrupts associated with SPM.

Example

```
#include <avr/boot.h>

void main(void) {
  boot_spm_interrupt_enable();
}
```

7.3 <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.

7.3.1 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

```
#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
```

7.3.2 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

```
#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
```

7.3.3 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

```
#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
```

7.3.4 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

```
#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
```

7.3.5 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

```
#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
```

7.3.6 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

```
#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
```

7.3.7 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

```
#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
```

7.3.8 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 it is in the range 0x20 to 0x7e inclusive.

Example

```
#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
```

7.3.9 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

```
#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
```

7.3.10 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

```
#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
```

7.3.11 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

```
#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
```

7.3.12 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 leading 0x because 0x is the prefix for a hexadecimal number but is not an actual hexadecimal digit.

Example

```
#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
```

7.3.13 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

```
#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
```

7.3.14 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

```
#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
```

7.4 <cpufunc.h> CPU Related Functions

The header file `cpufunc.h` consists of functions that relate to instruction execution.

7.4.1 _MemoryBarrier Macro

Implements a read/write memory barrier.

Include

`<avr/cpufunc.h>`

Prototype

```
void _MemoryBarrier(void);
```

Remarks

A memory barrier instructs the compiler to not cache any memory data in registers beyond the barrier. This can sometimes be more effective than blocking certain optimizations by declaring some object with a `volatile` qualifier.

Example

```
#include <avr/cpufunc.h>
int main(void)
{
  data = readMode();
  _MemoryBarrier();
  processData(&data);
}
```

7.4.2 _NOP macro

A macro that performs no operation.

Include

`<avr/cpufunc.h>`

Prototype

```
void _NOP(void);
```

Remarks

This macro executes a `nop` instruction.

This macro should not be used to implement delays. It is better use the functions from `<util/delay_basic.h>` or `<util/delay.h>` for this. As the `nop` instruction is not optimized away, it can be reliably used as the location for a breakpoint for debugging purposes.

Example

```
#include <avr/cpufunc.h>
int main(void)
{
  while(1)
    _NOP();
}
```

7.5 <delay.h> Delay Functions

The header file `delay.h` consists of functions that generate delays in program execution.

7.5.1 _delay_ms Function

Delay for the specified time.

Include

`<avr/delay.h>`

Prototype

```
void _delay_ms(double ms);
```

Arguments

`ms`	The time in milli seconds to delay.

Remarks

This function delays execution by using the `_delay_loop_2()` function. The macro `F_CPU` should be defined as a constant that specifies the CPU clock frequency (in Hertz). The compiler optimizers must be enabled for accurate delay times.

The maximal possible delay is 4294967.295 ms/F_CPU in MHz. Requesting values greater than the maximal possible delay will result in overflow and a delay of 0us.

Conversion of the requested number of milli seconds into clock cycles may not always result in integer value. By default, the number of clock cycles is rounded up to next integer. This ensures that there is at least the requested amount of delay.

By defining the macro `__DELAY_ROUND_DOWN__`, or `__DELAY_ROUND_CLOSEST__`, before including this header file, the algorithm can be made to round down, or round to closest integer, respectively.

Example

```
#define F_CPU 4000000UL
#include <avr/delay.h>
int main(void)
{
 _delay_ms(20);  // delay for 20 milli seconds
}
```

7.5.2 _delay_us Function

Delay for the specified time.

Include

`<avr/delay.h>`

Prototype

```
void _delay_us(double us);
```

Arguments

us	The time in micro seconds to delay.

Remarks

This function delays execution by using the `_delay_loop_1()` function. The macro `F_CPU` should be defined as a constant that specifies the CPU clock frequency (in Hertz). The compiler optimizers must be enabled for accurate delay times.

The maximal possible delay is 4294967.295 us/F_CPU in MHz. Requesting values greater than the maximal possible delay will result in overflow and a delay of 0us.

Conversion of the requested number of micro seconds into clock cycles may not always result in integer value. By default, the number of clock cycles is rounded up to next integer. This ensures that there is at least the requested amount of delay.

By defining the macro `__DELAY_ROUND_DOWN__`, or `__DELAY_ROUND_CLOSEST__`, before including this header file, the algorithm can be made to round down, or round to closest integer, respectively.

Example

```
#define F_CPU 4000000UL
#include <avr/delay.h>
int main(void)
{
 _delay_us(20);  // delay for 20 micro seconds
}
```

7.6 <errno.h> Errors

The header file `errno.h` consists of macros that provide error codes that are reported by certain library functions (see individual functions). The variable `errno` may return any value greater than 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 before another function call could change the value. At program start-up, `errno` is zero. Library functions will never set `errno` to zero.

7.6.1 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.

7.6.2 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.

7.6.3 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.

7.6.4 errno Variable

Contains the value of an error when an error occurs in a function.

Include

<errno.h>

Remarks

The variable errno is set to a non-zero integer value by a library function when an error occurs. At program start-up, errno is set to zero. errno should be reset to zero prior to calling a function that sets it.

7.7 <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.

7.7.1 DBL_DIG Macro

Number of decimal digits of precision in a double precision floating-point value.

Include

<float.h>

Value

Remarks

7.7.2 DBL_EPSILON Macro

The difference between 1.0 and the next larger representable double precision floating-point value.

Include

<float.h>

Value

Remarks

7.7.3 DBL_MANT_DIG Macro

Number of base-FLT_RADIX digits in a double precision floating-point significand.

Include

<float.h>

Value

The value 24.

Remarks

By default, a double type is the same size as a float type (32-bit representation).

7.7.4 DBL_MAX Macro

Maximum finite double precision floating-point value.

Include

<float.h>

Value

Remarks

By default, a double type is the same size as a float type (32-bit representation). The -fno-short-double switch allows the IEEE 64-bit representation to be used for a double precision floating-point value.

7.7.5 DBL_MAX_10_EXP Macro

Maximum integer value for a double precision floating-point exponent in base 10.

Include

<float.h>

Value

Remarks

7.7.6 DBL_MAX_EXP Macro

Maximum integer value for a double precision floating-point exponent in base FLT_RADIX.

Include

<float.h>

Value

Remarks

7.7.7 DBL_MIN Macro

Minimum double precision floating-point value.

Include

<float.h>

Value

Remarks

7.7.8 DBL_MIN_10_EXP Macro

Minimum negative integer value for a double precision floating-point exponent in base 10.

Include

<float.h>

Value

Remarks

7.7.9 DBL_MIN_EXP Macro

Minimum negative integer value for a double precision floating-point exponent in base `FLT_RADIX`.

Include

`<float.h>`

Value

Remarks

7.7.10 FLT_DIG Macro

Number of decimal digits of precision in a single precision floating-point value.

Include

`<float.h>`

Value

The value 6.

7.7.11 FLT_EPSILON Macro

The difference between 1.0 and the next larger representable single precision floating-point value.

Include

`<float.h>`

Value

The value 1.192093e-07.

7.7.12 FLT_MANT_DIG Macro

Number of base-`FLT_RADIX` digits in a single precision floating-point significand.

Include

`<float.h>`

Value

The value 24.

7.7.13 FLT_MAX Macro

Maximum finite single precision floating-point value.

Include

`<float.h>`

Value

The value 3.402823e+38.

7.7.14 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.

7.7.15 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.

7.7.16 FLT_MIN Macro

Minimum single precision floating-point value.

Include

`<float.h>`

Value

The value 1.175494e-38.

7.7.17 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.

7.7.18 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.

7.7.19 FLT_RADIX Macro

Radix of the exponent representation.

Include

`<float.h>`

Value

2

Remarks

The value 2 (binary).

7.7.20 FLT_ROUNDS Macro

Represents the rounding mode for floating-point operations.

-1	indeterminable
0	toward zero
1	to nearest representable value
2	toward positive infinity

3	toward negative infinity

Include

`<float.h>`

Value

1

Remarks

The value 1 (nearest representable value).

7.7.21 LDBL_DIG Macro

Number of decimal digits of precision in a long double precision floating-point value.

Include

`<float.h>`

Value

7.7.22 LDBL_EPSILON Macro

The difference between 1.0 and the next larger representable long double precision floating-point value.

Include

`<float.h>`

Value

7.7.23 LDBL_MANT_DIG Macro

Number of base-`FLT_RADIX` digits in a long double precision floating-point significand.

Include

`<float.h>`

Value

7.7.24 LDBL_MAX Macro

Maximum finite long double precision floating-point value.

Include

`<float.h>`

Value

7.7.25 LDBL_MAX_10_EXP Macro

Maximum integer value for a long double precision floating-point exponent in base 10.

Include

`<float.h>`

Value

7.7.26 LDBL_MAX_EXP Macro

Maximum integer value for a long double precision floating-point exponent in base `FLT_RADIX`.

Include

`<float.h>`

Value

7.7.27 LDBL_MIN Macro

Minimum long double precision floating-point value.

Include

`<float.h>`

Value

7.7.28 LDBL_MIN_10_EXP Macro

Minimum negative integer value for a long double precision floating-point exponent in base 10.

Include

`<float.h>`

Value

7.7.29 LDBL_MIN_EXP Macro

Minimum negative integer value for a long double precision floating-point exponent in base `FLT_RADIX`.

Include

`<float.h>`

Value

7.8 <inttypes.h> Integer Format Conversion

The content of the header file `inttypes.h` consists of format specifiers, which can be used with `printf`, and functions that work with greatest-width integer types.

7.8.1 Print Format Macros for Signed Integers

The macros in following table expand to character string literals and can be used with the `printf` family of functions when printing signed integer values.

