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

Notice to Development Tools Customers

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

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

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

Table of Contents

Notice to Development Tools Customers....1

1. Preface....4
   - 1.1. Conventions Used in This Guide....4
   - 1.2. Recommended Reading....5

2. Compiler Overview....6
   - 2.1. Device Description....6
   - 2.2. C Standards....6
   - 2.3. Hosts and Licenses....7
   - 2.4. Conventions....7
   - 2.5. Compatible Development Tools....7

3. Command-line Driver....8
   - 3.1. Invoking The Compiler....8
   - 3.2. The Compilation Sequence....10
   - 3.3. Runtime Files....14
   - 3.4. Compiler Output....14
   - 3.5. Compiler Messages....15
   - 3.6. Option Descriptions....16
   - 3.7. MPLAB X IDE Integration....47
   - 3.8. Microchip Studio Integration....55

4. C Language Features....67
   - 4.1. C Standard Compliance....67
   - 4.2. Device-Related Features....67
   - 4.3. Supported Data Types and Variables....73
   - 4.4. Memory Allocation and Access....86
   - 4.5. Operators and Statements....92
   - 4.6. Register Usage....94
   - 4.7. Functions....94
   - 4.8. Interrupts....98
   - 4.9. Main, Runtime Startup and Reset....102
   - 4.10. Libraries....105
   - 4.11. Mixing C and Assembly Code....107
   - 4.12. Optimizations....115
   - 4.13. Preprocessing....117
   - 4.14. Linking Programs....120

5. Utilities....123
   - 5.1. Archiver/Librarian....123
   - 5.2. Objdump....124

6. Implementation-Defined Behavior....125
   - 6.1. Overview....125
   - 6.2. Translation....125
   - 6.3. Environment....125
   - 6.4. Identifiers....126

6.5. Characters ... 126
6.6. Integers ... 126
6.7. Floating-Point ... 127
6.8. Arrays and Pointers ... 128
6.9. Hints ... 128
6.10. Structures, Unions, Enumerations, and Bit-Fields ... 128
6.11. Qualifiers ... 128
6.12. Pre-Processing Directives ... 129
6.13. Library Functions ... 129
6.14. Architecture ... 132

7. Common C Interface ... 133
7.1. Background - The Desire for Portable Code ... 133
7.2. Using the CCI ... 135
7.3. C Language Standard Refinement ... 135
7.4. C Language Standard Extensions ... 142
7.5. Compiler Features ... 155

8. Library Functions ... 156
8.1. Library Example Code ... 156
8.2. <boot.h> Bootloader Functions ... 158
8.3. <cpufunc.h> CPU Related Functions ... 166
8.4. <delay.h> Delay Functions ... 166
8.5. <pgmspace.h> ... 168
8.6. <sfr_defs.h> ... 174
8.7. <sleep.h> ... 176
8.8. <xc.h> ... 179
8.9. Built-in Functions ... 181

9. Document Revision History ... 187

Microchip Information ... 190
The Microchip Website ... 190
Product Change Notification Service ... 190
Customer Support ... 190
Product Identification System ... 191
Microchip Devices Code Protection Feature ... 191
Legal Notice ... 191
Trademarks ... 192
Quality Management System ... 193
Worldwide Sales and Service ... 194

1. Preface

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation. In most cases, formatting conforms to the *OASIS Darwin Information Typing Architecture (DITA) Version 1.3 Part 3: All-Inclusive Edition*, 19 June 2018.

Table 1-1. Documentation Conventions

Description	Implementation	Examples
References	DITA: cite	*MPLAB® XC8 C Compiler User's Guide for AVR® MCU.*
Emphasized text	Italics	...is the *only* compiler...
A window, window pane or dialog name.	DITA: wintitle	the **Output** window. the **New Watch** dialog.
A field name in a window or dialog.	DITA: uicontrol	Select the **Optimizations** option category.
A menu name or item.	DITA: uicontrol	Select the **File** menu and then **Save**.
A menu path.	DITA: menucascade, uicontrol	**File** > **Save**
A tab	DITA: uicontrol	Click the **Power** tab.
A software button.	DITA: uicontrol	Click the **OK** button.
A key on the keyboard.	DITA: uicontrol	Press the **F1** key.
File names and paths.	DITA: filepath	`C:/Users/User1/Projects`
Source code: inline.	DITA: codeph	Remember to `#define START` at the beginning of your code.

Source code: block.	DITA: codeblock	An example is:
		`#include <xc.h>`
		`main(void) {`
		`  while(1);`
		`}`

User-entered data.	DITA: userinput	Type in a device name, for example `PIC18F47Q10`.
Keywords	DITA: codeph	`static`, `auto`, `extern`
Command-line options.	DITA: codeph	`-Opa+, -Opa-`
Bit values	DITA: codeph	`0, 1`
Constants	DITA: codeph	`0xFF, 'A'`
A variable argument.	DITA: codeph + option	*`file`*`.o`, where *`file`* can be any valid file name.
Optional arguments	Square brackets [ ]	`xc8 [`*`options`*`] `*`files`*
Choice of mutually exclusive arguments; an OR selection.	Curly brackets and pipe character: { | }	`errorlevel {0|1}`
Replaces repeated text.	Ellipses...	*`var_name`* `[, `*`var_name`*`...]`

Represents code supplied by user.	Ellipses...	`void main (void)`
		`{ ...`
		`}`

A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	N'Rnnnn	4'b0010, 2'hF1
Device Dependent insignia. Specifies that a feature is not supported on all devices. Please see your device data sheet for details.	[DD]	Assembler Special Operators

1.2 Recommended Reading

This user's guide describes the use and features of the MPLAB XC8 C Compiler when building for AVR targets and using the ISO/IEC 9899:1999 Standard (C99) for programming languages. The following Microchip documents are available and recommended as supplemental reference resources.

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

This version of the compiler's user's guide is for projects that target 8-bit PIC devices.

MPLAB® XC8 C Compiler Release Notes for AVR® MCU

For the latest information on using MPLAB XC8 C Compiler, read MPLAB® XC8 C Compiler Release Notes (an HTML file) in the `docs` subdirectory of the compiler's installation directory. The release notes contain the latest information and known issues that might not be included in this user's guide.

Microchip Unified Standard Library Reference Guide

This guide contains information and examples relating to the types, macros, and functions defined by the C standard libraries and which is shipped with all MPLAB XC C compilers.

AVR® GNU Toolchain to MPLAB® XC8 Migration Guide

This guide describes the changes to source code and build options that might be required should you decide to migrate a C-based project from the AVR 8-bit GNU Toolchain to the Microchip MPLAB XC8 C Compiler.

Development Tools Release Notes

For the latest information on using other development tools, refer to the tool-specific Readme files in the docs subdirectory of the MPLAB X IDE installation directory.

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), plus assembly instruction implemented in addition to the basic set. Note that the avr1 family supports programming in assembly only.

See Print-devices for information on finding the full list of devices that are supported by the compiler. Support for a new device might be possible after downloading an updated Device Family Pack.

Table 2-1. Summary of Supported Device Families

Family	ArchID	SFR Offset	Additional Instructions
avr1	1	0x20	
avr2	2	0x20	
avr25	25	0x20	`lpmx`, `movw`
avr3	3	0x20	`jmp`
avr31	31	0x20	`jmp`, `elpm`
avr35	35	0x20	`jmp`, `lpmx`, `movw`
avr4	4	0x20	`mul`, `lpmx`, `movw`
avr5	5	0x20	`mul`, `jmp`, `lpmx`, `movw`
avr51	51	0x20	`mul`, `jmp`, `lpmx`, `movw`, `elpm`, `elpmx`
avr6	6	0x20	`mul`, `jmp`, `lpmx`, `movw`, `elpm`, `elpmx`, `eijmp`
avrtiny	100	0x0	
avrxmega2	102	0x0	`mul`, `jmp`, `lpmx`, `movw`
avrxmega3	103	0x0	`mul`, `jmp`, `lpmx`, `movw`
avrxmega4	104	0x0	`mul`, `jmp`, `lpmx`, `movw`, `elpm`, `elpmx`
avrxmega5	105	0x0	`mul`, `jmp`, `lpmx`, `movw`, `elpm`, `elpmx`
avrxmega6	106	0x0	`mul`, `jmp`, `lpmx`, `movw`, `elpm`, `elpmx`, `eijmp`
avrxmega7	107	0x0	`mul`, `jmp`, `lpmx`, `movw`, `elpm`, `elpmx`

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 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 (www.microchip.com/mplab/mplab-x-ide) for all 8-bit PIC and AVR devices
- The Microchip Studio for AVR® and SAM Devices (www.microchip.com/mplab/microchip-studio) for all 8-bit AVR devices
- The MPLAB X Simulator
- The Command-line MDB Simulator—see the Microchip Debugger (MDB) User’s Guide (DS52102)
- All Microchip debug tools and programmers (www.microchip.com/mplab/development-boards-and-tools)
- Demonstration boards and Starter kits that support 8-bit AVR devices

Check the tool's documentation to verify that it supports the device you plan to use.

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 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 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. The MPLAB X IDE also allows such files to be used. The file name is specified in the **XC8 Linker** > **Additional options** > **Use response file to link** field of the Project Properties.

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.

Typically, your IDE will allow the compiler to be selected in the project's properties, without the need for the `PATH` variable to be defined.

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 base 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. 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 name of the IDE project that contains them.

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*.

These terms can also be applied to assembly source files, which can be preprocessed and include other (`.inc`) files to produce an assembly module.

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

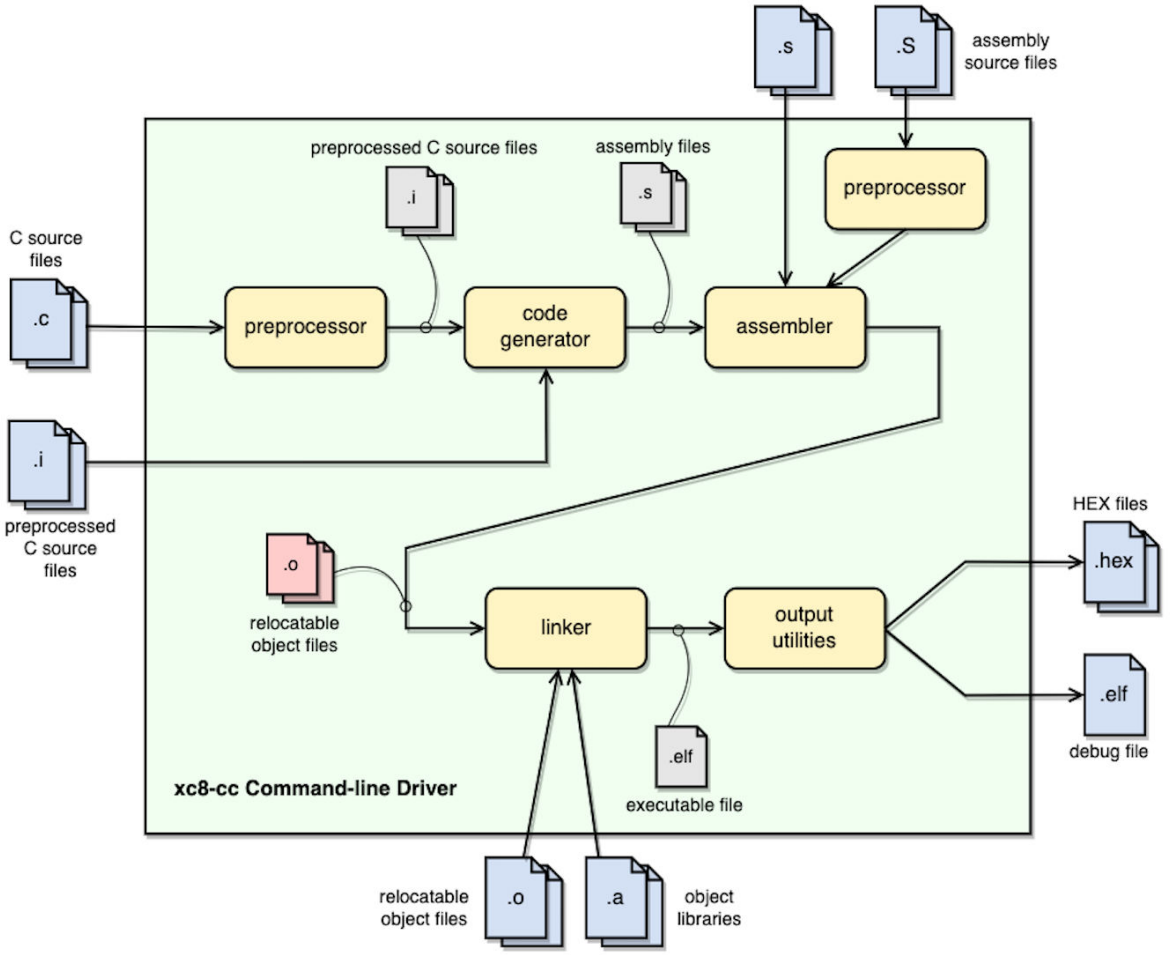
The main internal compiler applications and files are shown in the illustration below.

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 archiver (`xc8-ar`) internal application be executed directly. The archiver's command-line options are described in Utilities. A separate Hexmate User's Guide is available if you need to run this HEX file manipulation utility after building your project.

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 used by your development environment.

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 and make utilities must be employed to achieve incremental builds (see Multi-Step C Compilation).

Unless otherwise specified, an ELF file (this is called `a.out` without use of the `-o` option) 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 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 when building with an IDE (see also 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  -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 can also be performed by integrated development environments.

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 files corresponding to the unchanged sources file do not need to 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 perform the link step. The resulting 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 linker options are complex.

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 Library Example Code. For more information on creating and using your own libraries, see 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 Library Example Code 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 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 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
`.hex`	Intel HEX	`avr-objcopy` application

.....continued		
Extension	File Type	How created
.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.

The ELF/DWARF file is used by debuggers to obtain debugging information about the project and allows for more accurate debugging compared to other formats, such as COFF. Some aspects of the project's operation might not even be available to your debugger when using COFF. Development environments 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 a 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 a development environment, it will issue explicit options to the compiler that are based on the selections in the project's properties. 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 listed below are useful when compiling 8-bit Microchip AVR devices with the MPLAB XC8 compiler and are discussed in following sections.