Macro Name	Description	Value
PRId8	Decimal placeholder string for a signed 8-bit integer type.	"d"
PRId16	Decimal placeholder string for a signed 16-bit integer type.	"d"
PRId32	Decimal placeholder string for a signed 32-bit integer type.	"ld"
PRId64	Decimal placeholder string for a signed 64-bit integer type, where supported.	"lld"
PRIi8	Integer placeholder string for a signed 8-bit integer type.	"i"
PRIi16	Integer placeholder string for a signed 16-bit integer type.	"i"
PRIi32	Integer placeholder string for a signed 32-bit integer type	"li"
PRIi64	Integer placeholder string for a signed 64-bit integer type, where supported.	"lli"

...........continued

Macro Name	Description	Value
PRIdFAST8	Decimal placeholder string for the fastest signed integer type with a width of at least 8.	"d"
PRIdFAST16	Decimal placeholder string for the fastest signed integer type with a width of at least 16.	"d"
PRIdFAST32	Decimal placeholder string for the fastest signed integer type with a width of at least 32.	"ld"
PRIdFAST64	Decimal placeholder string for the fastest signed integer type with a width of at least 64, where supported.	"lld"
PRIiFAST8	Integer placeholder string for the fastest signed integer type with a width of at least 8 bits.	"i"
PRIiFAST16	Integer placeholder string for the fastest signed integer type with a width of at least 16.	"i"
PRIiFAST32	Integer placeholder string for the fastest signed integer type with a width of at least 32.	"li"
PRIiFAST64	Integer placeholder string for the fastest signed integer type with a width of at least 64, where supported.	"lli"
PRIdLEAST8	Decimal placeholder string for a signed integer type with a width of at least 8 bits.	"d"
PRIdLEAST16	Decimal placeholder string for a signed integer type with a width of at least 16 bits.	"d"
PRIdLEAST32	Decimal placeholder string for a signed integer type with a width of at least 32 bits.	"ld"
PRIdLEAST64	Decimal placeholder string for a signed integer type with a width of at least 64 bits, where supported.	"lld"
PRIiLEAST8	Integer placeholder string for a signed integer type with a width of at least 8 bits.	"i"
PRIiLEAST16	Integer placeholder string for a signed integer type with a width of at least 16 bits.	"i"
PRIiLEAST32	Integer placeholder string for a signed integer type with a width of at least 32 bits.	"li"

...........continued		
Macro Name	**Description**	**Value**
`PRIiLEAST64`	Integer placeholder string for a signed integer type with a width of at least 64 bits, where supported.	`"lli"`
`PRIdMAX`	Decimal placeholder string for a signed integer type with maximum width.	`"lld"`where 64-bit integers are supported; `"ld"` otherwise
`PRIiMAX`	Integer placeholder string for a signed integer type with maximum width.	`"lli"`where 64-bit integers are supported; `"ld"` otherwise
`PRIdPTR`	Decimal placeholder string for `intptr_t`.	`"ld"`
`PRIiPTR`	Integer placeholder string for `intptr_t`.	`"li"`

7.8.2 Print Format Macros for Unsigned Integers

The macros in following table expand to character string literals and can be used with the `printf` family of functions when printing unsigned integer values.

Macro Name	Description	Value
`PRIo8`	Octal placeholder string for an unsigned 8-bit integer type.	`"o"`
`PRIo16`	Octal placeholder string for an unsigned 16-bit integer type.	`"o"`
`PRIo32`	Octal placeholder string for an unsigned 32-bit integer type.	`"lo"`
`PRIo64`	Octal placeholder string for an unsigned 64-bit integer type, where supported.	`"llo"`
`PRIu8`	Unsigned placeholder string for an unsigned 8-bit integer type.	`"u"`
`PRIu16`	Unsigned placeholder string for an unsigned 16-bit integer type.	`"u"`
`PRIu32`	Unsigned placeholder string for an unsigned 32-bit integer type.	`"lu"`
`PRIu64`	Unsigned placeholder string for an unsigned 64-bit integer type, where supported.	`"llu"`
`PRIx8`/ **`PRIX8`**	Hexadecimal placeholder string for an unsigned 8-bit integer type.	`"x"/"X"`
`PRIx16`/ **`PRIX16`**	Hexadecimal placeholder string for an unsigned 16-bit integer type.	`"x"/"X"`
`PRIx32`/ **`PRIX32`**	Hexadecimal placeholder string for an unsigned 32-bit integer type.	`"lx"/"X"`

...........continued		
Macro Name	**Description**	**Value**
`PRIx64`/ `PRIX64`	Hexadecimal placeholder string for an unsigned 64-bit integer type, where supported.	`"llx"`/`"X"`
`PRIoFAST8`	Octal placeholder string for the fastest signed integer type with a width of at least 8.	`"o"`
`PRIoFAST16`	Octal placeholder string for the fastest signed integer type with a width of at least 16.	`"o"`
`PRIoFAST32`	Octal placeholder string for the fastest signed integer type with a width of at least 32.	`"lo"`
`PRIoFAST64`	Octal placeholder string for the fastest signed integer type with a width of at least 64, where supported.	`"llo"`
`PRIuFAST8`	Unsigned placeholder string for the fastest signed integer type with a width of at least 8 bits.	`"u"`
`PRIuFAST16`	Unsigned placeholder string for the fastest signed integer type with a width of at least 16.	`"u"`
`PRIuFAST32`	Unsigned placeholder string for the fastest signed integer type with a width of at least 32.	`"lu"`
`PRIuFAST64`	Unsigned placeholder string for the fastest signed integer type with a width of at least 64, where supported.	`"llu"`
`PRIxFAST8`/`PRIXFAST8`	Hexadecimal placeholder string for the fastest signed integer type with a width of at least 8.	`"x"`/`"X"`
`PRIxFAST16`/`PRIXFAST16`	Hexadecimal placeholder string for the fastest signed integer type with a width of at least 16.	`"x"`/`"X"`
`PRIxFAST32`/`PRIXFAST32`	Hexadecimal placeholder string for the fastest signed integer type with a width of at least 32.	`"lx"`/`"lX"`
`PRIxFAST64`/`PRIXFAST64`	Octal placeholder string for the fastest signed integer type with a width of at least 64, where supported.	`"llx"`/`"llX"`
`PRIoLEAST8`	Octal placeholder string for a signed integer type with a width of at least 8 bits.	`"o"`

...........continued		
Macro Name	**Description**	**Value**
`PRIoLEAST16`	Octal placeholder string for a signed integer type with a width of at least 16 bits.	`"o"`
`PRIoLEAST32`	Octal placeholder string for a signed integer type with a width of at least 32 bits.	`"lo"`
`PRIoLEAST64`	Octal placeholder string for a signed integer type with a width of at least 64 bits, where supported.	`"llo"`
`PRIuLEAST8`	Unsigned placeholder string for a signed integer type with a width of at least 8 bits.	`"u"`
`PRIuLEAST16`	Unsigned placeholder string for a signed integer type with a width of at least 16 bits.	`"u"`
`PRIuLEAST32`	Unsigned placeholder string for a signed integer type with a width of at least 32 bits.	`"lu"`
`PRIuLEAST64`	Unsigned placeholder string for a signed integer type with a width of at least 64 bits, where supported.	`"llu"`
`PRIoMAX`	Octal placeholder string for a signed integer type with maximum width.	`"llo"`where 64-bit integers are supported; `"lo"` otherwise
`PRIuMAX`	Unsigned placeholder string for a signed integer type with maximum width.	`"llu"`where 64-bit integers are supported; `"lu"` otherwise
`PRIxMAX/PRIXMAX`	Hexadecimal placeholder string for a signed integer type with maximum width.	`"llx"/"llX"` where 64-bit integers are supported; `"lx"/"lX"` otherwise
`PRIoPTR`	Octal placeholder string for `uintptr_t`.	`"lo"`
`PRIuPTR`	Unsigned placeholder string for `uintptr_t`.	`"lu"`
`PRIxPTR/PRIXPTR`	Hexadecimal placeholder string for `uintptr_t`.	`"lx"/"lX"`

7.8.3 Scan Format Macros for Signed Integers

The macros in following table expand to character string literals and can be used with the `scanf` family of functions when reading in signed integer values.

Macro Name	Description	Value
`SCNd8`	Decimal placeholder string for a signed 8-bit integer type.	`"hhd"`
`SCNd16`	Decimal placeholder string for a signed 16-bit integer type.	`"hd"`

...........continued

Macro Name	Description	Value
SCNd32	Decimal placeholder string for a signed 32-bit integer type.	"ld"
SCNd64	Decimal placeholder string for a signed 64-bit integer type, where supported.	"lld"
SCNi8	Integer placeholder string for a signed 8-bit integer type.	"hhi"
SCNi16	Integer placeholder string for a signed 16-bit integer type.	"hi"
SCNi32	Integer placeholder string for a signed 32-bit integer type.	"li"
SCNi64	Integer placeholder string for a signed 64-bit integer type, where supported.	"lli"
SCNdFAST8	Decimal placeholder string for the fastest signed integer type with a width of at least 8.	"hhd"
SCNdFAST16	Decimal placeholder string for the fastest signed integer type with a width of at least 16.	"hd"
SCNdFAST32	Decimal placeholder string for the fastest signed integer type with a width of at least 32.	"ld"
SCNdFAST64	Decimal placeholder string for the fastest signed integer type with a width of at least 64, where supported.	"lld"
SCNiFAST8	Integer placeholder string for the fastest signed integer type with a width of at least 8 bits.	"hhi"
SCNiFAST16	Integer placeholder string for the fastest signed integer type with a width of at least 16.	"hi"
SCNiFAST32	Integer placeholder string for the fastest signed integer type with a width of at least 32.	"li"
SCNiFAST64	Integer placeholder string for the fastest signed integer type with a width of at least 64, where supported.	"lli"
SCNdLEAST8	Decimal placeholder string for a signed integer type with a width of at least 8 bits.	"hhd"
SCNdLEAST16	Decimal placeholder string for a signed integer type with a width of at least 16 bits.	"hd"

..........continued		
Macro Name	**Description**	**Value**
SCNdLEAST32	Decimal placeholder string for a signed integer type with a width of at least 32 bits.	`"ld"`
SCNdLEAST64	Decimal placeholder string for a signed integer type with a width of at least 64 bits, where supported.	`"lld"`
SCNiLEAST8	Integer placeholder string for a signed integer type with a width of at least 8 bits.	`"hhi"`
SCNiLEAST16	Integer placeholder string for a signed integer type with a width of at least 16 bits.	`"hi"`
SCNiLEAST32	Integer placeholder string for a signed integer type with a width of at least 32 bits.	`"li"`
SCNiLEAST64	Integer placeholder string for a signed integer type with a width of at least 64 bits, where supported.	`"lli"`
SCNdMAX	Decimal placeholder string for a signed integer type with maximum width.	`"lld"`where 64-bit integers are supported; `"ld"` otherwise
SCNiMAX	Integer placeholder string for a signed integer type with maximum width.	`"lli"`where 64-bit integers are supported; `"ld"` otherwise
SCNdPTR	Decimal placeholder string for `intptr_t`.	`"ld"`
SCNiPTR	Integer placeholder string for `intptr_t`.	`"li"`

7.8.4 Scan Format Macros for Unsigned Integers

The macros in following table expand to character string literals and can be used with the `scanf` family of functions when reading in unsigned integer values.

Macro Name	Description	Value
SCNo8	Octal placeholder string for an unsigned 8-bit integer type.	`"hho"`
SCNo16	Octal placeholder string for an unsigned 16-bit integer type.	`"ho"`
SCNo32	Octal placeholder string for an unsigned 32-bit integer type.	`"lo"`
SCNo64	Octal placeholder string for an unsigned 64-bit integer type, where supported.	`"llo"`
SCNu8	Unsigned placeholder string for an unsigned 8-bit integer type.	`"hhu"`

...........continued

Macro Name	Description	Value
SCNu16	Unsigned placeholder string for an unsigned 16-bit integer type.	"hu"
SCNu32	Unsigned placeholder string for an unsigned 32-bit integer type.	"lu"
SCNu64	Unsigned placeholder string for an unsigned 64-bit integer type, where supported.	"llu"
SCNx8	Hexadecimal placeholder string for an unsigned 8-bit integer type.	"hhx"
SCNx16	Hexadecimal placeholder string for an unsigned 16-bit integer type.	"hx"
SCNx32	Hexadecimal placeholder string for an unsigned 32-bit integer type.	"lx"
SCNx64	Hexadecimal placeholder string for an unsigned 64-bit integer type, where supported.	"llx"
SCNoFAST8	Octal placeholder string for the fastest signed integer type with a width of at least 8.	"hho"
SCNoFAST16	Octal placeholder string for the fastest signed integer type with a width of at least 16.	"ho"
SCNoFAST32	Octal placeholder string for the fastest signed integer type with a width of at least 32.	"lo"
SCNoFAST64	Octal placeholder string for the fastest signed integer type with a width of at least 64, where supported.	"llo"
SCNuFAST8	Unsigned placeholder string for the fastest signed integer type with a width of at least 8 bits.	"hhu"
SCNuFAST16	Unsigned placeholder string for the fastest signed integer type with a width of at least 16.	"hu"
SCNuFAST32	Unsigned placeholder string for the fastest signed integer type with a width of at least 32.	"lu"
SCNuFAST64	Unsigned placeholder string for the fastest signed integer type with a width of at least 64, where supported.	"llu"
SCNxFAST8	Hexadecimal placeholder string for the fastest signed integer type with a width of at least 8.	"hhx"

...........continued

Macro Name	Description	Value
SCNxFAST16	Hexadecimal placeholder string for the fastest signed integer type with a width of at least 16.	`"hx"`
SCNxFAST32	Hexadecimal placeholder string for the fastest signed integer type with a width of at least 32.	`"lx"`
SCNxFAST64	Octal placeholder string for the fastest signed integer type with a width of at least 64, where supported.	`"llx"`
SCNoLEAST8	Octal placeholder string for a signed integer type with a width of at least 8 bits.	`"hho"`
SCNoLEAST16	Octal placeholder string for a signed integer type with a width of at least 16 bits.	`"ho"`
SCNoLEAST32	Octal placeholder string for a signed integer type with a width of at least 32 bits.	`"lo"`
SCNoLEAST64	Octal placeholder string for a signed integer type with a width of at least 64 bits, where supported.	`"llo"`
SCNuLEAST8	Unsigned placeholder string for a signed integer type with a width of at least 8 bits.	`"hhu"`
SCNuLEAST16	Unsigned placeholder string for a signed integer type with a width of at least 16 bits.	`"hu"`
SCNuLEAST32	Unsigned placeholder string for a signed integer type with a width of at least 32 bits.	`"lu"`
SCNuLEAST64	Unsigned placeholder string for a signed integer type with a width of at least 64 bits, where supported.	`"llu"`
SCNoMAX	Octal placeholder string for a signed integer type with maximum width.	`"llo"`where 64-bit integers are supported; `"lo"` otherwise
SCNuMAX	Unsigned placeholder string for a signed integer type with maximum width.	`"llu"`where 64-bit integers are supported; `"lu"` otherwise
SCNxMAX/SCNXMAX	Hexadecimal placeholder string for a signed integer type with maximum width.	`"llx"`/`"llX"` where 64-bit integers are supported; `"lx"`/`"lX"` otherwise
SCNoPTR	Octal placeholder string for `uintptr_t`.	`"lo"`
SCNuPTR	Unsigned placeholder string for `uintptr_t`.	`"lu"`

...........continued		
Macro Name	**Description**	**Value**
SCNxPTR/SCNXPTR	Hexadecimal placeholder string for `uintptr_t`.	`"lx"/"lX"`

7.9 <iso646.h> Alternate Spellings

The `<iso646.h>` header file consists of macros that can be used to replace the logical and bitwise operators.

7.9.1 iso6464 Alternate Spelling Macros

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

7.10 <limits.h> Implementation-Defined Limits

The header file `limits.h` consists of macros that define the minimum and maximum values of integer types. Each of these macros can be used in `#if` preprocessing directives.

Table 7-1. 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`	127
CHAR_MIN	Minimum value of a `char`	-128
INT_MAX	Maximum value of a `int`	32767
INT_MIN	Minimum value of a `int`	-32768
LLONG_MAX	Maximum value of a `long long int`	9223372036854775807
LLONG_MIN	Minimum value of a `long long int`	-9223372036854775808
LONG_MAX	Maximum value of a `long int`	2147483647

...........continued		
Macro Name	**Description**	**Value**
LONG_MIN	Minimum value of a `long int`	-2147483648
MB_LEN_MAX	Maximum number of bytes in a multibyte character	1
SCHAR_MAX	Maximum value of a `signed char`	127
SCHAR_MIN	Minimum value of a `signed char`	-128
SHRT_MAX	Maximum value of a `short int`	32767
SHRT_MIN	Minimum value of a `short int`	-32768
UCHAR_MAX	Maximum value of an `unsigned char`	255
UINT_MAX	Maximum value of an `unsigned int`	65535
ULLONG_MAX	Maximum value of a `long long unsigned int`	18446744073709551615
ULONG_MAX	Maximum value of a `long unsigned int`	4294967295
USHRT_MAX	Maximum value of an `unsigned short int`	65535

7.11 <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` if the 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. Each function description includes a definition of the function’s response to such values.

7.11.1 HUGE_VAL

`HUGE_VAL` is returned by a function on a range error (e.g., the function tries to return a value too large to be represented in the target precision).

Include

`<math.h>`

Remarks

`HUGE_VAL` is returned if a function result is negative and is too large (in magnitude) to be represented in the target precision. When the printed result is +/- `HUGE_VAL`, it will be represented by +/- `inf`.

7.11.2 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 in radians in the range of 0 to pi (inclusive).

Remarks

A domain error occurs if x is less than -1 or greater than 1.

Example

```
#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
```

7.11.3 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 in radians in the range of 0 to pi (inclusive).

Remarks

A domain error occurs if x is less than -1 or greater than 1.

Example

```
#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
```

7.11.4 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 of -pi/2 to +pi/2 (inclusive).

Remarks

A domain error occurs if x is less than -1 or greater than 1.

Example

```
#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
```

7.11.5 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

Return Value

Returns the arc sine in radians in the range of -pi/2 to +pi/2 (inclusive).

Remarks

A domain error occurs if x is less than -1 or greater than 1.

Example

```
#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
```

7.11.6 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 radians in the range of -pi/2 to +pi/2 (inclusive).

Remarks

No domain or range error will occur.

Example

```
#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
```

7.11.7 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 radians in the range of -pi/2 to +pi/2 (inclusive).

Remarks

No domain or range error will occur.

Example

```
#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
```

7.11.8 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 of -pi to pi (inclusive) with the quadrant determined by the signs of both parameters.

Remarks

A domain error occurs if both `x` and `y` are zero or both `x` and `y` are +/- infinity.

Example

```
#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
```

7.11.9 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 of -pi to pi with the quadrant determined by the signs of both parameters.

Remarks

A domain error occurs if both `x` and `y` are zero or both `x` and `y` are +/- infinity.

Example

```
#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
```

7.11.10 cbrt Function

Calculates the 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 non-negative square root of `x`.

Example

```
#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
```

7.11.11 cbrtf Function

Calculates the 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 non-negative square root of `x`.

Example

```
#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
```

7.11.12 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

```
#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
```

7.11.13 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

```
#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
```

7.11.14 copysign Function

Returns the value of x but with the sign of y.

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

```
#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
```

7.11.15 copysignf Function

Returns the value of x but with the sign of y.

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

```
#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
```

7.11.16 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` in radians in the ranges of -1 to 1 inclusive.

Remarks

A domain error will occur if `x` is a NaN or infinity.

Example

```
#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
```

7.11.17 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` in radians in the ranges of -1 to 1 inclusive.

Remarks

A domain error will occur if `x` is a NaN or infinity.

Example

```
#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
```

7.11.18 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`.

Remarks

A range error will occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.19 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`.

Remarks

A range error will occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.20 exp Function

Calculates the exponential function of x (e raised to the power 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`. On an overflow, `exp` returns `inf` and on an underflow `exp` returns 0.

Remarks

A range error occurs if the magnitude of `x` is too large.

Example

```
#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
```

7.11.21 expf Function

Calculates the exponential function of x (e raised to the power 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`. On an overflow, `expf` returns `inf` and on an underflow `expf` returns 0.

Remarks

A range error occurs if the magnitude of `x` is too large.

Example

```
#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
```

7.11.22 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

```
#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
```

7.11.23 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

```
#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
```

7.11.24 fdim Function

Calculates the positive difference between the two arguments.

Include

<math.h>

Prototype

`double fdim(double x, double y);`

Argument

x any double precision floating-point number

y any double precision floating-point number

Return Value

Returns the positive difference between the two argument, that being `x - y` when `x` is larger than `y`, and 0 for all other values of `x`.

Remarks

A range error might occur.

Example

```
#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
```

7.11.25 fdmif Function

Calculates the positive difference between the two arguments.

Include

`<math.h>`

Prototype

`float fdimf(float x, float y);`

Argument

x	any single precision floating-point number
y	any single precision floating-point number

Return Value

Returns the positive difference between the two argument, that being `x - y` when `x` is larger than `y`, and 0 for all other values of `x`.

Remarks

A range error might occur.

Example

```
#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
```

7.11.26 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

```
#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
```

7.11.27 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

```
#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
```

7.11.28 fma Function

Returns the value of `x * y + z`.