Table 3-4. AVR Device-Specific Options

Option (links to explanatory section)	Controls
`-m[no-]accumulate-args`	How arguments are passed between functions.
`-m[no-]call-prologues`	How functions save and restore registers.
`-m[no-]const-data-in-config-mapped-progmem`	Whether the compiler will place `const`-qualified data into a 32k flash section that is then mapped into data memory.
`-m[no-]const-data-in-progmem`	Whether `const`-qualified objects are placed in program or data memory.
`-mcpu=`*`device`*	The target device or architecture to compile.
`-m[no-]diva`	Whether the DIVA module is utilized for some integer operations.
`-mdfp=`*`path`*	The source of device-specific information.
`-mfuse-action`	Adjust code memory size, based on configuration fuses values.
`-mno-interrupts`	How the stack pointer is changed.
`-f[no-]jump-tables`	Whether jump tables are used in `switch()` statements.
`-m[no-]relax`	Optimization of call/jump instructions.
`-mreserve=`*`space@start:end`*	The address ranges in the memory space to be reserved.
`-mshort-calls`	How function calls are encoded.
`-msmart-io=`*`level`*	The feature set of the IO library linked in.
`-msmart-io-format="`*`format`*`"`	Manual inclusion of IO library features.
`-m[no-]strict-X`	The use of the X register.
`-m[no-]tiny-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-isr-prologues Option

The `-mcall-isr-prologues` option changes how interrupt functions save registers on entry and how those registers are restored when the interrupt routine terminates. It works in a similar way to the `-mcall-prologues` option, but only affects interrupt functions (ISRs).

If this option is not specified, the registers that need to be preserved by ISRs will be saved and restored by code inside those functions. When an ISR calls another function, there will be a large number of registers that need to be preserved. If the `-mcall-isr-prologues` option is used, this preservation code is extracted as subroutines that are called at the appropriate points in the ISR.

Use of this option can reduce code size if there is more than one interrupt function that contains a call to another function, but it can increase the code's execution time.

3.6.1.3 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.

3.6.1.4 Const-data-in-config-mapped-progmem Option

The `-mconst-data-in-config-mapped-progmem` option stores objects qualified with `const` to an area of program memory that the compiler will ensure is mapped into the data memory space for some devices.

Certain devices, such as the AVR DA and AVR DB family, have an SFR (e.g. `FLMAP`) that specifies which 32 KB section of program memory will be mapped into the data memory space. The `-mconst-data-in-config-mapped-progmem` option can be used to have the linker place all `const`-qualified data within one of these 32 KB sections and automatically initialize the relevant SFR register to ensure that these objects are mapped into data memory, where they will be accessed more efficiently in terms of program size. The option must be used in conjunction with the `-mconst-data-in-progmem` option; however, this option is enabled by default.

The `-mconst-data-in-config-mapped-progmem` option is automatically enabled for devices that can support this mapping feature. In that case, the feature can be disabled by using the `-mno-const-data-in-config-mapped-progmem` option, forcing `const`-qualified objects to be accessed from program memory.

As the `-mconst-data-in-config-mapped-progmem` option affects both how code is generated and the choice of library to link against, it (or its negated `no-` form) must be used with both compile and link steps of the build process.

When this option is selected, the runtime startup code will set the `FLMAPLOCK` bit. Doing so prevents any modification of the `FLMAP` register. This is necessary since the code generated by the compiler assumes the page mapped by the compiler at compile time will still be mapped at runtime. If a program could change the mapping at runtime, any code that reads `const` data might fail.

Although the mapped page cannot be changed at runtime, you have some control over which page will be mapped. If you use the `-mconst-data-in-config-mapped-progmem` option as well as the `__at()` construct to place `const` data in a page of your choice, the linker will attempt to allocate all `const` data in the same page and ensure that page is mapped. For example:

```
const char __at(0xc000) arr[3] = {3, 1, 2};  // Force arr into page 1
const char arr2[3] = {3, 1, 2};              // arr2 will be placed in the same page
```

If `__at()` is used more than once to force objects into different pages, the compiler will generate an error and suggest disabling the `-mconst-data-in-config-mapped-progmem` option. If you place some `const` data using `__at()` but you also need other `const` data in an alternate page that is not mapped (will be accessed using LPM instructions), then use the `__memx` specifier with the other data. For example:

```
const char __at(0xc000) arr[3] = {8, 1, 2};        // Force arr into mapped page 1
const char __at(0x300) __memx arr2[3] = {7, 1, 2}; // Force arr2 into page 0
```

If more than 32 KB of `const`-qualified data is defined or the section holding this data cannot otherwise be located in the mapped memory section, an error from the linker will be issued.

The macro `__AVR_CONST_DATA_IN_CONFIG_MAPPED_PROGMEM__` is defined if this option is enabled and `const`-qualified objects are located in a mapped section of memory.

3.6.1.5 Const-data-in-progmem Option

The `-mconst-data-in-progmem` option changes the location of where objects defined using just the `const` type qualifier are stored.

By default, all objects defined using just the `const` type qualifier are located in and accessed from program memory, and this action can be made explicit by using the `-mconst-data-in-progmem` option. Any pointer to a `const`-qualified object will be able to read both data and program memory and determine at runtime which space is to be accessed. See also Objects in Program Space.

The `-mno-const-data-in-progmem` option forces all objects defined using just the `const` type qualifier to be copied to data memory, where they can be read using different instructions when required. Disabling this feature does not affect code built for devices that have flash mapped into data memory, for example the avrxmega3 and avrtiny architectures. Pointers to objects qualified with just `const` will always read from data memory, and alternatives to the string functions normally provided by `<string.h>` must be used to access strings that have been located in program memory.

The macro `__AVR_CONST_DATA_IN_PROGMEM__` is defined if this feature is enabled and `const`-qualified objects are located in and accessed from program memory.

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.

3.6.1.6 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 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 8 KB program memory.
`avr25`	avr2 with `movw` and `lpm Rx,Z[+]` instructions.
`avr3`	Classic core with up to 64 KB extended program memory.
`avr31`	Classic core with 128 KB of program memory.
`avr35`	avr3 with `movw` and `lpm Rx,Z[+]` instructions.
`avr4`	Enhanced core up to 8 KB program memory.
`avr5`	Enhanced core up to 64 KB program memory.
`avr51`	Enhanced core 128 KB program memory.
`avr6`	Enhanced core 256 KB program memory.
`avrxmega2`	XMEGA core, up to 64 KB program memory, up to 64 KB data address space.
`avrxmega3`	XMEGA core with program memory mapped into data address space.
`avrxmega4`	XMEGA core, up to 128 KB program memory, up to 64 KB data address space.
`avrxmega5`	XMEGA core, up to 128 KB program memory, greater than 64 KB data address space.
`avrxmega6`	XMEGA core, greater than 128 KB program memory, up to 64 KB data address space.
`avrxmega7`	XMEGA core, greater than 128 KB program memory, greater than 64 KB data address space.
`avrtiny`	tinyAVR core, 16 registers.

3.6.1.7 DIVA Option

The `-mdiva` option enables the use of routines that use the DIVA module to perform integer hardware divisions, shifts and rotates on devices that implement this module. This is the default

operation for these devices if this option is not used. See Division, Shift and Modulo Operations for more information.

The `-mno-diva` form of this option disables use of these routines, having these operations instead performed by conventional library routines that will be larger and slower than the DIVA routines.

3.6.1.8 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 the `dfp` directory of the compiler distribution.

The Microchip development environments 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/` (where *`DFP`* represents the path to the DFP) before the standard search paths.

3.6.1.9 Fuse-action Option

The `-mfuse-action` option adjusts the linker's memory allocation, based on the value of certain configuration fuses. The option takes an argument to indicate the required action. At present, the only allowable argument is `define-appdata`.

Some AVR devices have Flash memory arranged into contiguous BOOT, APPCODE, and APPDATA sections. See your device data sheet to see if this arrangement is relevant for your target device, and how the memory operates. On these devices, the `BOOTSIZE` and `CODESIZE` configuration words control the length of the BOOT and APPCODE sections, hence also control the starting address of the APPDATA section.

The linker ordinarily places code and read-only data in any available program Flash memory; however, it is preferable that the linker does not place anything in the APPDATA section unless explicitly requested to do so. With larger programs for example, code might “overflow” from the APPCODE region into the APPDATA section.

The `-mfuse-action=define-appdata` option has the compiler interpret the `BOOTSIZE` and `CODESIZE` configuration words so that it can determine the starting address of the APPDATA section. It then adjusts the linker's options so that code and read-only data will not be linked into the APPDATA section, unless the user explicitly requests to do so (by using absolute addresses and the `--section-start` option). If no fuse section contents are present, then the default value for those fuse registers is used for these calculations. This is the default action for relevant devices if no option is specified.

The `-mno-fuse-action` form of this option explicitly disables this feature, allowing placement of code and read-only data in any available Flash memory.

3.6.1.10 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.

3.6.1.11 Jump-tables Option

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

3.6.1.12 Relax Option

The `-mrelax` option enables several optimizations that relate to the efficient use of call and jump instructions.

These optimizations include replacement of the long-form call and jump instructions with shorter and/or faster relative calls and jumps when the relative forms of the instructions can be determined to be in range of their destination. For example, code such as:

```
  call foo
  ...

foo:
  ...
```

can be replaced with:

```
  rcall foo
  ...

foo:
  ...
```

when `foo` is within range of the relative call. (For more information, see Function Pointers).

When relative calls would be out of range, the compiler will attempt to replace them with `rcall` instructions to a `jmp` instruction that will 'trampoline' execution to the required address. For example, code such as:

```
  call foo
  ...
  call foo
  ...
  call foo
```

can be changed to:

```
  rcall tramp_foo
  ...
  rcall tramp_foo
  ...
  rcall tramp_foo
  ...

tramp_foo:
  jmp foo
```

provided the trampoline is within range of the relative calls. The compiler will re-use an existing `jmp` instruction to the required destination if possible, to further reduce code size. Such transformations will slow execution speed but can improve code size if there multiple calls that can be replaced.

Additionally, redundant return instructions are removed from functions with a tail call. For example, code such as:

```
  ...
  call last
  return
last:
  ...
  return
```

can be replaced with:

```
  ...
  jmp last
last:
  ...
  return
```

The `-mno-relax` form of this option does not apply these optimizations and is the default action if no option is specified.

3.6.1.13 Reserve Option

The `-mreserve=`*`ranges`* option allows you to reserve memory normally used by the program. This option has the general form:

```
-mreserve=space@start:end
```

where *`space`* can be either of `ram` or `rom`, denoting the data and program memory spaces, respectively; and *`start`* and *`end`* are addresses, denoting the range to be excluded. For example, `-mreserve=ram@0x100:0x101` will reserve two bytes starting at address 100h from the data memory.

3.6.1.14 Smart-io Option

The `-msmart-io=`*`level`* option in conjunction with the IO format string conversion specifications detected in your program control the feature set (hence size) of the library code that is linked in to perform formatted IO through functions like `printf`. See Smart IO Routines for more information on how the smart IO feature operates.

A numerical level of operation can be specified and these have the meaning shown in the following table.

Table 3-6. Smart IO Implementation Levels

Level	Smart IO features; linked library
0	Disabled; Full-featured library (largest code size)
1	Enabled; Minimal-featured library (smallest code size)

When the smart IO feature is disabled (`-msmart-io=0`), a full implementation of the IO functions will be linked into your program. All features of the IO library functions will be available, and these may consume a significant amount of the available program and data memory on the target device.

The default setting is for smart IO to be enabled with a minimal feature set. This can be made explicit by using either the `-msmart-io=1` or `-msmart-io` option. When thus enabled, the compiler will link in the least complex variant of the IO library that implements all of the IO functionality required by the program, based on the conversion specifications detected in the program's IO function format strings. This can substantially reduce the memory requirements of your program, especially if you can eliminate in your program the use of floating-point features.

If the format string in a call to an IO function is not a string literal, the compiler will not be able to detect the exact usage of the IO function and a full-featured variant of the IO library will be linked into the program image, even with smart IO enabled. In this instance, the `-msmart-io-format` option can be used to force the compiler to instead link in a less complex variant of the library

whose exact feature set is controlled using the `-msmart-io-format` option (see Smart-io-format Option).

These options should be used consistently across all program modules to ensure an optimal selection of the library routines included in the program image.

3.6.1.15 Smart-io-format Option

The `-msmart-io-format="fmt"` option, where *`fmt`* is a string containing formatted, printf-style conversion specifications, notifies the compiler that the listed specifications are used by smart IO functions and that the IO library code linked in must be able to process them. See Smart IO Routines for more information on this feature.

To reduce code size, the compiler customizes library code associated with the print and scan families of smart IO functions, based on the conversion specifications present in the format strings collated across all calls to these functions. This feature is controlled by the `-msmart-io` option.

In some situations, the compiler is unable to determine usage information from the formatted IO function call. In such cases, the `-msmart-io-format="fmt"` option can be used to indicate the required conversion specifications for the linked IO functions, for example, `-msmart-io-format="%d%i%s."` Without this option, the compiler makes no assumptions about IO usage and ensures that fully functional IO functions are linked into the final program image.

For example, consider the following calls to smart IO functions.

```
vscanf("%d:%li", va_list1);
vprintf("%-s%d", va_list2);
vprintf(fmt1, va_list3);   // ambiguous usage
vscanf(fmt2, va_list4);    // ambiguous usage
```

When processing the last two calls, the compiler cannot deduce any usage information from either the format strings or the arguments. In these instances, the `-msmart-io-format` option can be used and will potentially allow more optimal formatted IO functions to be generated, thus reducing the program's code size. For example, if the format strings pointed to by `fmt1` and `fmt2` collectively use only the `"%d"`, `"%i"` and `"%s"` conversion specifiers, the `-msmart-io-format="%d%i%s"` option should be issued.

The *`fmt`* string may contain any valid printf-style conversion specification, including flags and modifiers (for example, `"%-13.9ls"`), and should reflect exactly those conversion specifications used by the functions whose usage is ambiguous. Failure to include a specification in the *`fmt`* argument where it has been used by the formatted IO functions might result in code failure.

If *`fmt`* is an empty string or contains no discernible conversion specifications, an error will be issued. Characters other than those in valid conversion specifications are permitted but will be ignored.

This option may be used multiple times on the command line. The conversion specifications used with each option are accumulated.

3.6.1.16 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 (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.

3.6.1.17 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.

The `-mno-strict-X` form of this option allows the compiler to use the X register for indexed addressing (X+const, X-const). This is the default action if no option is specified.

3.6.1.18 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. It can be disabled by using the `-mno-tiny-stack` form of this option.

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 3-7. Kind-of-Output Control Options

Option (links to explanatory section)	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 sources files that follow on the command line.

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 another language is specified with a `-x` option, or the `-x none` option, which turns off the language specification entirely for subsequent files. The allowable languages are tabulated below.

Table 3-8. 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 -mcpu=atmega3290p -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.

The displayed help might differ for different device families, which can be indicated using the `-mcpu` option.