Include

<math.h>

Prototype

`double fma(double x, double y, double z);`

Argument

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>  /* for fma          */
#include <stdio.h> /* for printf */

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
```

7.11.29 fmaf Function

Returns the value of `x * y + z`.

Include

`<math.h>`

Prototype

`float fmaf(float x, float y, float z);`

Argument

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>  /* for fmaf          */
#include <stdio.h> /* for printf */

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
```

7.11.30 fmax Function

Returns the value of the larger argument.

Include

`<math.h>`

Prototype

`double fmax(double x, double y);`

Argument

x any double precision floating-point number

y any double precision floating-point number

Return Value

Returns the value of the larger argument.

Example

```
#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
```

7.11.31 fmaxf Function

Returns the value of the smaller argument.

Include

`<math.h>`

Prototype

`float fmaxf(float x, float y);`

Argument

x any single precision floating-point number

y any single precision floating-point number

Return Value

Returns the value of the larger argument.

Example

```
#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
```

7.11.32 fmin Function

Returns the value of the smaller argument.

Include

`<math.h>`

Prototype

`double fmin(double x, double y);`

Argument

x	any double precision floating-point number
y	any double precision floating-point number

Return Value

Returns the value of the smaller argument.

Example

```
#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 larger of -5.700000 and 2.000000 is -5.700000
```

7.11.33 fminf Function

Returns the value of the smaller 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 value of the smaller argument.

Example

```
#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 larger of -5.700000 and 2.000000 is -5.700000
```

7.11.34 fmod Function

Calculates the remainder of x/y as a double precision value.

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` divided by `y`.

Remarks

If `y` = 0, a domain error occurs. If `y` is non-zero, the result will have the same sign as `x` and the magnitude of the result will be less than the magnitude of `y`.

Example

```
#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
```

7.11.35 fmodf Function

Calculates the remainder of x/y as a single precision value.

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 divided by y.

Remarks

If y = 0, a domain error occurs. If y is non-zero, the result will have the same sign as x and the magnitude of the result will be less than the magnitude of y.

Example

```
#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
```

7.11.36 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

```
#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
```

7.11.37 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

```
#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
```

7.11.38 hypot Function

Calculates the square root of a 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`.

Remarks

A range error might occur.

Example

```
#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
```

7.11.39 hypotf Function

Calculates the square root of a sum of squared single precision floating-point values.

Include

`<math.h>`

Prototype

`float hypot(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`.

Remarks

A range error might occur.

Example

```
#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
```

7.11.40 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`. A range error may occur if `x` is 0.

Remarks

A range error might occur if `x` is 0.

Example

```
#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
```

7.11.41 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`. A range error may occur if `x` is 0.

Remarks

A range error might occur if `x` is 0.

Example

```
#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
```

7.11.42 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 its argument is finite, i.e. is not infinite or NaN.

Example

```
#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
```

7.11.43 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 its argument is either positive or negative infinity; zero otherwise.

Example

```
#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
```

7.11.44 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 its argument is NaN (not a number); false otherwise.

Example

```
#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
```

7.11.45 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 ex);`

Arguments

x	floating-point value
ex	integer exponent

Return Value

Returns x * 2^`ex`. On an overflow, `ldexp` returns `inf` and on an underflow, `ldexp` returns 0.

Remarks

A range error will occur on overflow or underflow.

Example

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

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

  errno = 0;
  x = -0.625;
  n = 2;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
```

```
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexp(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 2.5;
  n = 3;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  printf("For a number = %f and an exponent = %d\n",
         x, n);
  printf("  ldexp(%f, %d) = %f\n\n",
         x, n, y);

  errno = 0;
  x = 15.0;
  n = 10000;
  y = ldexp (x, n);
  if (errno)
    perror("Error");
  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

Error: range error
For a number = 15.000000 and an exponent = 10000
  ldexp(15.000000, 10000) = inf
```

7.11.46 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 ex);`

Arguments

x	floating-point value
ex	integer exponent

Return Value

Returns x * 2^`ex`. On an overflow, `ldexp` returns `inf` and on an underflow, `ldexpf` returns 0.

Remarks

A range error will occur on overflow or underflow.

Example

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

int main(void)
{
```

```
    float x,y;
    int n;

    errno = 0;
    x = -0.625F;
    n = 2;
    y = ldexpf (x, n);
    if (errno)
      perror("Error");
    printf("For a number = %f and an exponent = %d\n", x, n);
    printf("  ldexpf(%f, %d) = %f\n\n", x, n, y);

    errno = 0;
    x = 2.5F;
    n = 3;
    y = ldexpf (x, n);
    if (errno)
      perror("Error");
    printf("For a number = %f and an exponent = %d\n", x, n);
    printf("  ldexpf(%f, %d) = %f\n\n", x, n, y);

    errno = 0;
    x = 15.0F;
    n = 10000;
    y = ldexpf (x, n);
    if (errno)
      perror("Error");
    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

Error: range error
For a number = 15.000000 and an exponent = 10000
  ldexpf(15.000000, 10000) = inf
```

7.11.47 llrint Function

Returns the 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 integer is returned as an integer value.

Remarks

The value returned is unspecified if the rounded value is outside the range of the return type. A range error may occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.48 llrintf Function

Returns the 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 integer is returned as an integer value.

Remarks

The value returned is unspecified if the rounded value is outside the range of the return type. A range error may occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.49 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 logarithm of `x`. `-inf` is returned if `x` is 0 and NaN is returned if `x` is a negative number.

Remarks

A domain error occurs if `x` ≤ 0.

Example

```
#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
```

7.11.50 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 logarithm of `x`. `-inf` is returned if `x` is 0 and NaN is returned if `x` is a negative number.

Remarks

A domain error occurs if `x` ≤ 0.

Example Output

```
#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
```

7.11.51 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`. `-inf` is returned if `x` is 0 and NaN is returned if `x` is a negative number.

Remarks

A domain error occurs if `x` ≤ 0.

Example

```
#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
```

7.11.52 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 double precision floating-point positive number

Return Value

Returns the base-10 logarithm of `x`. `-inf` is returned if `x` is 0 and NaN is returned if `x` is a negative number.

Remarks

A domain error occurs if `x` ≤ 0.

Example

```
#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
```

7.11.53 lrint Function

Returns the 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 an integer value.

Remarks

The value returned is unspecified if the rounded value is outside the range of the return type. A range error may occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.54 lrintf Function

Returns the 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 an integer value.

Remarks

The value returned is unspecified if the rounded value is outside the range of the return type. A range error may occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.55 lround Function

Returns the 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 an integer value.

Remarks

The value returned is unspecified should the rounded value fall outside the range of the return type. A range error might occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.56 lroundf Function

Returns the 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 an integer value.

Remarks

The value returned is unspecified should the rounded value fall outside the range of the return type. A range error might occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.57 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 a stored the integer part

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

```
#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
```

7.11.58 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 a stored the integer part

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

```
#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
```

7.11.59 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 `y` is 0, `pow` returns 1. If `x` is 0.0 and `y` is less than 0, `pow` returns inf and a domain error occurs. If the result overflows or underflows, a range error occurs.

Example

```
#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
```

7.11.60 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 `y` is 0, `pow` returns 1. If `x` is 0.0 and `y` is less than 0, `pow` returns inf and a domain error occurs. If the result overflows or underflows, a range error occurs.

Example

```
#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
```

7.11.61 round Function

Returns the 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 floating-point value.

Example

```
#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
```

7.11.62 roundf Function

Returns the 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 floating-point value.

Example

```
#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
```

7.11.63 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

```
#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
```

7.11.64 sin Function

Calculates the trigonometric sine function of a double precision floating-point value.

Include

<math.h>

Prototype

`double sin (double x);`

Argument

x	value for which to return the sine

Return Value

Returns the sine of `x` in radians in the ranges of -1 to 1 inclusive.

Remarks

A domain error will occur if `x` is a NaN or infinity.

Example

```
#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
```

7.11.65 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` in radians in the ranges of -1 to 1 inclusive.

Remarks

A domain error will occur if `x` is a NaN or infinity.

Example

```
#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
```

7.11.66 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

A range error will occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.67 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

A range error will occur if the magnitude of `x` is too large.

Example

```
#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
```

7.11.68 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 negative, a domain error occurs.

Example

```
#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
```

7.11.69 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 negative, a domain error occurs.

Example

```
#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
```

7.11.70 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 in radians.

Remarks

A domain error will occur if x is a NaN or infinity.

Example

```
#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
```

7.11.71 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` in radians.

Remarks

A domain error will occur if `x` is a NaN or infinity.

Example

```
#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
```

7.11.72 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

```
#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
```

7.11.73 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

```
#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
```

7.11.74 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 floating-point value.

Example

```
#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
```

7.11.75 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 floating-point value.

Example

```
#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
```

7.12 <pgmspace.h>

The `<pgmspace.h>` header declares macros that relate to reading of program memory. Most of these features are included largely for legacy code, as program memory can be read more easily using the const-data-in-progmem feature.

In addition to the functions listed here, this header also provides program memory variants of functions that are provided by `<string.h>`. These functions suffix `_P` to the standard string function name, for example, `strcpy_P`. These functions read from program memory, rather than from data memory. Again, these functions are provided for legacy projects, as the const-data-in-progmem feature allows for reading of program memory using the standard function provided by `<string.h>`.

7.12.1 pgm_get_far_address Macro

Obtain a far address from an object.

Include

`<avr/pgmspace.h>`

Prototype

```
uint_farptr_t pgm_get_far_address(object);
```

Remarks

Obtains the far (32-bit) address of *`object`*.

Example

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romObj = 0x55;
int main(void)
{
  uint_farptr_t * cp;
  cp = pgm_get_far_address(&romObj);
}
```

7.12.2 pgm_read_byte Macro

Read a byte from the program space with a near address.

Include

`<avr/pgmspace.h>`

Prototype

```
unsigned char pgm_read_byte(unsigned int);
```

Remarks

Read a byte from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romObj = 0x55;
int main(void)
{
  unsigned char val;
  val = pgm_read_byte(&romObj);
}
```

7.12.3 pgm_read_byte_far Macro

Read a byte from the program space with a far address.

Include

`<avr/pgmspace.h>`

Prototype

```
unsigned char pgm_read_byte_far(unsigned long int);
```

Remarks

Read a byte from the program space with a 32-bit (far) address. These functions use the `elpm` instruction, and so can access any address in program memory

Example

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romObj = 0x55;
int main(void)
{
  unsigned char val;
  val = pgm_read_byte_far(&romObj);
}
```

7.12.4 pgm_read_byte_near Macro

Read a byte from the program space with a near address.

Include

`<avr/pgmspace.h>`

Prototype

```
unsigned char pgm_read_byte_near(unsigned int);
```

Remarks

Read a byte from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romObj = 0x55;
int main(void)
{
  unsigned char val;
  val = pgm_read_byte_near(&romObj);
}
```

7.12.5 pgm_read_dword_near Macro

Read a double-width word from the program space with a near address.

Include

`<avr/pgmspace.h>`

Prototype

```
unsigned long int pgm_read_dword_near(unsigned int);
```

Remarks

Read a 32-bit word from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
const unsigned long PROGMEM romObj = 0x55;
int main(void)
{
  unsigned long val;
  val = pgm_read_dword_near(&romObj);
}
```

7.12.6 pgm_read_dword_far Macro

Read a double-width word from the program space with a far address.

Include

`<avr/pgmspace.h>`

Prototype

```
unsigned long int pgm_read_byte_far(unsigned long int);
```

Remarks

Read a 32-bit word from the program space with a 32-bit (far) address. These functions use the `elpm` instruction, and so can access any address in program memory

Example

```
#include <avr/pgmspace.h>
const unsigned long int PROGMEM romObj = 0x55;
int main(void)
{
  unsigned long int val;
  val = pgm_read_dword_far(&romObj);
}
```

7.12.7 pgm_read_dword_near Macro

Read a double-width word from the program space with a near address.

Include

<avr/pgmspace.h>

Prototype

```
unsigned long int pgm_read_dword_near(unsigned int);
```

Remarks

Read a 32-bit word from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
const unsigned long PROGMEM romObj = 0x55;
int main(void)
{
  unsigned long val;
  val = pgm_read_dword_near(&romObj);
}
```

7.12.8 pgm_read_float Macro

Read a floating-point object from the program space with a near address.

Include

<avr/pgmspace.h>

Prototype

```
float pgm_read_word(unsigned int);
```

Remarks

Read a `float` object from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
const float PROGMEM romObj = 1.23;
int main(void)
{
  float val;
  val = pgm_read_float(&romObj);
}
```

7.12.9 pgm_read_float_far Macro

Read a floating-point object from the program space with a far address.

Include

<avr/pgmspace.h>

Prototype

```
float pgm_read_byte_far(unsigned long int);
```

Remarks

Read a `float` object from the program space with a 32-bit (far) address. These functions use the `elpm` instruction, and so can access any address in program memory

Example

```
#include <avr/pgmspace.h>
const float PROGMEM romObj = 1.23;
int main(void)
{
  float val;
  val = pgm_read_float_far(&romObj);
}
```

7.12.10 pgm_read_float_near Macro

Read a floating-point object from the program space with a near address.

Include

`<avr/pgmspace.h>`

Prototype

```
float pgm_read_word_near(unsigned int);
```

Remarks

Read a `float` object from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
const float PROGMEM romObj = 1.23;
int main(void)
{
  float val;
  val = pgm_read_float_near(&romObj);
}
```

7.12.11 pgm_read_ptr Macro

Read a pointer from the program space with a near address.

Include

`<avr/pgmspace.h>`

Prototype

```
void * pgm_read_ptr(unsigned int);
```

Remarks

Read a 16-bit generic pointer from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
unsigned int input;
const unsigned int PROGMEM * romPtr = &input;
int main(void)
{
  unsigned int * val;
  val = (unsigned int *)pgm_read_ptr(&romPtr);
}
```

7.12.12 pgm_read_ptr_far Macro

Read a pointer from the program space with a far address.

Include

`<avr/pgmspace.h>`

Prototype

```
void * pgm_read_byte_far(unsigned long int);
```

Remarks

Read a 16-bit generic pointer from the program space with a 32-bit (far) address. These functions use the `elpm` instruction, and so can access any address in program memory

Example

```
#include <avr/pgmspace.h>
unsigned int input;
const unsigned int PROGMEM * romPtr = &input;
int main(void)
{
  unsigned int * val;
  val = pgm_read_ptr_far(&romPtr);
}
```

7.12.13 pgm_read_ptr_near Macro

Read a pointer from the program space with a near address.

Include

`<avr/pgmspace.h>`

Prototype

```
void * pgm_read_ptr_near(unsigned int);
```

Remarks

Read a 16-bit generic pointer from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
unsigned int input;
const unsigned int PROGMEM * romPtr = &input;
int main(void)
{
  unsigned int * val;
  val = (unsigned int *)pgm_read_ptr_near(&romPtr);
}
```

7.12.14 pgm_read_word Macro

Read a word from the program space with a near address.

Include

`<avr/pgmspace.h>`

Prototype

```
unsigned int pgm_read_word(unsigned int);
```

Remarks

Read a 16-bit word from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
const unsigned int PROGMEM romObj = 0x55;
int main(void)
{
  unsigned int val;
```

```
  val = pgm_read_word(&romObj);
}
```

7.12.15 pgm_read_word_far Macro

Read a word from the program space with a far address.

Include

`<avr/pgmspace.h>`

Prototype

```
unsigned int pgm_read_byte_far(unsigned long int);
```

Remarks

Read a 16-bit word from the program space with a 32-bit (far) address. These functions use the `elpm` instruction, and so can access any address in program memory

Example

```
#include <avr/pgmspace.h>
const unsigned int PROGMEM romObj = 0x55;
int main(void)
{
  unsigned int val;
  val = pgm_read_word_far(&romObj);
}
```

7.12.16 pgm_read_word_near Macro

Read a word from the program space with a near address.

Include

`<avr/pgmspace.h>`

Prototype

```
unsigned int pgm_read_word_near(unsigned int);
```

Remarks

Read a 16-bit word from the program space with a 16-bit (near) address.

Example

```
#include <avr/pgmspace.h>
const unsigned int PROGMEM romObj = 0x55;
int main(void)
{
  unsigned int val;
  val = pgm_read_word_near(&romObj);
}
```

7.12.17 PSTR Macro

Obtain a pointer to a string in program space.

Include

`<avr/pgmspace.h>`

Prototype

```
const PROGMEM char * PSTR(string);
```

Remarks

Obtain a pointer to a string in program space.

Example

```
#include <avr/pgmspace.h>
int main(void)
{
 const PROGMEM char * cp;
  cp = PSTR("hello");
}
```

7.13 <sfr_defs.h>

The header file `sfr_defs.h` consists of functions that assist with accessing special function registers.

7.13.1 _BV macro

A macro that allows bit operations.

Include

`<avr/sfr_defs.h>`

Prototype

```
void _BV(bit_number);
```

Remarks

This macro converts a bit number into a byte value, thus allowing you to access bits within an address.

The compiler will use a hardware `sbi` or `cbi` instruction to perform the access if appropriate, or a read or write operation otherwise.

Example

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  PORTB |= _BV(PB1);  // set bit #1 of PORTB
  EECR &= ~(_BV(EEPM4) | _BV(EEPM5));  // clear bits #4 and #5 in EECR
}
```

7.13.2 Bit_is_clear Macro

A macro that returns true if a bit is clear in an SFR.

Include

`<avr/sfr_defs.h>`

Prototype

```
int _bit_is_clear(sfr, bit_number);
```

Remarks

This macro tests to see if the specified bit in *sfr* is clear, returning true if that is the case; 0 otherwise.

Example

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  if(bit_is_clear(EIMSK, PCIE2))
    proceed = 1;
}
```

7.13.3 bit_is_set Macro

A macro that returns true if a bit is set in an SFR.

Include

`<avr/sfr_defs.h>`

Prototype

```
int _bit_is_set(sfr, bit_number);
```

Remarks

This macro tests to see if the specified bit in *sfr* is set, returning true if that is the case; 0 otherwise.

Example

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  if(bit_is_set(EIMSK, PCIE2))
    proceed = 0;
}
```

7.13.4 loop_until_bit_is_clear Macro

A macro that waits until a bit becomes clear in an SFR.

Include

`<avr/sfr_defs.h>`

Prototype

```
int loop_until_bit_is_clear(sfr, bit_number);
```

Remarks

This macro loops until the specified bit in *sfr* becomes clear.

Example

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  loop_until_bit_is_clear(PINA, PINA3);
  process();
}
```

7.13.5 loop_until_bit_is_set Macro

A macro that waits until a bit becomes set in an SFR.

Include

`<avr/sfr_defs.h>`

Prototype

```
int loop_until_bit_is_set(sfr, bit_number);
```

Remarks

This macro loops until the specified bit in *sfr* becomes set.

Example

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  startConversion();
  loop_until_bit_is_set(ADCSRA, ADIF);
}
```

7.14 <sleep.h>

The 8-bit AVR devices can be put into different sleep modes. Refer to your device data sheet for details of these modes.

7.14.1 set_sleep_mode Macro

Specify how the device is to behave in sleep mode.

Include

`<avr/sleep.h>`

Prototype

```
void set_sleep_mode(mode);
```

Remarks

This macro sets the appropriate bits in the device registers to specify how the device will behavein sleep mode. The mode argument can be the appropriate macros defined once you include `<avr/sleep.h>`, such as `SLEEP_MODE_IDLE`, `SLEEP_MODE_PWR_DOWN`, `SLEEP_MODE_PWR_SAVE`, `SLEEP_MODE_STANDBY`, etc.

Example

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  set_sleep_mode(SLEEP_MODE_IDLE);
  cli();
  if (some_condition)
  {
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
}
```

7.14.2 sleep_bod_disable Macro

Disable brown out detection prior to sleep.

Include

`<avr/sleep.h>`

Prototype

```
void sleep_bod_disable(void);
```

Remarks

This macro can be used to disable brown out detection prior to putting the device to sleep. This macro generates inlined assembly code that will correctly implement the timed sequence for disabling the brown out detector (BOD).

However, there is a limited number of cycles after the BOD has been disabled that the device can be put into sleep mode, otherwise the BOD will not truly be disabled.

Not all devices have this feature.

Example

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  initSystem();
  sleep_enable();
  sleep_bod_disable();
  sei();
  sleep_cpu();
  sleep_disable();
}
```

7.14.3 sleep_cpu Macro

Put the device to sleep.

Include

<avr/sleep.h>

Prototype

```
void sleep_cpu(void);
```

Remarks

This macro puts the device to sleep. The sleep enable bit must be set beforehand for the device to sleep.