For example:

```
xc8-cc -mcpu=atmega48pb --help
Usage: avr-gcc [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase
  --help                   Display this information
  --target-help            Display target specific command line options
  --help={common|optimizers|params|target|warnings|[^]{joined|separate|undocumented}}[,...]
                           Display specific types of command line options
  (Use '-v --help' to display command line options of sub-processes)
  --version                Display compiler version information
  -dumpspecs               Display all of the built in spec strings
  -dumpversion             Display the version of the compiler
  -dumpmachine             Display the compiler's target processor
  -print-search-dirs       Display the directories in the compiler's search path
  -print-libgcc-file-name  Display the name of the compiler's companion library
  -print-file-name=<lib>   Display the full path to library <lib>
  -print-prog-name=<prog>  Display the full path to compiler component <prog>
  -print-multiarch         Display the target's normalized GNU triplet, used as
                           a component in the library path
```

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 3-9. C Dialect Control Options

Option (links to explanatory section)	Controls
`-ansi`	Strict ANSI conformance.
`-aux-info` *filename*	The generation of function prototypes.
`-f[no-]common`	The placement of global variables defined without an initializer
`-f[no-]asm`	Keyword recognition.
`-f[no-]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.
`-std=`*standard*	The C language standard.

3.6.3.1 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 extension include 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 -mcpu=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 Common Option

The `-fcommon` option tells the compiler to allow merging of tentative definitions by the linker. This is the default action if no option is specified.

The definition for a file-scope object without a storage class specifier or initializer is known as a tentative definition in the C standard. Such definitions are treated as external references when the

`-fcommon` option is specified and will be placed in a common block. As such, if another compilation unit has a full definition for an object with the same name, the definitions are merged and storage allocated. If no full definition can be found, the linker will allocate unique memory for the object tentatively defined. Tentative definitions are thus distinct from declarations of a variable with the `extern` keyword, which never allocate storage.

In the following code example:

extra.c
```
int a = 42;  /* full definition */
```

main.c
```
int a;       /* tentative definition */
int main(void) {
  ...
}
```

The object `a` is defined in `extra.c` and tentatively defined in `main.c`. Such code will build when using the `-fcommon` option, since the linker will resolve the tentative definition for `a` with the full definition in `extra.c`.

The `-fno-common` form of this option inhibits the merging of tentative definitions by the linker, treating tentative definitions as a full definition if the end of the translation unit is reached and no definition has appeared with an initializer. Building the above code, for example, would result in a `multiple definition of 'a'` error, since the tentative definition and initialized definition would both attempt to allocate storage for the same object. If you are using this form of the option, the definition of `a` in `main.c` should be written `extern int a;` to allow the program to build.

3.6.3.4 Ext Option

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

Table 3-10. 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.

3.6.3.5 Asm Option

The `-fasm` option reserves the use of `asm`, `inline` and `typeof` as keywords, preventing them from being defined as identifiers. This is the default action if no option is specified.

The `-fno-asm` form of this option restricts the recognition of these keywords. You can, instead, use the keywords `__asm__`, `__inline__` and `__typeof__`, which have identical meanings.

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

3.6.3.6 No-builtin Option

The `-fbuiltin` option has the compiler produce specialized code that avoids a function call 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. This is the default behavior if no option is specified.

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

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_`. There is no `-fbuiltin-function` form of this option.

3.6.3.7 Signed-bitfields Option

The `-fsigned-bitfields` 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-bitfields` or the `-fno-unsigned-bitfields` 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.

3.6.3.8 Signed-char Option

The `-fsigned-char` option forces 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 `-funsigned-char` (or `-fno-signed-char` option) makes this type explicit.

The `-fsigned-char` (or `-fno-unsigned-char` option) makes it explicit that plain `char` types are to be treated as signed integers.

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.

3.6.3.9 Std Option

The `-std=`*`standard`* option specifies the C standard to which the compiler assumes source code will conform. The allowable standards are listed below.

Table 3-11. Acceptable Language Standards

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

3.6.3.10 Unsigned-bitfields Option

The `-funsigned-bitfields` 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-bitfields` or the `-fno-signed-bitfields` 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.

3.6.3.11 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 which 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. Most of these specific warning options also have a negative form, beginning with `-Wno-`, to turn off warnings; for example, `-Wno-implicit`. These will be shown in the form a `-W[no-]`*`warning`*. Some only have a negative form, which will be shown as a `-Wno-`*`warning`*.

The options shown in the tables below are the more commonly used warning options that control messages issued by the compile. In the (linked) sections that follow, the options that affect warnings generally are discussed.

Table 3-12. Warning and Error Options Implied By All Warnings

Option (links to explanatory section)	Controls
`-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.
`-f[no-]syntax-only`	Checking code for syntax errors only.
`-W[no-]`*`msg`*	Whether a message is enabled or disabled.
`-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.
`-W[no-]format`	Warnings from inappropriate `printf()` arguments.
`-W[no-]implicit`	Warnings implied by both `-Wimplicit-int` and `-Wimplicit-function-declaration`.
`-W[no-]implicit-function-declaration`	Warnings from use of undeclared function.
`-W[no-]implicit-int`	Warnings from declarations not specifying a type.
`-W[no-]main`	Warnings from unusual main definition.
`-W[no-]missing-braces`	Warnings from missing braces.
`-W[no-]multichar`	Warnings from multi-character constant.
`-W[no-]parentheses`	Warnings from missing precedence.
`-W[no-]return-type`	Warnings from missing return type.
`-W[no-]sequence-point`	Warnings from sequence point violations.
`-W[no-]switch`	Warnings from missing or extraneous case values.
`-W[no-]system-headers`	Warnings from code within system headers.
`-W[no-]trigraphs`	Warnings from use of trigraphs.
`-W[no-]uninitialized`	Warnings from use of uninitialized variables.
`-W[no-]unknown-pragmas`	Warnings from use of unknown pragma.
`-W[no-]unused`	Warnings from unused objects and constructs.
`-W[no-]unused-function`	Warnings from unused static function.
`-W[no-]unused-label`	Warnings from unused labels.
`-W[no-]unused-parameter`	Warnings from unused parameter.

.....continued	
Option (links to explanatory section)	Controls
-W[no-]unused-variable	Warnings from unused variable.
-W[no-]unused-value	Warnings from unused value.

Table 3-13. Warning Options Not Implied by All Warnings

Option	Controls
-W[no-]extra	The generation of additional warning messages.
-W[no-]aggregate-return	Warnings from aggregate objects being returned.
-W[no-]bad-function-cast	Warnings from functions cast to a non-matching type.
-W[no-]cast-qual	Warnings from discarded pointer qualifiers.
-W[no-]conversion	Warnings from implicit conversions that can alter values.
-W[no-]error	Generation of errors instead of warnings for dubious constructs.
-W[no-]inline	Warnings when functions cannot be in-lined.
-W[no-]larger-than=*len*	Warnings when defining large objects.
-W[no-]long-long	Warnings from use of `long long`.
-W[no-]missing-declarations	Warnings when functions are not declared.
-W[no-]missing-format-attribute	Warnings with missing format attributes.
-W[no-]missing-noreturn	Warnings from potential noreturn attribute omissions.
-W[no-]missing-prototypes	Warnings when functions are not declared with prototype.
-W[no-]nested-externs	Warnings from `extern` declarations.
-Wno-deprecated-declarations	Whether warnings are produced for deprecated declarations.
-W[no-]pointer-arith	Warnings when taking size of unsized types.
-W[no-]redundant-decls	Warnings from redundant declarations.
-W[no-]shadow	Warnings when local objects shadow other objects.
-W[no-]sign-compare	Warnings from signed comparisons.
-W[no-]strict-prototypes	Warnings from K&R function declarations.
-W[no-]traditional	Warnings from traditional differences.
-W[no-]undef	Warnings from undefined identifiers.
-W[no-]unreachable-code	Warnings from unreachable code.
-W[no-]write-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: Enable or Disable Message Option

The `-W`*`msg`* option enables warning and advisory messages. The *`msg`* argument can be the name of a message issued by the compiler (e.g. `no-implicit-int`). The special message argument `all` indicates that all messages should be enabled. This option has precedence over the `-w` option,

which disables all warning messages, so it can be used to re-enable selected messages when `-w` has been used.

The `-Wno-`*`msg`* form of the option disables the indicated message, using the same arguments as the enabling form of this option. Where the *`msg`* argument corresponds to a warning or advisory message, this message will never be issued by the compiler or assembler. If the argument corresponds to an error or is not recognized, the compiler will indicate via an error that this operation is not permitted.

The name of the message is printed in compiler warnings (as shown in the following example of a return-type warning).

```
main.c:12:1: warning: control reaches end of non-void function [-Wreturn-type]
```

3.6.4.5 W: Disable all Warnings Option

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

3.6.4.6 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.

3.6.4.7 Werror Option

The `-Werror` option can be used to promote compiler warnings into errors. As with any other error, these promoted messages will prevent the application from completing.

The `-Werror` form of this option promotes any compiler warning message issued into an error. This option might be useful in applications where warnings are not permitted for functional safety reasons.

The `-Werror=`*`string`* form of this option promotes warnings identified by the string into errors. For example, `-Werror=return-type` will promote the warning `control reaches end of non-void function [-Wreturn-type]` to an error. Only one string may be specified with this option; however, you may use this option as many times as required.

The `-Wno-error=`*`string`* form of this option ensure that the warnings identified are never promoted to an error, even if the `-Werror` form of this option is in effect.

3.6.4.8 Wextra 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 3-14. Debugging Options

Option (links to explanatory section)	Controls
`-g`	The type of debugging information generated.
`-mchp-stack-usage`	The generation of stack usage information and warnings.
`-mcodecov`	Instrumentation of the output to provide code coverage information.
`-Q`	Printing of diagnostics associated with each function as it is compiled and statistics about each pass on conclusion.
`-save-temps[=dir]`	Whether, and where, intermediate files should be kept after compilation.

3.6.5.1 G: Produce Debugging Information Option

The `-g`*`format`* 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 3-15. Supported Debugging File Formats

Format	Debugging file format
`-g`*`level`*	Amount of debugging information produced, where *`level`* is a value from 0 thru 3.
`-gdwarf-3`	ELF/DWARF release 3.

By default, the compiler produces DWARF release 2 files. The `-g`*`level`* form of this option can be used in addition to a `-g` option that specifies the debugging format.

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.

3.6.5.2 Stack Guidance Option

The `-mchp-stack-usage` option analyzes the program and reports on the estimated maximum depth of any stack used by a program. The option can only be enabled with a PRO license.

See Stack Guidance for more information on the stack guidance reports that are produced by the compiler.

3.6.5.3 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 Code Coverage for more information.

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

3.6.5.4 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.

3.6.5.5 Save-temps Option

The `-save-temps` option instructs the compiler to keep temporary files after compilation has finished. You might find the generated `.i` and `.s` temporary files particularly useful for troubleshooting and they are often used by the Microchip Support team when you enter a support ticket.

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

3.6.6 Options for Controlling Optimization

The options listed below control compiler optimizations and are described in following sections.

Table 3-16. General Optimization Options

Option (links to explanatory section)	License	Builds with
`-O0`	All	No optimizations (default).
`-O1` `-O`	All	Optimization level 1.
`-O2`	All	Optimization level 2.
`-O3`	PRO only	Speed-orientated Optimizations.
`-Og`	All	Better debugging.
`-Os`	PRO only	Size-orientated optimizations.
`-f[no-]data-sections`	All	Each data object assigned to its own section.
`-f[no-]fat-lto-objects`	PRO only	Both object code and an internal representation written to the object files.
`-f[no-]function-sections`	All	Each function assigned to its own section.
`-f[no-]inline-functions`	All	Integration of all simple functions into their callers.
`-f[no-]lto`	PRO only	The standard link-time optimizer.
`-flto-partition=algorithm`	PRO only	The specified algorithm to partition object files when running the link-time optimizer.

.....continued

Option (links to explanatory section)	License	Builds with
`-f[no-]omit-frame-pointer`	All	The stack pointer used in preference to the frame pointer.
`-mno-pa-on-file=`*`filename`*	PRO	Procedural abstraction disabled for the specified file.
`-mno-pa-on-function=`*`function`*	PRO	Procedural abstraction disabled for the specified function.
`-mno-pa-outline-calls`	PRO	Procedural abstraction disabled across function calls.
`-f[no-]section-anchors`	All	Static objects accessed relative to one symbol.
`-f[no-]unroll-[all-]loops`	All	Loops unrolled to improve execution speed.
`-f[no-]use-linker-plugin`	PRO only	A linker plugin with link-time optimizations.
`-f[no-]whole-program`	PRO only	The whole-program optimizations.
`-m[no-]gas-isr-prologues`	All	Optimized context switch code for small interrupt service routines.
`-mpa-callcost-shortcall`	PRO only	More aggressive procedural abstraction.
`-mpa-iterations=`*`n`*	PRO only	The specified number of procedural abstraction iterations.
`--nofallback`	All	Only the selected optimization level and with no license-imposed fall back to a lesser level.

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 Data-sections Option

The `-fdata-sections` option places each object in its own section to assist with garbage collection and potential code size reductions.

When enabled, this option has each object placed in its own section named after the object. When used in conjunction with garbage collection performed by the linker (enabled using the`-Wl,--gc-sections` driver option) the final output might be smaller. It can, however, negatively impact other code generation optimizations, so confirm whether this option is of benefit with each project.

The `-fno-data-sections` form of this option does not force each object to a unique section. This is the default action if no option is specified.

3.6.6.8 Fat-lto-objects Option

The `-ffat-lto-objects` option requests that the compiler generate fat object files, which contain both object code and GIMPLE (one of GCC's internal representations), written to unique ELF sections. Such objects files are useful for library code that could be linked with projects that do and do not use the standard link-time optimizer, controlled by the `-flto` option.

The `-fno-fat-lto-objects` form of this option, which is the default if no option is specified, suppresses the inclusion of the object code into object files, resulting in faster builds. However, such object files must always be linked using the standard link-time optimizer.

3.6.6.9 Function-sections Option

The `-ffunction-sections` option places each function in its own section to assist with garbage collection and potential code size reductions.

When enabled, this option has each function placed in its own section named after the function. When used in conjunction with garbage collection performed by the linker (enabled using the `-Wl,--gc-sections` driver option) the final output might be smaller. The more granular sections resulting from the use of this option might also help the linker do a better job at allocating related sections together, potentially saving additional code size when the `-mrelax` option is also used. The `-ffunction-sections` option can, however, hamper other code generation optimizations, so confirm whether this option is of benefit with each project.

Note also that if functions are removed, their debugging information will remain in the ELF file, potentially imperiling the debuggability of code. When displaying source code information, the debugger looks for associations between addresses and the functions whose code resides at those locations, and it might wrongly consider that an address belongs to a removed function. As removed functions are not allocated memory, their "assigned" address will not change from being 0, thus the debugger is more likely to show references to a removed function when interpreting addresses closer to 0.

The `-fno-function-sections` form of this option does not force each function into a unique section. This is the default action if no option is specified.

3.6.6.10 Inline-functions Option

The `-finline-functions` option considers all functions for inlining, even if they are not declared `inline`.

If all calls to a given function are inlined, and the function is declared `static`, then the function is normally not output as a stand-alone assembler routine.

This option is automatically enabled at optimization levels `-O3` and `-Os`.

The `-fno-inline-functions` form of this option will never inline function that have not been marked as `inline`.

3.6.6.11 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 -O3 -flto -mcpu=atmega3290p foo.c
xc8-cc -c -O3 -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
```

Link time optimizations do not require the presence of the whole program to operate. If the program does not require any symbols to be exported, it is possible to combine `-flto` with `-fwhole-program` to allow the interprocedural optimizers to use more aggressive assumptions which may lead to improved optimization opportunities.

The bytecode files are versioned and there is a strict version check, so bytecode files generated in one version of the XC8 compiler might not work with a different version.

The `-fno-lto` form of this option does not run the standard link-time optimizer. This is the default action if this option is not specified.

3.6.6.12 Lto-partition Option

The `-flto-partition=`*`algorithm`* option controls the algorithm used to partition object files when running the link-time optimizer. Multiple partitions can be optimized in parallel, which might reduce the memory and processing time required by the compiler.

The argument `none` disables partitioning entirely and executes the link-time optimization step directly from the whole program analysis (WPA) phase. This mode of operation will produce the most optimal results at the expense of larger compiler memory requirements and longer build times, although this is unlikely to be an issue with small programs.

The `one` algorithm argument specifies that link-time optimizations should be performed on the whole program as a single unit, rather than splitting it into smaller partitions. This can result in better optimization opportunities, especially for interprocedural analysis and transformations, but it also requires the most memory and computational time during the link phase.

Selecting the option argument to `1to1` splits the program so that each original compilation unit becomes a separate partition. This can reduce the computational memory and time requirements significantly but might also reduce the runtime performance benefits of the optimization.

The default argument is `balanced`, which specifies partitioning into a given number of parts of roughly the same size, based on the call graph. This algorithm tries to balance the trade-off between computational memory and time, as well as runtime performance.

3.6.6.13 Omit-frame-pointer Option

The `-fomit-frame-pointer` option instructs the compiler to directly use the stack pointer to access objects on the stack and, if possible, omit code that saves, initializes, and restores the frame register. It is enabled automatically at all non-zero optimization levels.

Negating the option, using `-fno-omit-frame-pointer`, might assist debugging optimized code; however, this option does not guarantee that the frame pointer will always be used.

3.6.6.14 No-pa-on-file Option

The `-mno-pa-on-file=`*`filename`* option disables procedural abstraction of the code contained in the specified object or library archive file. There is no warning if the specified file is not processed.

3.6.6.15 No-pa-on-function Option

The `-mno-pa-on-function=`*`function`* option disables procedural abstraction of the code contained in the specified function. There is no warning if the specified function does not exist.

3.6.6.16 No-pa-outline-calls Option

The `-mno-pa-outline-calls` option prevents procedural abstraction optimizations from outlining code that contains function calls.

The compiler performs various checks to ensure it is safe to outline code blocks when procedural abstraction optimizations are enabled. One such check is to ensure that any routines called in a block do not access objects on the stack, such as stack-based arguments or return values. If a call to such a routine is outlined, the additional call to the abstracted routine inserted by the compiler (which will alter the stack pointer when the return address is pushed) can in some situations result in the wrong stack data being read. The compiler will not outline code that contains calls to routines that access the stack, but will allow calls to be outlined at other times. The `-mno-pa-outline-calls` option disables procedural abstraction of code that contains calls, even when the compiler believes that such optimizations are safe.

3.6.6.17 Section-anchors Option

The `-fsection-anchors` option allows access of `static` objects to be performed relative to one symbol.

When enabled, this option might generate code that accesses multiple `static` objects as an offset from one base address, rather than accessing each object separately. Clearly, this optimization can only improve code size if more than one `static` object is defined in your program. Although this option is available for unlicensed compilers, it works best with level `s` optimizations (`-Os`), which are only available with a PRO license.

3.6.6.18 Unroll-loops/unroll-all-loops Options

The `-funroll-loops` and `-funroll-all-loops` options control speed-orientated optimizations that attempt to remove branching delays in loops. Unrolled loops typically increase the execution speed of the generated code, at the expense of larger code size.

The `-funroll-loops` option unrolls loops where the number of iterations can be determined at compile time or when code enters the loop. This option is enabled with `-fprofile-use`. The `-funroll-all-loops` option is more aggressive, unrolling all loops, even when the number of iterations is unknown. It is typically less effective at improving execution speed than the `-funroll-loops` option.

The `-fno-unroll-loops` and `-fno-unroll-all-loops` forms of these options do not unroll any loops and are the default actions if no options are specified.

3.6.6.19 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. This is the default action if no option is specified. The `-fwhole-program` option should not be used with the `-fuse-linker-plugin` option.

The `-fno-use-linker-plugin` option disables the use of a linker plugin during link-time optimization.

3.6.6.20 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.

The `-fno-whole-program` form of this option disables these optimizations and this is the default action if no option is specified.

3.6.6.21 ISR Prologues Option

The `-m[no-]gas-isr-prologues` option controls the context switch code generated for small interrupt service routines (ISRs).

If this feature is not enabled, the compiler will save registers R0, R1, and SREG, and clear register R1 in the context save code associated with ISRs and restore these registers in the context restore code. When enabled, this feature will have the assembler scan the ISR for register usage and only save these registers if required. For small ISRs, the result will be smaller and faster executing interrupt code. This feature can be safely used with other compiler features, such as code coverage, stack guidance, and procedural abstraction, which will continue to work as expected.

This feature is automatically enabled at level 1 and higher optimizations (except when the `-Og` option is enabled). It can be disabled for the entire program by using the `-mno-gas-isr-prologues` option; alternatively, it can be disabled for particular ISRs by using the `no_gccisr` function attribute with that ISR definition.

3.6.6.22 Pa-callcost-shortcall Option

The `-mpa-callcost-shortcall` option performs more aggressive procedural abstraction, in the hope that the linker can relax long calls and reduce code size. This option can, however, increase code size if the underlying assumptions are not realized, so monitor if it is of benefit to each project.

3.6.6.23 Pa-iterations Option

The `-mpa-iterations=`*n* option allows the number of procedural abstraction iterations to be changed from the default of 2. More iterations might reduce code size, but could impact the code build times.

3.6.6.24 Nofallback Option

The `--nofallback` option can be used to ensure that the compiler is not inadvertently executed with optimizations below the that specified by the `-O` option.

For example, if an unlicensed compiler was requested to run with level `s` optimizations, without this option, it would normally revert to a lower optimization level and proceed. With this option, the compiler will instead issue an error and compilation will terminate. Thus, this option can ensure that builds are performed with a properly licensed compiler.

3.6.7 Options for Controlling the Preprocessor

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

Table 3-17. Preprocessor Options

Option (links to explanatory section)	Controls
`-C`	Preserve comments
`-dletters`	Preserve macro definitions
`-Dmacro` `-Dmacro=defn`	Define a macro
`-fno-show-column`	Omit column numbers in diagnostics
`-H`	Print header file name
`-imacros file`	Include file macro definitions only
`-include file`	Include file
`-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
`-P`	Don't generate #line directives
`-trigraphs`	Support trigraphs
`-Umacro`	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 `-dletters` 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 `-dM` option generates a list of `#define` directives for all the macros defined during the execution of the preprocessor, including predefined macros. These will indicate the replacement string if one was defined. For example, the output might include:

```
#define _CP0_BCS_TAGLO(c,s) _bcsc0(_CP0_TAGLO, _CP0_TAGLO_SELECT, c, s)
#define PORTE PORTE
#define _LATE_LATE1_LENGTH 0x00000001
#define _IPC22_w_POSITION 0x00000000
```

You can use this option to show those macros which are predefined by the compiler or header files.

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

Table 3-18. Preprocessor Debugging Information

Letter	Produces
`D`	Similar output to `-dM` but lacking predefined macros. The normal preprocessor output is also included in this output.
`M`	Full macro output, as described in the main text before this table.
`N`	Similar output to `-dD` but only the macro names are output with each definition.

3.6.7.3 D: Define a Macro

The `-Dmacro` option allows you to define a preprocessor macro and the `-Dmacro=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 `-Dmacro` 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 `-Dmacro=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 Preprocessor Arithmetic for clarification of how the replacement text might be used.

Defining macros as C string literals requires escaping the quote characters (`" "`) used by the string. If a quote is intended to be included and passed to the compiler, it should be escaped with a backslash character (`\`). If a string includes a space character, the string should have additional quotes around it.

For example, to pass the C string, `"hello world"`, (including the quote characters and the space in the replacement text), use `-DMY_STRING="\"hello world\""`. You could also place quotes around the entire option: `"-DMY_STRING=\"hello world\""`. These formats can be used on any platform. Escaping the space character, as in `-DMY_STRING=\"hello\ world\"` is only permitted with macOS and Linux systems and will not work under Windows, and hence it is recommended that the entire option be quoted to ensure portability.

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 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`.

3.6.7.8 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.

3.6.7.9 MF: Specify Dependency File Option

The `-MF` *`file`* option specifies a file in which to write the dependencies for the `-M` or `-MM` options. If no `-MF` option is given, the preprocessor sends the rules to the same place it would have sent preprocessed output.

When used with the driver options, `-MD` or `-MMD`, `-MF`, overrides the default dependency output file.

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 MP 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 P: Don't Generate #line Directives Option

The `-P` option tells the preprocessor not to generate `#line` directives in the preprocessed output. Used with the `-E` option.

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 3-19. Assembly Options

Option (links to explanatory section)	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 pass the `-a` option to the assembler, requesting that an assembly list file be produced.

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 or that can't be parsed by the `-Wa` option.

3.6.9 Mapped Assembler Options

The option tabulated below is a commonly used assembler option.

Table 3-20. Mapped Assembler Options

Option	Controls
`-Wa,-a=file.lst`	The generation of a named assembly list file.

3.6.10 Options for Linking

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

Table 3-21. Linking Options

Option (links to explanatory section)	Controls
`-mno-data-init`	Suppression of the `.dinit` section and data initialization.
`-llibrary`	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.
`-T`	The linker script to use with the build.
`-u`	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 No-data-init Option

The `-mno-data-init` driver option attempts to exclude the data initialization code performed at runtime startup.

When used, this option prevents the compiler from emitting special symbol references that force the data initialization code to be linked in to the project. Thus, this option will typically decrease the size of projects, as well as increase their startup speed; however, objects in data memory will not be cleared or initialized. Note that any library code linked into the project that references these special symbols will force the inclusion of the usual data initialization code, even when this option is used.

3.6.10.2 L: Specify Library File Option

The `-llibrary` option scans the named library file 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.

By default the linker is directed to search `<install-path>/avr/lib` for libraries specified with the `-l` option. This behavior can be overridden using environment variables.

See also the `INPUT` and `OPTIONAL` linker script directives.

3.6.10.3 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.

The compiler may generate calls to `memcmp`, `memset` and `memcpy`, even when this option is specified. As these symbols are usually resolved by entries in the standard compiler libraries, they should be supplied through some other mechanism when this option is specified.

3.6.10.4 Nostartfiles Option

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

3.6.10.5 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.

3.6.10.6 S: Remove Symbol Information Option

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

3.6.10.7 T: Use Linker Script Option

The `-T` *`script`* option specifies a linker script to use when linking.

Ordinarily, the linker will use a default linker script obtained from the compiler installation. The `-T` option allows you to specify an application-specific linker script to use. The MPLAB X IDE uses this option when you add a linker script to your project.

3.6.10.8 U: Add Undefined Symbol Option

The `-u` *`symbol`* option adds an undefined symbol that will be present at the link stage. To resolve the symbol, the linker will search library modules for its definition, thus this option is useful if you want to force a library module to be linked in. It is legitimate to use this option multiple times with different symbols to force loading of additional library modules.

3.6.10.9 Wl: Pass Option To The Linker, Option

The `-Wl,`*`option`* option passes *`option`* to the linker application where it will be interpreted as a linker option. If *`option`* contains commas, it is split into multiple options at the commas. Any linker option specified will be added to the default linker options passed by the driver and these will be executed before the default options by the linker.

3.6.10.10 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 listed below are commonly used linker options.

Table 3-22. Mapped Linker Options

Option	Controls
`-Wl,--gc-sections`	Reclamation of unused memory through garbage collection.
`-Wl,-Map=`*`mapfile`*	The generation of a linker map file.
`-Wl,--section-start=`*`section`*`=`*`addr`*	Placement of custom-named sections at the specified address. It cannot be used to place standard sections, like (`.data`, `.bss`, `.text`), which must be placed using a `-Wl,-T` option.
`-Wl,-T`*`section`*`=`*`address`*	The address allocated to the standard (`.data`, `.bss`, `.text`) sections (see Changing and Linking the Allocated Section).

3.6.12 Options for Directory Search

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

Table 3-23. Directory Search Options

Option (links to explanatory section)	Controls
`-idirafter` *`dir`*	Additional directories searched for headers after searching system paths
`-I`*`dir`*	The directories searched for preprocessor include files
`-iquote`	Specify quoted include file search path
`-L`*`dir`*	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`* list of directories to be searched for header files during preprocessing. The directory is searched only after all other search paths (including the standard search directories as well as those specified by the `-I` and `-Iquote` options) have been searched. If this option is used more than once, the directories they specify are searched in a left-to-right order as they appear on the command line.

3.6.12.2 I: Specify Include File Search Path Option

The `-I`*`dir`* 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.