Example

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  cli();
  if (some_condition)
  {
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
}
```

7.14.4 sleep_disable Macro

Bring the device out of sleep mode.

Include

<avr/sleep.h>

Prototype

```
void sleep_disable(void);
```

Remarks

This macro clears the sleep enable bit, preventing the device from entering sleep mode.

Example

```
#include <xc.h>
#include <avr/sleep.h>
```

```
int main(void)
{
  initSystem();
  sleep_enable();
  sei();
  sleep_cpu();
  sleep_disable();
}
```

7.14.5 sleep_enable Macro

Put the device into sleep mode.

Include

`<avr/sleep.h>`

Prototype

```
void sleep_enable(void);
```

Remarks

This macro sets the sleep enable bits, putting the device into sleep mode so it may be placed into sleep when required using `sleep_cpu()`.

How the device is brought out of sleep mode depends on the specific mode selected with the `set_sleep_mode()` function.

Example

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  cli();
  if (some_condition)
  {
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
}
```

7.14.6 sleep_mode Macro

Enable sleep mode and put the device to sleep.

Include

`<avr/sleep.h>`

Prototype

```
void sleep_mode(void);
```

Remarks

This macro clears the sleep enable bit, puts the device to sleep, and disables the sleep enable bit afterwards. As this macro might cause race conditions in some situations, you might prefer to use the macros which perform these individual steps, ensuring that interrupts are enabled immediately prior to the device being put to sleep.

Example

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  initSystem();
```

```
  sleep_mode();
}
```

7.15 <stdarg.h> Variable Argument Lists

The header file `stdarg.h` supports functions with variable argument lists. This allows functions to have arguments without corresponding parameter declarations. There must be at least one named argument. The variable arguments are represented by ellipses (...). An object of type `va_list` must be declared inside the function to hold the arguments. `va_start` will initialize the variable to an argument list, `va_arg` will access the argument list and `va_end` will end the use of the argument.

7.15.1 va_list Type

The type `va_list` declares a variable that will refer to each argument in a variable-length argument list.

Include

`<stdarg.h>`

Example

See `va_arg`.

7.15.2 va_arg Macro

Gets the current argument.

Include

`<stdarg.h>`

Prototype

`#define va_arg(va_list ap, Ty)`

Arguments

ap	pointer to list of arguments
Ty	type of argument to be retrieved

Return Value

Returns the current argument

Remarks

`va_start` must be called before `va_arg`.

Example

```
#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
```

7.15.3 va_end Macro

Ends the use of `ap`.

Include

`<stdarg.h>`

Prototype

`#define va_end(va_list ap)`

Argument

ap	pointer to list of arguments

Remarks

After a call to `va_end`, the argument list pointer, `ap`, is considered to be invalid. Further calls to `va_arg` should not be made until the next `va_start`. In the 16-bit compiler, `va_end` does nothing, so this call is not necessary but should be used for readability and portability.

Example

See `va_arg`.

7.15.4 va_start Macro

Sets the argument pointer `ap` to first optional argument in the variable-length argument list.

Include

`<stdarg.h>`

Prototype

`#define va_start(va_list ap, last_arg)`

Arguments

ap	pointer to list of arguments
last_arg	last named parameter before the optional arguments

Example

See `va_arg`.

7.16 <stdbool.h> Boolean Types and Values

Enter a short description of your concept here (optional).

The header file `stdbool.h` consists of types and macros that are usable when working with boolean types.

7.16.1 stdbool.h Types and Values

bool
Alternate type name to `_Bool`.

Include

`<stdbool.h>`

Remarks

The `bool` macro allows the use of an alternate type name to `_Bool`.

true
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
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
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.

7.17 <stddef.h> Common Definitions

The header file `stddef.h` consists of several types and macros that are of general use in programs.

7.17.1 stddef.h Types and Macros

ptrdiff_t
Type used to represent the difference in two pointer values.

Include

`<stdbool.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 used to represent the result of the `sizeof` operator.

Include

`<stdbool.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 used to represent the values of the largest extended character set.

Include

`<stdbool.h>`

Remarks

The `wchar_t` type is an integer type that is used to represent the values of the largest extended character set.

7.17.2 offsetof Macro

Gives the offset of a structure member from the beginning of the structure.

Include

`<stddef.h>`

Prototype

```
#define offsetof(T, mbr)
```

Arguments

T	name of structure
mbr	name of member in structure T

Return Value

Returns the offset in bytes of the specified member (`mbr`) from the beginning of the structure.

Remarks

The macro `offsetof` is undefined for bit-fields. An error message will occur if bit-fields are used.

Example

```
#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. 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.

7.18 <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.

7.18.1 Fastest Minimum-Width Integer Types

Type	Description	Definition
`int_fast8_t`	The fastest signed integer of at least 8 bits width.	`signed char`
`int_fast16_t`	The fastest signed integer of at least 16 bits width.	`short`
`int_fast24_t`	The fastest signed integer of at least 24 bits width.	`__int24`
`int_fast32_t`	The fastest signed integer of at least 32 bits width.	`long`
`int_fast64_t`	The fastest signed integer of at least 64 bits width, where supported.	`long long`
`uint_fast8_t`	The fastest unsigned integer of at least 8 bits width.	`unsigned char`
`uint_fast16_t`	The fastest unsigned integer of at least 16 bits width.	`unsigned short`
`uint_fast24_t`	The fastest unsigned integer of at least 24 bits width.	`__uint24`

...........continued

Type	Description	Definition
uint_fast32_t	The fastest unsigned integer of at least 32 bits width.	`unsigned long`
uint_fast64_t	The fastest unsigned integer of at least 64 bits width, where supported.	`unsigned long long`

7.18.2 Fixed Width Integer Types

Type	Description	Definition
int8_t	Signed integer of exactly 8 bits width.	`signed char`
int16_t	Signed integer of exactly 16 bits width.	`short`
int24_t	Signed integer of exactly 24 bits width.	`__int24`
int32_t	Signed integer of exactly 32 bits width.	`long`
int64_t	Signed integer of exactly 64 bits width, where supported.	`long long`
uint8_t	Unsigned integer of exactly 8 bits width.	`unsigned char`
uint16_t	Unsigned integer of exactly 16 bits width.	`unsigned short`
uint24_t	Unsigned integer of exactly 24 bits width.	`__uint24`
uint32_t	Unsigned integer of exactly 32 bits width.	`unsigned long`
uint64_t	Unsigned integer of exactly 64 bits width, where supported.	`unsigned long long`

7.18.3 Greatest Width Integer Types

Type	Description	Definition
intmax_t	Signed integer type large enough to hold any signed integer type.	64-bit `long long` where supported; 32-bit `long long` otherwise.
uintmax_t	Unsigned integer type large enough to hold any unsigned integer type.	64-bit `unsigned long long` where supported; 32-bit `unsigned long long` otherwise.

7.18.4 Integer Types For Pointer Objects

Type	Description	Definition
intptr_t	Signed integer type capable of holding a pointer to `void`, such that conversion back to a pointer to `void` will produce the original pointer value.	`int`

...........continued		
Type	**Description**	**Definition**
`uintptr_t`	Unsigned integer type capable of holding a pointer to `void`, such that conversion back to a pointer to `void` will produce the original pointer value.	`unsigned int`

7.18.5 Limits for Greatest Width Integer Types

Type	Description	Definition
`INTMAX_MIN`	Minimum value of largest width signed integer type.	-9223372036854775808 where 64-bit `long long` supported; -2147483648 otherwise.
`INTMAX_MAX_t`	Maximum value of largest width signed integer type.	9223372036854775807 where 64-bit `long long` supported; 2147483647 otherwise.
`UINTMAX_MAX_t`	Maximum value of largest width unsigned integer type.	18446744073709551615 where 64-bit `long long` supported; 4294967295 otherwise.

7.18.6 Limits of Fastest Minimum-Width Integer Types

Type	Description	Definition
`INT_FAST8_MIN`	Minimum value of fastest signed integer with at least 8 bits width.	-128
`INT_FAST8_MAX`	Maximum value of fastest signed integer with at least 8 bits width.	127
`UINT_FAST8_MAX`	Maximum value of fastest unsigned integer with at least 8 bits width.	255
`INT_FAST16_MIN`	Minimum value of fastest signed integer with at least 16 bits width.	-32768
`INT_FAST16_MAX`	Maximum value of fastest signed integer with at least 16 bits width.	32767
`UINT_FAST16_MAX`	Maximum value of fastest unsigned integer with at least 16 bits width.	65535
`INT_FAST24_MIN`	Minimum value of fastest signed integer with at least 24 bits width.	-8388608
`INT_FAST24_MAX`	Maximum value of fastest signed integer with at least 24 bits width.	8388609
`UINT_FAST24_MAX`	Maximum value of fastest unsigned integer with at least 24 bits width.	16777215
`INT_FAST32_MIN`	Minimum value of fastest signed integer with at least 32 bits width.	-2147483648
`INT_FAST32_MAX`	Maximum value of fastest signed integer with at least 32 bits width.	2147483647

...........continued		
Type	**Description**	**Definition**
`UINT_FAST32_MAX`	Maximum value of fastest unsigned integer with at least 32 bits width.	4294967295
`INT_FAST64_MIN`	Minimum value of fastest signed integer with at least 64 bits width.	-9223372036854775808
`INT_FAST64_MAX`	Maximum value of fastest signed integer with at least 64 bits width.	9223372036854775807
`UINT_FAST64_MAX`	Maximum value of fastest unsigned integer with at least 64 bits width.	18446744073709551615

7.18.7 Limits of Fixed-Width Integer Types

Type	Description	Definition
`INT8_MIN`	Minimum value of signed integer with 8 bits width.	-128
`INT8_MAX`	Maximum value of signed integer with 8 bits width.	127
`UINT8_MAX`	Maximum value of unsigned integer with 8 bits width.	255
`INT16_MIN`	Minimum value of signed integer with 16 bits width.	-32768
`INT16_MAX`	Maximum value of signed integer with 16 bits width.	32767
`UINT16_MAX`	Maximum value of unsigned integer with 16 bits width.	65535
`INT24_MIN`	Minimum value of signed integer with 24 bits width.	-8388608
`INT24_MAX`	Maximum value of signed integer with 24 bits width.	8388609
`UINT24_MAX`	Maximum value of unsigned integer with 24 bits width.	16777215
`INT32_MIN`	Minimum value of signed integer with 32 bits width.	-2147483648
`INT32_MAX`	Maximum value of signed integer with 32 bits width.	2147483647
`UINT32_MAX`	Maximum value of unsigned integer with 32 bits width.	4294967295
`INT64_MIN`	Minimum value of signed integer with 64 bits width.	-9223372036854775808
`INT64_MAX`	Maximum value of signed integer with 64 bits width.	9223372036854775807
`UINT64_MAX`	Maximum value of unsigned integer with 64 bits width.	18446744073709551615