Note: Do not use this option to specify any MPLAB XC8 system include paths. The compiler drivers automatically select the default language libraries and their respective include-file directory for you. Manually adding a system include path may disrupt this mechanism and cause the incorrect files to be compiled into your project, causing a conflict between the include files and the library. Note that adding a system include path to your project properties has never been a recommended practice.

3.6.12.3 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"`, and the directory is searched only after the current

working directory has been searched. If this option is used more than once, the directories they specify are searched in a left-to-right order as they appear on the command line.

3.6.12.4 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.

3.6.12.5 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 3-24. Code Generation Convention Options

Option (links to explanatory section)	Controls
`-f[no-]short-enums`	The size of `enum` types

3.6.13.1 Short-enums Option

The `-fshort-enums` option allocates the smallest possible integer type (with a size of 1, 2, or 4 bytes) to an `enum` such that the range of possible values can be held.

The `-fno-short-enums` form of this option forces each `enum` type to be 2-bytes wide (the size of the `int` type). This is the default action if no option is specified. When using this option, generated code 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

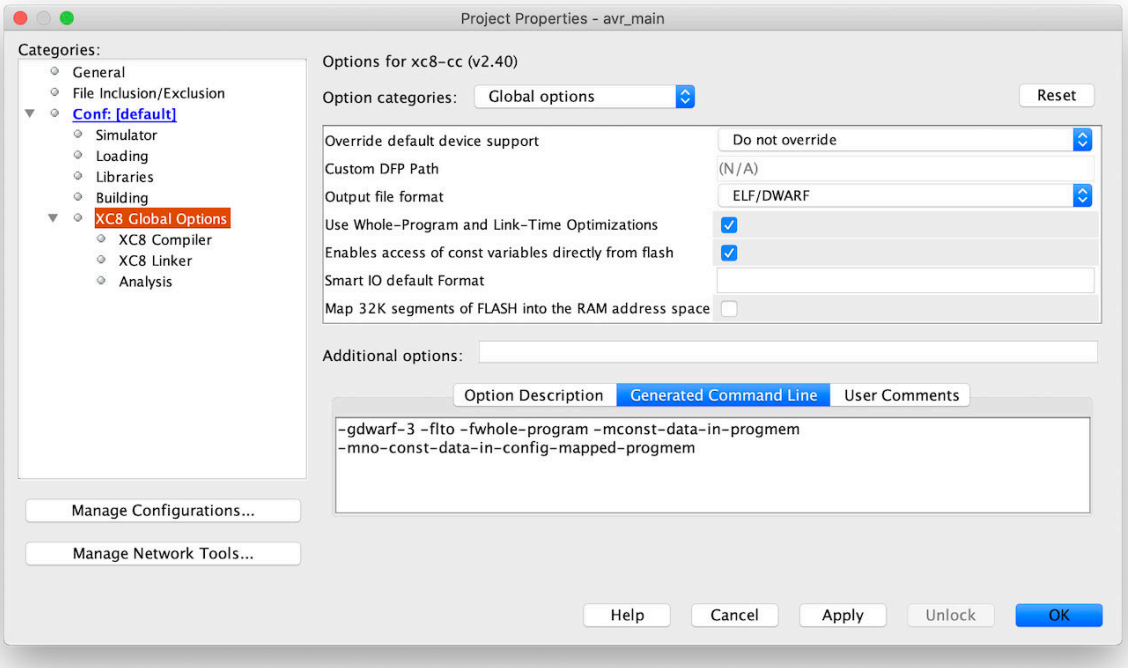
The following descriptions map the MPLAB X IDE's **Project Properties** controls to the MPLAB XC8 command-line driver options. Reference is given to the relevant section in the user's guide to learn more about the option's function. In the IDE, click any option to see online help and examples shown in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

3.7.1.1 XC8 Global Options Category

The following sections describe the options available in the XC8 Global Options category.

3.7.1.1.1 XC8 Global Options - Global Options

Figure 3-2. XC8 Global Options - Global Options



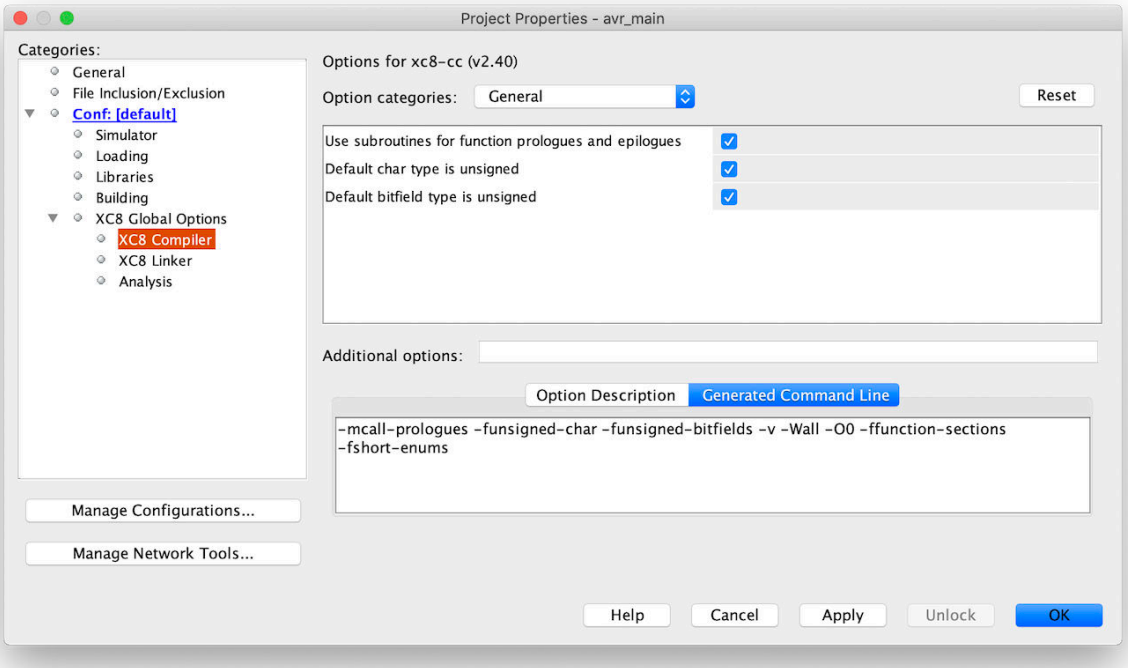
Override default device support	This selector allows you to indicate how Device Family Pack (DFP) management should be performed. The **Do not override** selection will let the MPLAB X IDE provide a list of DFPs that can be selected. If you would like to use a DFP that you have manually downloaded, select **User specified location** and then enter the path to the DFP in the **Custom DFP path** field. You may also select **Compiler location**, which will use the DFPs that ship with the compiler rather than the IDE.
Custom DFP path	If you have selected **User specified location** for the **Override default device support** option, enter the path to the DFP you wish to use in this field.
Output file format	This selector specifies the output file type. See G: Produce Debugging Information Option.
Use Whole-program and Link-Time Optimizations	Selecting this checkbox will run more aggressive optimizations over the whole combined compilation unit. See Lto Option and Whole-program Optimizations Option.
Enables access of const variables directly from flash	Selecting this checkbox enables the const-in-progmem feature, which places `const`-qualified objects into program memory space for some devices. See Const-data-in-progmem Option.
Smart IO default Format	This field allows you to fine-tune the specifications that must be processed by smart IO functions when the compiler is unable to determine usage information from the formatted IO function cal. See Smart-io-format Option
Map 32K segments of FLASH into the RAM address space	Selecting this checkbox will place `const`-qualified objects into a segment of Flash memory that is mapped into the data space. See Const-data-in-config-mapped-progmem Option.

3.7.1.2 XC8 Compiler Category

The follow sections describe the options available in the XC8 Compiler category.

3.7.1.2.1 XC8 Compiler - General Options

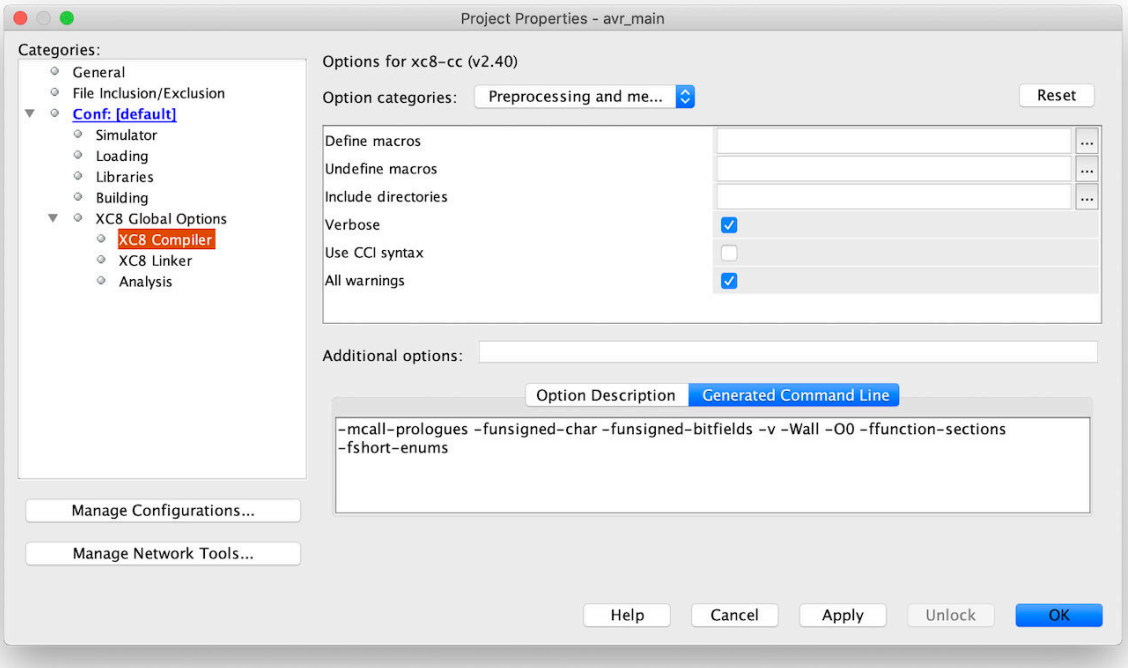
Figure 3-3. XC8 Compiler - General Options



Use subroutines for function prologues and epilogues	This selection allows you to change how functions save registers on entry and how those registers are restored on function exit. See Call-prologues Option.
Default char type is unsigned	Enabling this checkbox indicates that a plain `char` type will be treated as an `unsigned char`. See Signed-char Option and 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 Signed-bitfields Option and Unsigned-bitfields Option.

3.7.1.2.2 XC8 Compiler - Preprocessing and Messaging Options

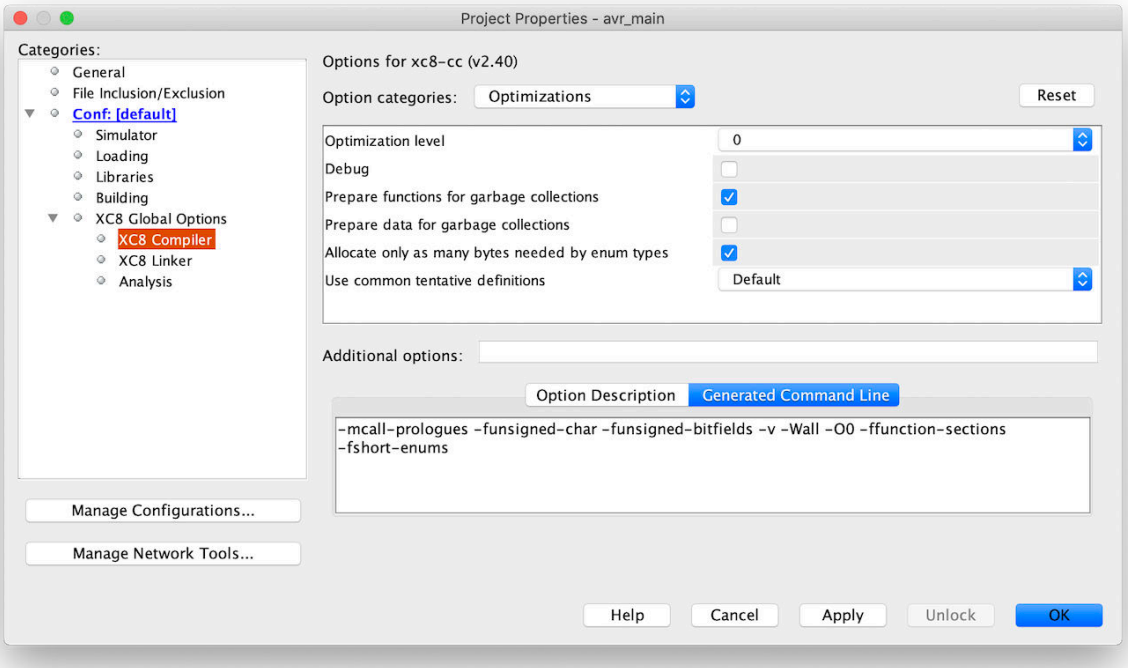
Figure 3-4. XC8 Compiler - Preprocessing and Messaging Options



Define macros	This field allows you to define preprocessor macros. See D: Define a Macro.
Undefine macros	This field allows you to undefine preprocessor macros. See U: Undefine Macros.
Include directories	This field allows you to specify the directories searched for header files. See I: Specify Include File Search Path Option.
Verbose	Enabling this checkbox shows the command lines used to build the project. See V: Verbose Compilation.
Use CCI syntax	Enabling this checkbox requests that the CCI language extension be enforced. See Ext Option.
All warnings	Enabling this checkbox requests that no compiler warning messages will be suppressed when building. See Wall Option.

3.7.1.2.3 XC8 Compiler - Optimizations options

Figure 3-5. XC8 Compiler - Optimization Options



Note that some of the compiler options specified by fields in other Project Property Categories can affect the size and execution speed of your project. Consider using the Compiler Advisor, accessible via the MPLAB X IDE **Tools** > **Analysis** > **Compiler Advisor** menu item, to compare the size of your project when built with different combination of compiler options.

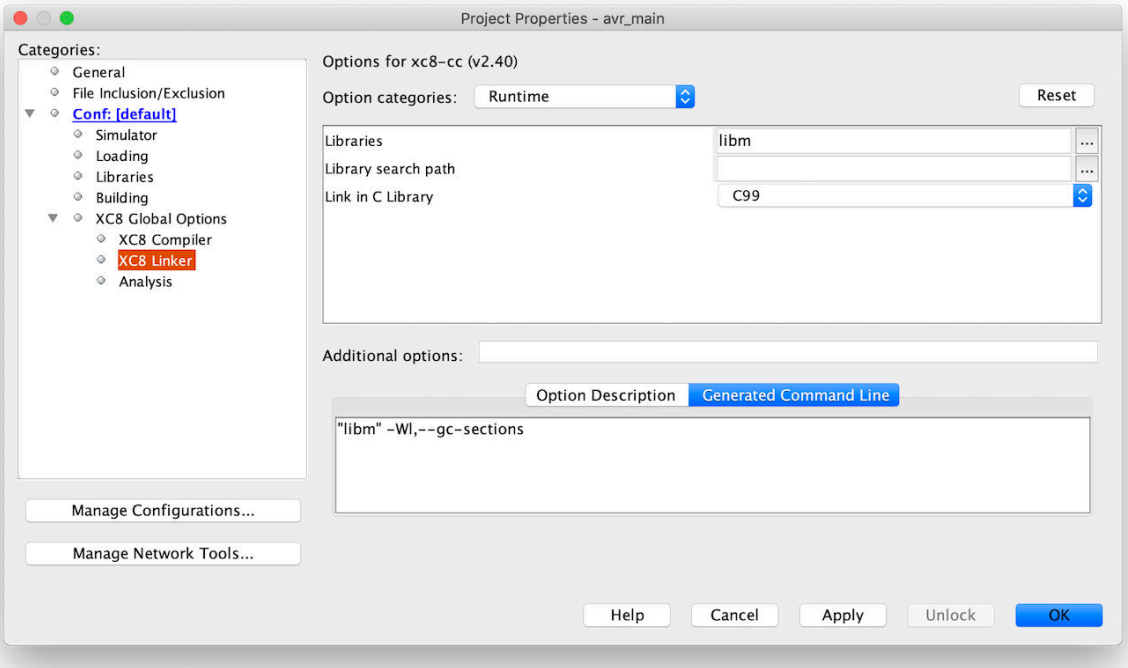
Optimization level	This selector controls the level to which programs are optimized. See Options for Controlling Optimization.
Debug	This checkbox inhibits aggressive optimization that can impact on the debugability of code. See Og: Better Debugging Option.
Prepare functions for garbage collection	Selecting this checkbox places each function in its own section to assist with garbage collection and potential code size reductions. See Function-sections Option.
Prepare data for garbage collection	Selecting this checkbox places each object in its own section to assist with garbage collection and potential code size reductions. See Data-sections Option.
Allocate only as many bytes as needed by enum types	Selecting this checkbox allocates the smallest possible integer type to an `enum` such that the range of possible values can be held. See Short-enums Option.
Use common tentative definitions	Selecting Enable from this drop-down menu allows the linker to resolve all tentative definitions of the same variable in different compilation units to the same object. Selecting Disable inhibits the merging of tentative definitions by the linker. Selecting Default enables tentative definition resolution. See Common Option.

3.7.1.3 XC8 Linker Category

The follow sections describe the options available in the XC8 Linker category.

3.7.1.3.1 XC8 Linker - Runtime Options

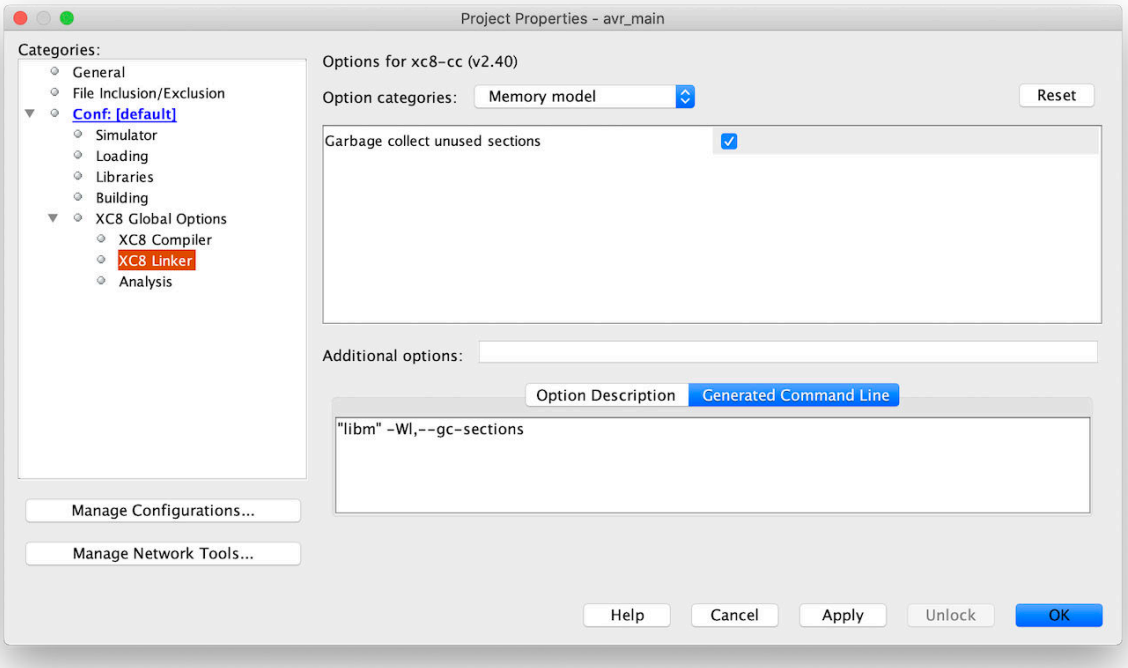
Figure 3-6. XC8 Linker - Runtime Options



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 L: Specify Library Search Path Option.
Link in C library	This selector specifies whether the standard libraries will be linked. See Nodefaultlibs Option.

3.7.1.3.2 XC8 Linker - Memory Model Options

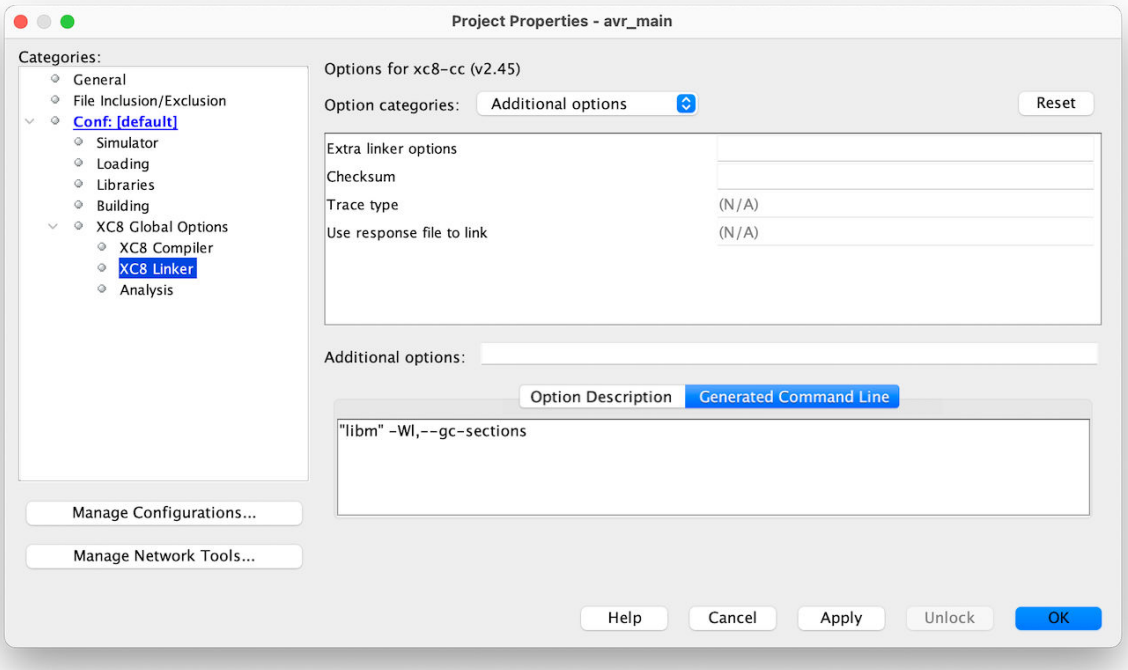
Figure 3-7. XC8 Linker - Memory model options



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

3.7.1.3.3 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 Wl: Pass Option To The Linker, Option.

Checksum If required, enter an argument in this field to generate a hash value over data in the HEX file. The argument mimics that used by the `-mchecksum` option used by MPLAB XC8 for PIC devices, specifically:

```
start-end@destination[,specifications]
```

which calculates a hash from *start* to *end* addresses, placing the result at *destination*, and the specifications are as follows:

- `width=`*n* selects the width of the hash result in bytes. A negative width will store the result in little-endian byte order.
- `offset=`*nnnn* specifies an initial value or offset to be added to the hash.
- `algorithm=`*n* selects the hash algorithm to use.
- `code=`*nn*`.`*Base* is a hexadecimal code that will trail each byte in the hash result. This can allow each byte of the hash result to be embedded within an instruction.
- `polynomial=`*nn* hexadecimal value which is the polynomial to be used for CRC hashes.
- `revword=`*n* reads the data in reserve order with the word width specified.
- `skip=`*n*`.`*Bytes* skips the MSB in the word width specified.
- `xor=`*n* is a hexadecimal value that will be XORed with the hash result before it stored.

Check the *MPLAB® XC8 C Compiler User's Guide for PIC® MCU* for full information regarding this option. Note that this field does not map to any compiler option, but will instead trigger a post-build execution of the Hexmate utility.

Trace type This selector is not yet implemented.

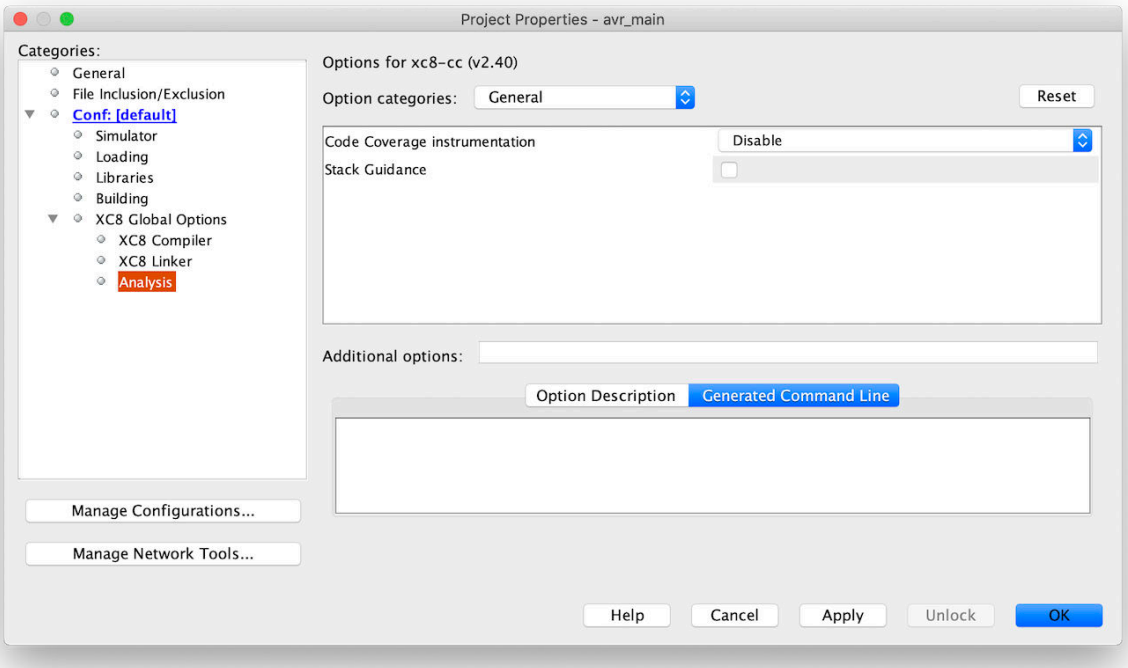
Use response file to link This field allows a command-line options file to be used by the compiler during the link step, in preference to the other link-step settings in the project properties. See Long Command Lines. This option is only relevant when running MPLAB X IDE under Windows.

3.7.1.4 XC8 Analysis Category

The follow sections describe the options available in the XC8 Analysis category.

3.7.1.4.1 XC8 Analysis - General

Figure 3-9. XC8 Analysis - General options



Code Coverage instrumentation This selector controls whether the program output will be instrumented with assembler sequences that record execution of the code they represent and whose data can facilitate analysis of the extent to which a project's source code has been executed. See Codecov Option and Code Coverage.

Stack Guidance This checkbox enables the compiler's stack guidance feature for licensed PRO compilers. The feature estimates the maximum depth of any stack used by a program and prints a report. See Stack Guidance Option and Stack Guidance.

3.8 Microchip Studio Integration

The 8-bit language tools may be integrated into and controlled from Microchip Studio for AVR® and SAM Devices, to provide a GUI-based development of application code for the 8-bit AVR MCU families of devices.

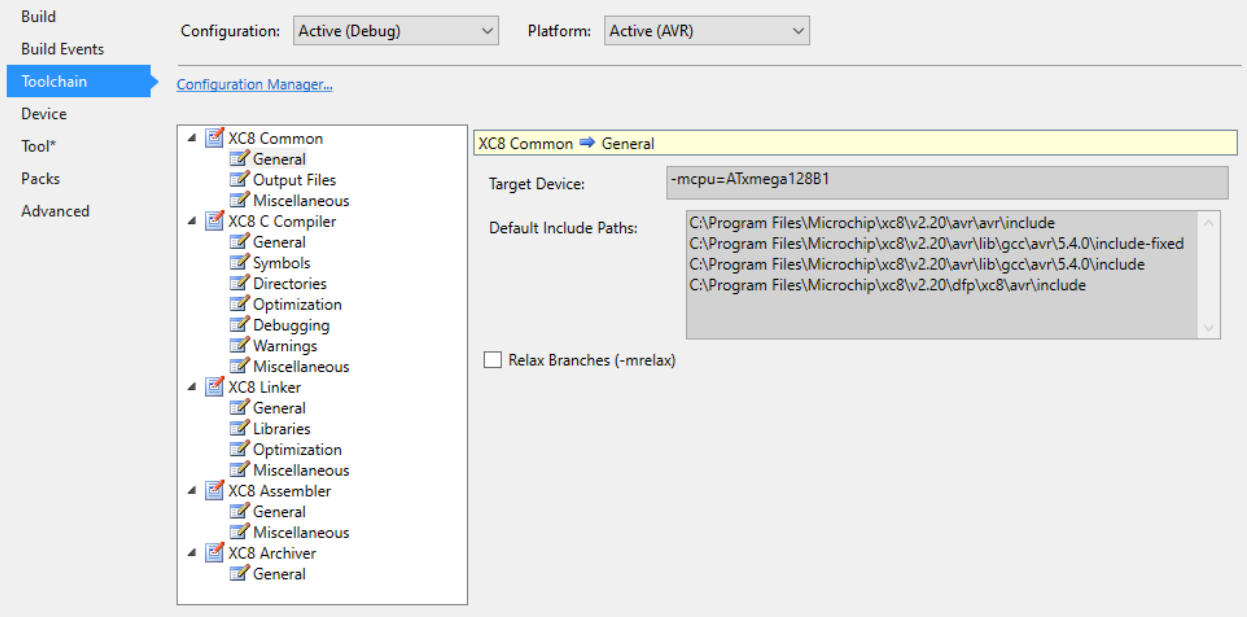
For installation of the IDE, and the creation and setup of projects to use the MPLAB XC8 C Compiler, see the *Microchip Studio 7 User Guide* (DS50002718).

3.8.1 Microchip Studio Option Equivalents

The following descriptions map Microchip Studio's **Toolchain** 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.

3.8.1.1 XC8 Common - General

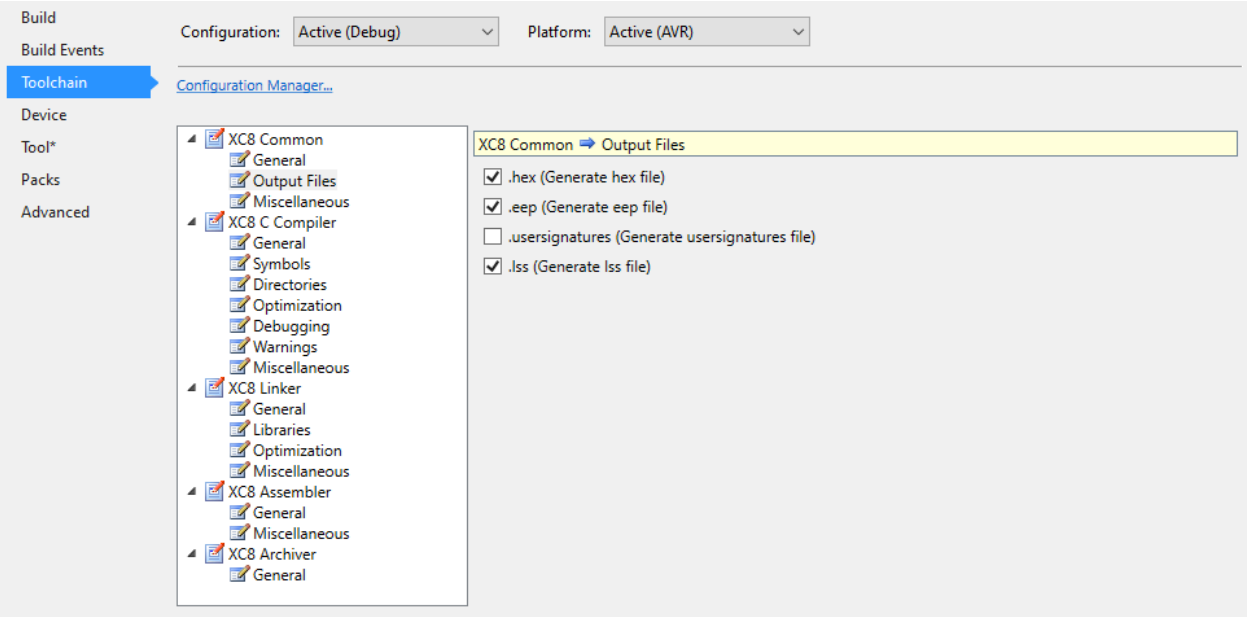
Figure 3-10. XC8 Common - General



Relax Branches Selecting this checkbox enables the optimization of the long form of call and jump instructions. See Relax Option.

3.8.1.2 XC8 Common - Output Files

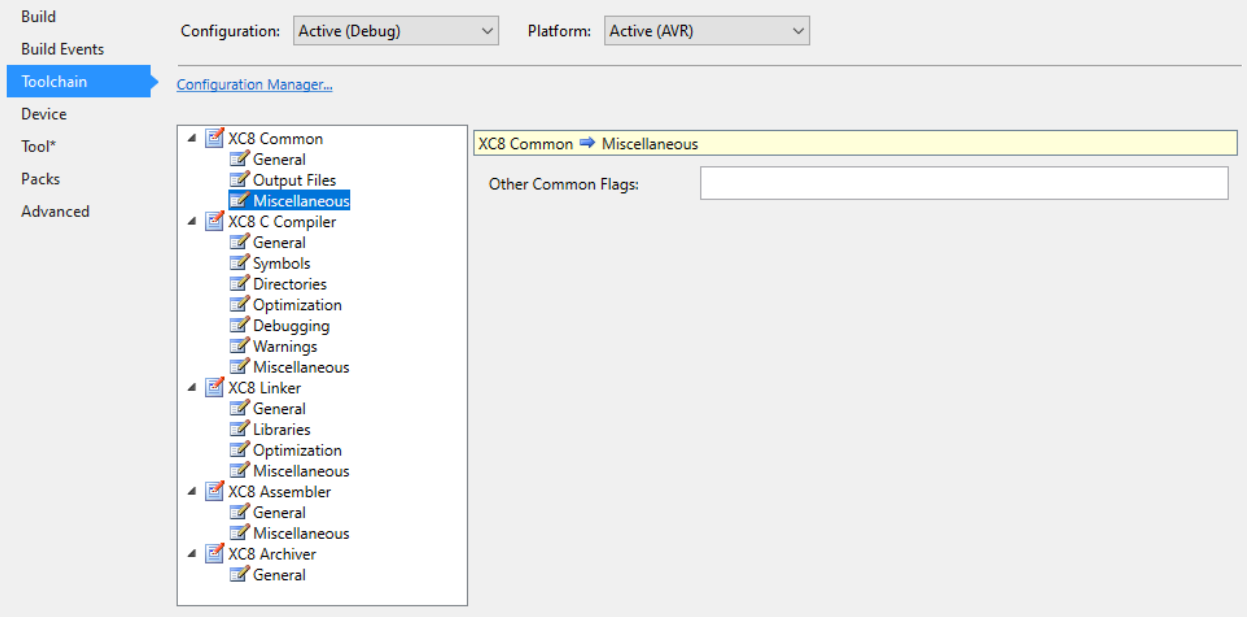
Figure 3-11. XC8 Common - Output Files



Generate hex file	When this checkbox is selected, the `avr-objcopy` application will be invoked after linking to generate a HEX file in addition to the ELF file.
Generate eep file	When this checkbox is selected, the `avr-objcopy` application will be invoked after linking to have `.eeprom` sections extracted from the ELF file and stored to a `.eep` file.
Generate usersignatures file	When this checkbox is selected, the `avr-objcopy` application will be invoked after linking to have `.user_signatures` sections extracted from the ELF file and stored to a `.usersignatures` file.