7.18.8 Limits of Integer Types for Pointer Objects

Type	Description	Definition
`INTPTR_MIN`	The minimum value of a signed integer type capable of holding a pointer.	-32768
`INTPTR_MAX`	The maximum value of a signed integer type capable of holding a pointer.	32767
`UINTPTR_MAX`	The maximum value of an unsigned integer type capable of holding a pointer.	65535

7.18.9 Limits of Minimum-Width Integer Types

Type	Description	Definition
`INT_LEAST8_MIN`	Minimum value of signed integer with at least at least 8 bits width.	-128
`INT_LEAST8_MAX`	Maximum value of signed integer with at least 8 bits width.	127
`UINT_LEAST8_MAX`	Maximum value of unsigned integer with at least 8 bits width.	255
`INT_LEAST16_MIN`	Minimum value of signed integer with at least 16 bits width.	-32768
`INT_LEAST16_MAX`	Maximum value of signed integer with at least 16 bits width.	32767
`UINT_LEAST16_MAX`	Maximum value of unsigned integer with at least 16 bits width.	65535
`INT_LEAST24_MIN`	Minimum value of signed integer with at least 24 bits width.	-8388608
`INT_LEAST24_MAX`	Maximum value of signed integer with at least 24 bits width.	8388609
`UINT_LEAST24_MAX`	Maximum value of unsigned integer with at least 24 bits width.	16777215
`INT_LEAST32_MIN`	Minimum value of signed integer with at least 32 bits width.	-2147483648
`INT_LEAST32_MAX`	Maximum value of signed integer with at least 32 bits width.	2147483647
`UINT_LEAST32_MAX`	Maximum value of unsigned integer with at least 32 bits width.	4294967295
`INT_LEAST64_MIN`	Minimum value of signed integer with at least 64 bits width.	-9223372036854775808
`INT_LEAST64_MAX`	Maximum value of signed integer with at least 64 bits width.	9223372036854775807
`UINT_LEAST64_MAX`	Maximum value of unsigned integer with at least 64 bits width.	18446744073709551615

7.18.10 Limits of Other Integer Types

Type	Description	Definition
PTRDIFF_MIN	Minimum value of the `ptrdiff_t` type.	-32678
PTRDIFF_MAX	Maximum value of the `ptrdiff_t` type.	32767
SIG_ATOMIC_MAX	Maximum value of the `sig_atomic_t` type.	2147483647
SIZE_MAX	Maximum value of the `size_t` type.	4294967295
WCHAR_MAX	Maximum value of the `wchar_t` type.	65535
WINT_MIN	Minimum value of the `wint_t` type.	-2147483648
WINT_MAX	Maximum value of the `wint_t` type.	2147483647

7.18.11 Minimum Width Integer Types

Type	Description	Definition
int_least8_t	Signed integer of at least 8 bits width.	`signed char`
int_least16_t	Signed integer of at least 16 bits width.	`short`
int_least24_t	Signed integer of at least 24 bits width.	`__int24`
int_least32_t	Signed integer of at least 32 bits width.	`long`
int_least64_t	Signed integer of at least 64 bits width, where supported.	`long long`
uint_least8_t	Unsigned integer of at least 8 bits width.	`unsigned char`
uint_least16_t	Unsigned integer of at least 16 bits width.	`unsigned short`
uint_least24_t	Unsigned integer of at least 24 bits width.	`__uint24`
uint_least32_t	Unsigned integer of at least 32 bits width.	`unsigned long`
uint_least64_t	Unsigned integer of at least 64 bits width, where supported.	`unsigned long long`

7.19 <stdio.h> Input and Output

The header file `stdio.h` consists of types, macros and functions that provide support to perform input and output operations.

Streams are not supported by this implementation. Standard functions and macros associated with streams are not present, and their use will generate an error.

The `stdio.h` file contains functions that use input and output formats. The input formats, or scan formats, are used for reading data. Their descriptions can be found under `scanf`, but they are also used by the other functions in the scanf family. The output formats, or print formats, are used for writing data. Their descriptions can be found under `printf`. These print formats are also used by other printf-family functions.

7.19.1 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

Same effect as `fgetc` with the argument `stdin`.

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|
```

7.19.2 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 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 null 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.

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
```

7.19.3 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

```
#include <stdio.h>

int main(void)
{
  FILE *myfile;

  if ((myfile = fopen("samp.fil", "r+")) == NULL)
    perror("Cannot open samp.fil");
  else
    printf("Success opening samp.fil\n");
```

```
  fclose(myfile);
}
```

Example Output

```
Cannot open samp.fil:  file open error
```

7.19.4 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

There must be exactly the same number of arguments as there are format specifiers. If the are less arguments than match the format specifiers, the output is undefined. If there are more arguments than match the format specifiers, the remaining arguments are discarded. Each format specifier begins with a percent sign followed by optional fields and a required type as shown here:

`%[flags][width][.precision][size]type`

`flags`

`-`	Left justify the value within a given field width.
`0`	Use 0 for the pad character instead of space (which is the default).
`+`	Generate a plus sign for positive signed values.
space	Generate a space or signed values that have neither a plus nor a minus sign.
`#`	To prefix 0 on an octal conversion, to prefix 0x or 0X on a hexadecimal conversion, or to generate a decimal point and fraction digits that are otherwise suppressed on a floating-point conversion.

`width`

Specify the number of characters to generate for the conversion. If the asterisk (*) is used instead of a decimal number, the next argument (which must be of type int) 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.

`precision`

The field width can be followed with dot (`.`) and a decimal integer representing the precision that specifies one of the following:

- minimum number of digits to generate on an integer conversion
- number of fraction digits to generate on an e, E, or f conversion
- maximum number of significant digits to generate on a g or G conversion
- maximum number of characters to generate from a C string on an s conversion

If the period appears without the integer, the integer is assumed to be zero. If the asterisk (*) is used instead of a decimal number, the next argument (which must be of type int) will be used for the precision.

`size`

h modifier	Used with type d, i, o, u, x, X; converts the value to a `short int` or `unsigned short int`.
h modifier	Used with n; specifies that the pointer points to a `short int`.
l modifier	Used with type d, i, o, u, x, X; converts the value to a `long int` or `unsigned long int`.
l modifier	Used with n; specifies that the pointer points to a `long int`.
l modifier	Used with c; specifies a wide character.
l modifier	Used with type e, E, f, F, g, G; converts the value to a `double`.
ll modifier	Used with type d, i, o, u, x, X; converts the value to a `long long int` or `unsigned long long int`.
ll modifier	Used with n; specifies that the pointer points to a `long long int`.
L modifier	Used with e, E, f, g, G; converts the value to a `long double`.

`type`

d, i	`signed int`.
o	`unsigned int` in octal.
u	`unsigned int` in decimal.
x	`unsigned int` in lowercase hexadecimal.
X	`unsigned int` in uppercase hexadecimal.
e, E	`double` in scientific notation.
f	`double` decimal notation.
g, G	`double` (takes the form of e, E or f as appropriate).
c	`char` - a single character.
s	`string`.
p	value of a pointer.
n	The associated argument shall be an integer pointer into which is placed the number of characters written so far. No characters are printed.
%	A `%` character is printed.

Example

```
#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
```

7.19.5 putc Function

Puts a character to the stream.

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

`putc` is the same as the function `fputc`.

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 = putc(*y, stdout);
    putc('|', stdout);
```

```
  }
}
```

Example Output

```
T|h|i|s| |i|s| |t|e|x|t|
|
```

7.19.6 putch Function

Put a character to `stdout`.

Include

`<stdio.h>`

Prototype

`void putch(char c);`

Argument

c	the character to be written

Remarks

The `putch()` function is provided as an empty stub which can be completed as required by the project. It must be defined if you intend to use the `printf()` function. Typically this function will send its argument to a peripheral that you intend to associate with `stdout`.

Example

```
#include <stdio.h>

const char * x = "This is a string";

int main(void)
{
  const char * cp;

  cp = x;
  while(*cp)
    putch(*cp++);
  putch('\n');
}
```

Example Output

```
This is a string
```

7.19.7 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.

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
```

7.19.8 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.

Example

```
#include <stdio.h>

int main(void)
{
  char buf[] = "This is text\n";

  puts(buf);
  puts("|");
}
```

Example Output

```
This is text

|
```

7.19.9 scanf Function

Scans formatted text from `stdin`.

Include

`<stdio.h>`

Prototype

`int scanf(const char *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.

Remarks

Each format specifier begins with a percent sign followed by optional fields and a required type as shown here:

`%[*][`*`width`*`][`*`modifier`*`]`*`type`*

`*`

Indicates assignment suppression. This will cause the input field to be skipped and no assignment made.

`width`

Specify the maximum number of input characters to match for the conversion, not including white space that can be skipped.

`modifier`

h modifier	Used with type d, i, o, u, x, X; converts the value to a `short int` or `unsigned short int`.
h modifier	Used with n; specifies that the pointer points to a `short int`.
l modifier	Used with type d, i, o, u, x, X; converts the value to a `long int` or `unsigned long int`.
l modifier	Used with n; specifies that the pointer points to a `long int`.
l modifier	Used with c; specifies a wide character.
l modifier	Used with type e, E, f, F, g, G; converts the value to a `double`.
ll modifier	Used with type d, i, o, u, x, X; converts the value to a `long long int` or `unsigned long long int`.
ll modifier	Used with n; specifies that the pointer points to a `long long int`.
L modifier	Used with e, E, f, g, G; converts the value to a `long double`.

`type`

d, i	`signed int`.
o	`unsigned int` in octal.
u	`unsigned int` in decimal.
x	`unsigned int` in lowercase hexadecimal.
X	`unsigned int` in uppercase hexadecimal.
e, E	`double` in scientific notation.
f	`double` decimal notation.
g, G	`double` (takes the form of e, E or f as appropriate).
c	`char` - a single character.

s	`string`.
p	value of a pointer.
n	The associated argument shall be an integer pointer into which is placed the number of characters written so far. No characters are printed.
%	A `%` character is printed.

Example

For MPLAB X Simulator:

```
#include <stdio.h>
 #include <libpic30.h>

int main(void)
{
  int number, items;
  char letter;
  char color[30], string[30];
  float salary;

  __attach_input_file("UartIn.txt");
  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);
}
// If not using the simulator, remove these lines:
// #include <libpic30.h>
// __attach_input_file("uart_in.txt");
```

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
```

7.19.10 sprintf Function

Prints formatted text to a string.

Include

`<stdio.h>`

Prototype

`int sprintf(char *`**`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 format argument has the same syntax and use that it has in `printf`.

Example

```
#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
```

7.19.11 sscanf Function

Scans formatted text from a string.

Include

`<stdio.h>`

Prototype

`int sscanf(const char *s, const char *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

The format argument has the same syntax and use that it has in `scanf`.

Example

```
#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
```

7.20 <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.

7.20.1 stdlib.h Types and Macros

The following types are included in `stdlib.h`:

- div_t
- ldiv_t
- wchar_t

The following macros are included in `stdlib.h`:

- EXIT_FAILURE
- EXIT_SUCCESS
- MB_CUR_MAX
- RAND_MAX

7.20.1.1 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`.

7.20.1.2 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`.

7.20.1.3 wchar_t

A type that holds a wide character value. In `stdlib.h` and `stddef.h`.

7.20.1.4 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.

7.20.1.5 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.

7.20.1.6 RAND_MAX

Maximum value capable of being returned by the rand function.

Value

32767

7.20.2 abort Function

Aborts the current process.

Include

`<stdlib.h>`

Prototype

`void abort(void);`

Remarks

The default implementation of `abort` calls the `exit` function with argument value of 1, which in turn loops infinitely.

Example

```
#include <stdlib.h>

int process(char * ptr)
{
  if(ptr == NULL) {
    abort();
  }
  return modul(*prt);
}
```

7.20.3 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

```
#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
```

7.20.4 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	function to be called

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 the exit function call.

Example

```
#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
```

7.20.5 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

```
#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 = "27.835:e2" float = 2783.500000
String = "Number1"  float = 0.000000
```

7.20.6 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

```
#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
```

7.20.7 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

```
#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
```

7.20.8 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

```
#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
```

7.20.9 calloc Function

Allocates an array in memory and initializes the elements to 0.

Include

`<stdlib.h>`

Prototype

`void *calloc(size_t nelem, size_t size);`

Arguments

nelem	number of elements
size	length of each element

Return Value

Returns a pointer to the allocated space if successful; otherwise, returns a null pointer.

Remarks

Memory returned by `calloc` is aligned correctly for any size data element and is initialized to zero. This function requires a heap.

Example

```
/* This program allocates memory for the      */
/* array 'i' of long integers and initializes */
/* them to zero.                              */

#include <stdio.h>  /* for printf, NULL */
#include <stdlib.h> /* for calloc, free */

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
```

7.20.10 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`). Division by zero will invoke the math exception error, which, by default, will cause a Reset. Write a math error handler to do something else.

Example

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

void __attribute__((__interrupt__))
_MathError(void)
{
  printf("Illegal instruction executed\n");
  abort();
}

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);

    x = 7;
    y = 0;
    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

For div(7, 0)
Illegal instruction executed
ABRT
```

7.20.11 exit Function

Terminates program after clean up.

Include

`<stdlib.h>`

Prototype

`void exit(int status);`

Argument

`status`	exit status

Remarks

`exit` calls any functions registered by `atexit` in reverse order of registration, flushes buffers, closes stream, closes any temporary files created with `tmpfile` and resets the processor. This function is customizable. See `pic30-libs`.

Example

```
#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
```

7.20.12 free Function

Frees memory.

Include

`<stdlib.h>`

Prototype

`void free(void *ptr);`

Argument

ptr	points to 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. This function requires a heap.

Example

```
#include <stdio.h>  /* for printf, sizeof, */
                    /* NULL                */
#include <stdlib.h> /* for malloc, free    */

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
```

7.20.13 getenv Function

Get a value for an environment variable.

Include

<stdlib.h>

Prototype

`char *getenv(const char *name);`

Argument

name	name of environment variable

Return Value

Returns a pointer to the value of the environment variable if successful; otherwise, returns a null pointer.

Example

```
#include <stdlib.h> /* for getenv       */

int main(void)
{
  char *incvar;

  incvar = getenv("INCLUDE");
  if (incvar == NULL)
    abort();
}
```

7.20.14 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

```
#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
```

7.20.15 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

```
#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.

7.20.16 malloc Function

Allocates memory.

The default implementation of `malloc` will require an additional 4 bytes of heap memory per allocation.

The legacy library's `malloc` will use an additional 2 bytes of heap memory per allocation.

Include

`<stdlib.h>`

Prototype

`void *malloc(size_t size);`

Argument

size	number of characters to allocate

Return Value

Returns a pointer to the allocated space if successful; otherwise, returns a null pointer.

Remarks

`malloc` does not initialize memory it returns. This function requires a heap.

Example

```
#include <stdio.h>  /* for printf, sizeof, */
                    /* NULL                */
#include <stdlib.h> /* for malloc, free    */

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
```

7.20.17 strtod Function

Description

Convert string to `double` floating-point value.

Include

`<stdlib.h>`

Prototype

`double strtod(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 `strtod` function attempts to convert the first part of the string pointed to by `nptr` to a `double` floating-point value.

Any initial whitespace characters in the string are skipped. The following characters represent the floating-point constant. Conversion stops once an unrecognized character is encountered in the string.

The expected form of the floating-point constant is 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 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:

Example

```
#include <stdlib.h> /* for strtod */
#include <stdio.h> /* for printf */

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"
```

7.20.18 strtof Function

Description

Convert string to `float` floating-point value.

Include

`<stdlib.h>`

Prototype

`float strtod(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.

Any initial whitespace characters in the string are skipped. The following characters represent the floating-point constant. Conversion stops once an unrecognized character is encountered in the string.

The expected form of the floating-point constant is 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 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:

Example

```
#include <stdlib.h> /* for strtof */
#include <stdio.h> /* for printf */

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"
```

7.20.19 strtol Function

Description

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.

Example

```
#include <stdlib.h> /* for strtol */
#include <stdio.h> /* for printf */

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"
```

7.20.20 strtold Function

Description

Convert string to `long double` floating-point value.

Include

<stdlib.h>

Prototype

`long double strtod(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.

Any initial whitespace characters in the string are skipped. The following characters represent the floating-point constant. Conversion stops once an unrecognized character is encountered in the string.

The expected form of the floating-point constant is 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 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:

Example

```
#include <stdlib.h> /* for strtold */
#include <stdio.h> /* for printf */

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"
```

7.20.21 strtoul Function

Description

Convert string to `unsigned long` integer value.

Include

<stdlib.h>

Prototype

`unsigned 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 `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.

Example

```
#include <stdlib.h> /* for strtoul */
#include <stdio.h> /* for printf */

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"
```

7.20.22 system Function

Execute a command.

Include

`<stdlib.h>`

Prototype

`int system(const char *s);`

Argument

s	command to be executed

Default Behavior

As distributed, this function acts as a stub or placeholder for your function. If s is not NULL, an error message is written to `stdout` and the program will reset; otherwise, a value of -1 is returned.

7.21 <time.h> Date and Time Functions

The header file `time.h` consists of types, macros and functions that manipulate date and time.

7.21.1 time.h Types and Macros

clock_t

Stores processor time values.

Prototype

```
typedef unsigned long clock_t
```

struct tm

Structure used to hold the time and date (calendar time).

Prototype

```
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*/
}
```

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.

time_t

Represents calendar time values.

Prototype

```
typedef unsigned long time_t
```

The following macro is included in `time.h`

CLOCKS_PER_SEC

Number of processor clocks per second.

Prototype

```
extern clock_t clock(void);
```

Value

User defined.

Remarks

This is defined as an `extern` variable whose value is provided at link time.

7.21.2 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

```
#include <time.h>  /* for asctime, tm */
#include <stdio.h> /* for printf      */

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
```

7.21.3 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

```
#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
```

7.21.4 difftime Function

Find the difference between two times.

Include

`<time.h>`

Prototype

`int32_t 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

This function differs to the standard C function in that it returns a long integer type rather than `double`. This is due to a lack of a 64-bit `double` type.

Example

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

volatile int i;

int main(void)
{
  clock_t start, stop;
  int32_t 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 = %ld\n", elapsed);
}
```

Example Output

```
start = 0
stop = 317
Elapsed time = 317
```

7.21.5 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

```
#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
```

7.21.6 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 16-bit 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

```
#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
```

7.21.7 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

```
#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
```

7.21.8 strftime Function

Formats the time structure to a string based on the format parameter.

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 format parameters follow:

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	appropriate date and time representation
%d	day of the month (01-31)
%H	hour of the day (00-23)
%I	hour of the day (01-12)
%j	day of the year (001-366)
%m	month of the year (01-12)
%M	minute of the hour (00-59)
%p	AM/PM designator
%S	second of the minute (00-61) allowing for up to two leap seconds
%U	week number of the year where Sunday is the first day of week 1 (00-53)
%w	weekday where Sunday is day 0 (0-6)
%W	week number of the year where Monday is the first day of week 1 (00-53)

%x	appropriate date representation
%X	appropriate time representation
%y	year without century (00-99)
%Y	year with century
%Z	time zone (possibly abbreviated) or no characters if time zone is unavailable
%%	percent character %

Example

```
#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.
```

7.21.9 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. This function is customizable (see `pic30-libs`).

Example

```
#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 */
  time(&ticks); /* get time */
  printf("Time = %ld\n", ticks);
}
```

Example Output

```
Time = 256
```

8. Document Revision History

Revision A (March 2018)
Initial release of this document, adapted from the MPLAB XC8 C Compiler User's Guide, DS50002053.

Revision B (March 2019)

- Added information relating to `const`-specified objects being located in program memory
- Added information on the new code coverage feature
- Added information relating to chipinfo HTML files
- Added descriptions and screen captures of the MPLAB X IDE project property dialogs corresponding to the compiler command-line options
- Updated configuration bit information
- Clarified information relating to absolute objects
- Updated predefined macros table
- Miscellaneous corrections and improvements

Revision C (March 2020)

- This guide has been migrated to a new authoring and publication system; you may see differences in the formatting compared to previous revisions
- The documentation for the standard libraries has been updated
- Updated information relating to the structure of the DFPs
- Clarified and expanded information relating to optimizations

The Microchip Website

Microchip provides online support via our website at http://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 http://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: http://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 specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "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

Trademarks

ISBN: 978-1-5224-5830-2

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit http://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: http://www.microchip.com/support Web Address: http://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: 45-4450-2828 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