Generate lss file	When this checkbox is selected, the `avr-objdump` application will be invoked after linking to generate an assembly list file (`.lss` extension).

3.8.1.3 XC8 Common - Miscellaneous

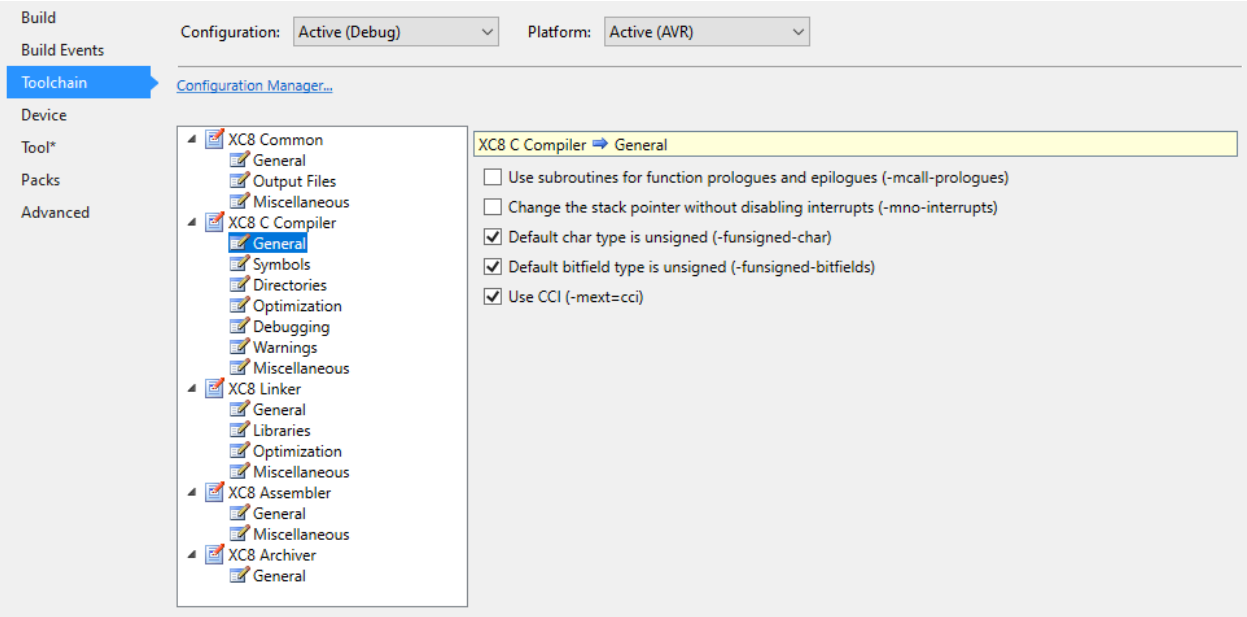
Figure 3-12. XC8 Common - Miscellaneous



Other Common Flags	Enter additional options in this field to affect all stages of compilation.

3.8.1.4 XC8 C Compiler - General

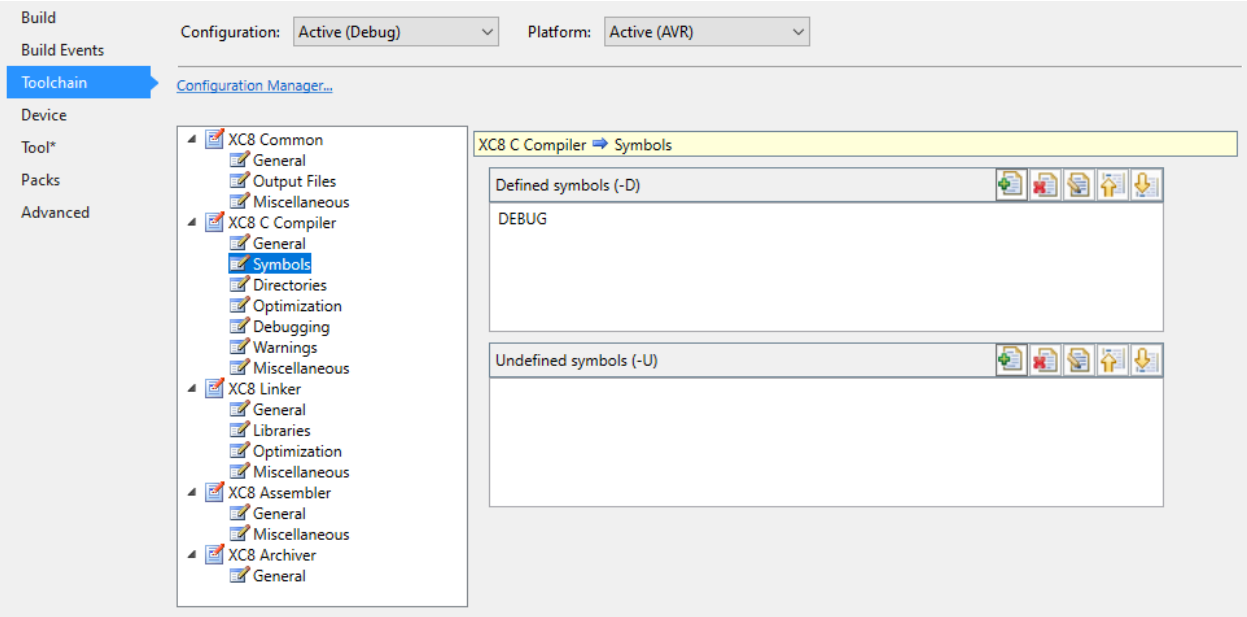
Figure 3-13. XC8 C Compiler - General



Use subroutines for function prologues and epilogues	This option affects how functions save and restore registers on entry and exit. See Call-prologues Option.
Change the stack pointer without disabling interrupts	This option controls whether interrupts should be disabled when the stack pointer is changed. See No-interrupts Option.
Default char type is unsigned	This option controls the signedness used for a plain `char`. See Signed-char Option.
Default bitfield type is unsigned	This option controls the signedness used for a plain `int` bit-field. See Signed-char Option.
Use CCI	Selecting this option indicates that source code should conform to the Common C Interface. See Ext Option.

3.8.1.5 XC8 C Compiler - Symbols

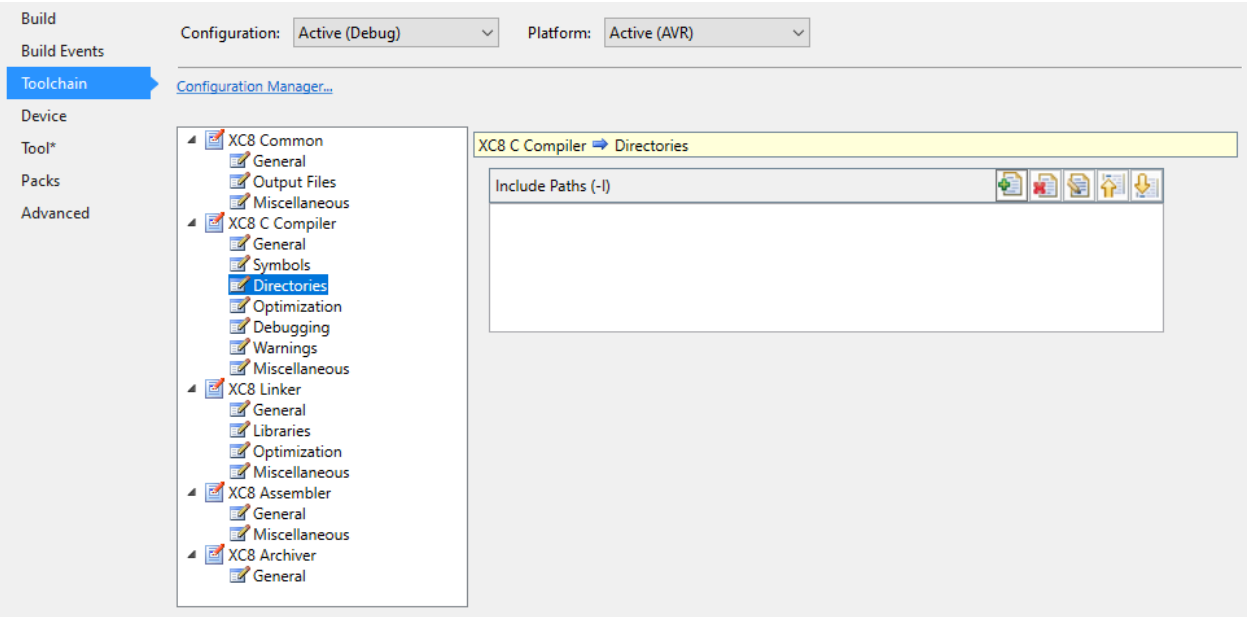
Figure 3-14. XC8 C Compiler - Symbols



Defined symbols	These controls allow you to define preprocessor macro symbols and values. See D: Define a Macro.
Undefined symbols	These controls allow you to undefine preprocessor macro symbols and values. See U: Undefine Macros.

3.8.1.6 XC8 C Compiler - Directories

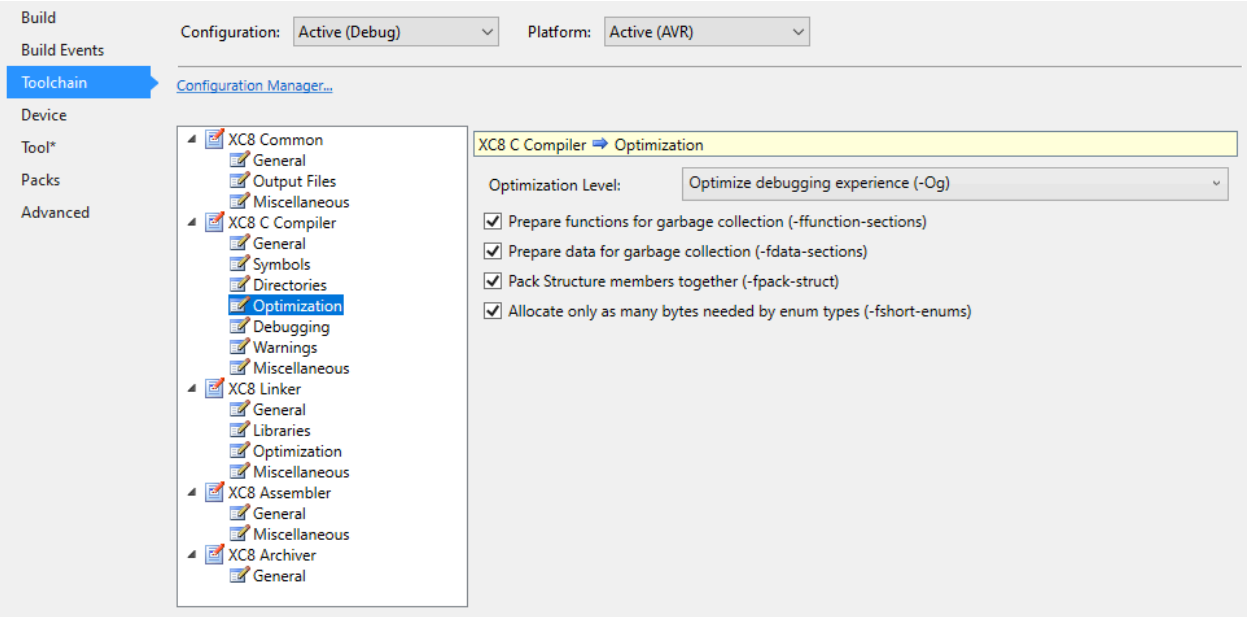
Figure 3-15. XC8 C Compiler - Directories



Include paths	These controls allow you to specify search paths for include files. See I: Specify Include File Search Path Option.

3.8.1.7 XC8 C Compiler - Optimization

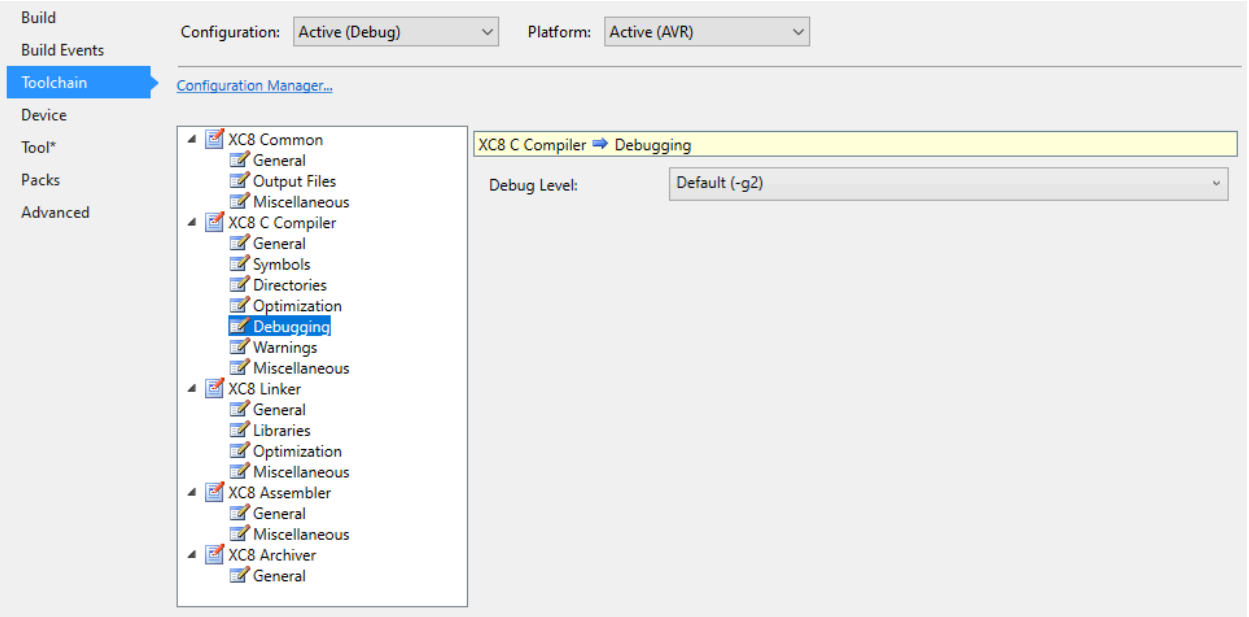
Figure 3-16. XC8 C Compiler - Optimization



Optimization level	This selector allows you to choose the degree and types of optimization performed. See, for example, O0: Level 0 Optimizations.
Prepare functions for garbage collection	Selecting this checkbox makes function memory allocation suitable for garbage collection. See Function-sections Option.
Prepare data for garbage collection	Selecting this checkbox makes object memory allocation suitable for garbage collection. See Data-sections Option.
Pack structure members together	This option has no effect for 8-bit AVR devices.
Allocate only as many bytes as needed by enum types	Enabling this option allocates the smallest possible integer type to an `enum`. See Short-enums Option.

3.8.1.8 XC8 C Compiler - Debugging

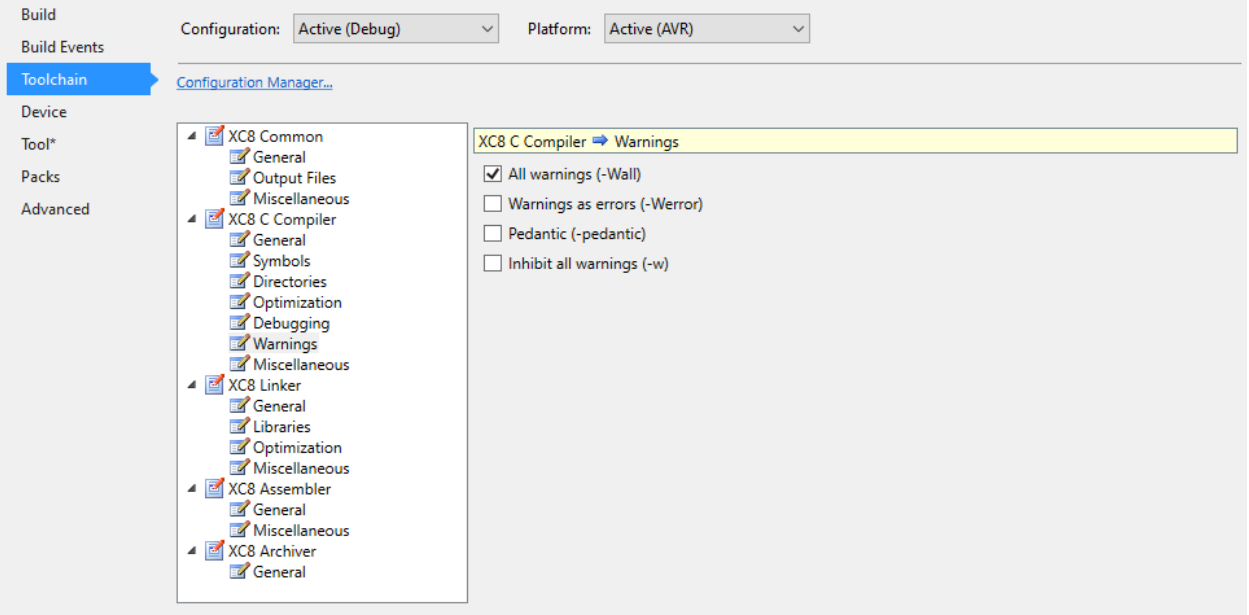
Figure 3-17. XC8 C Compiler - Debugging



Debug level This selector controls the amount of debug information produced. See G: Produce Debugging Information Option.

3.8.1.9 XC8 C Compiler - Warnings

Figure 3-18. XC8 C Compiler - Warnings



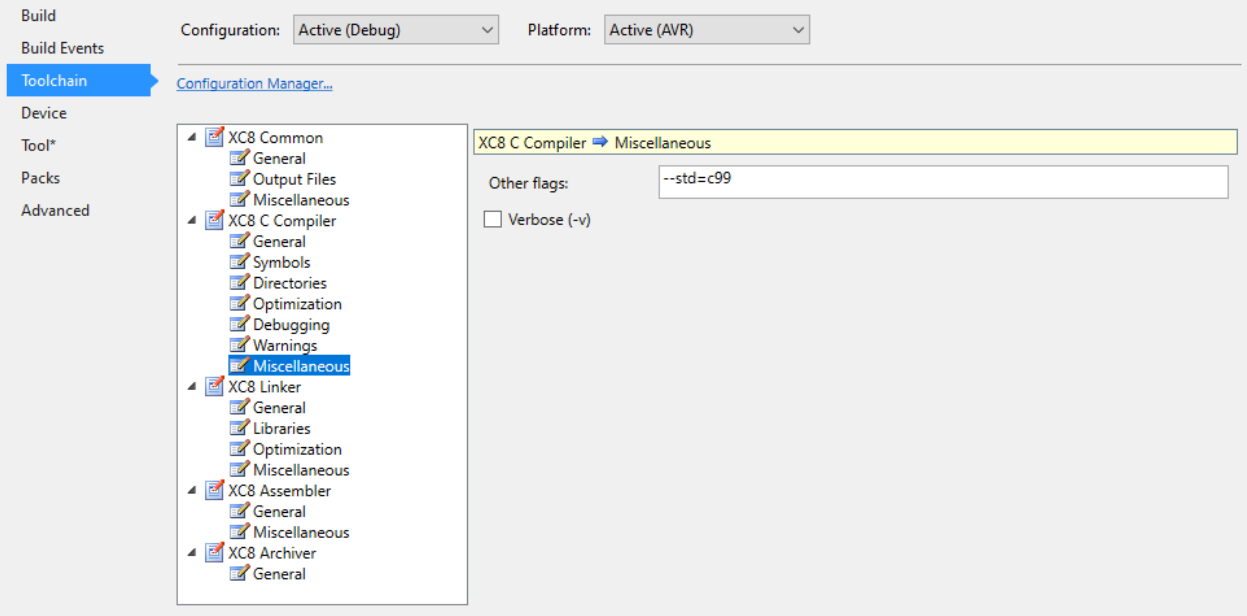
All warnings Selecting this checkbox enables warnings for all questionable code constructs. See Wall Option.

Warnings as errors Selecting this checkbox will have warnings messages treated like errors that will prevent compilation from completing. See Werror Option.

Pedantic	Selecting this checkbox will ensure that programs do not use forbidden extensions and that warnings are issued when a program does not follow ISO C. See Pedantic Option.
Inhibit all warnings	Selecting this checkbox will inhibits all warning messages. See W: Disable all Warnings Option.

3.8.1.10 XC8 C Compiler - Miscellaneous

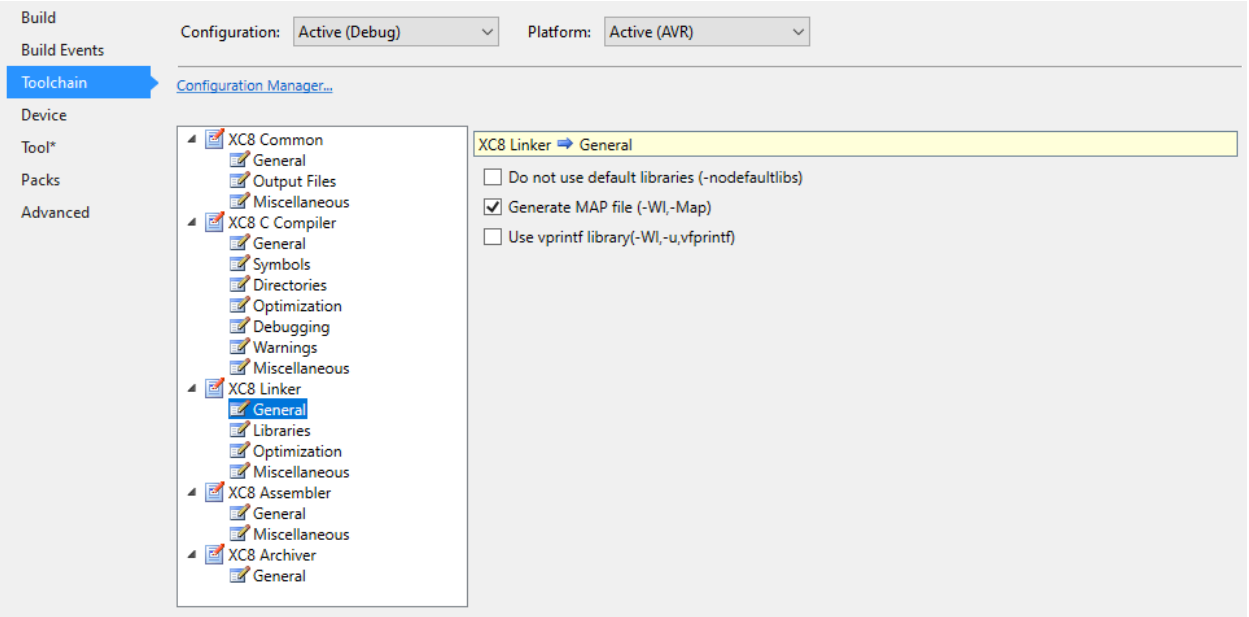
Figure 3-19. XC8 C Compiler - Miscellaneous



Other flags	Enter additional options in this field to affect the code generation stage of compilation.
Verbose	Selecting this checkbox specifies verbose compilation, where the command-line arguments of all internal compiler applications will be printed when building. See V: Verbose Compilation.

3.8.1.11 XC8 Linker - General

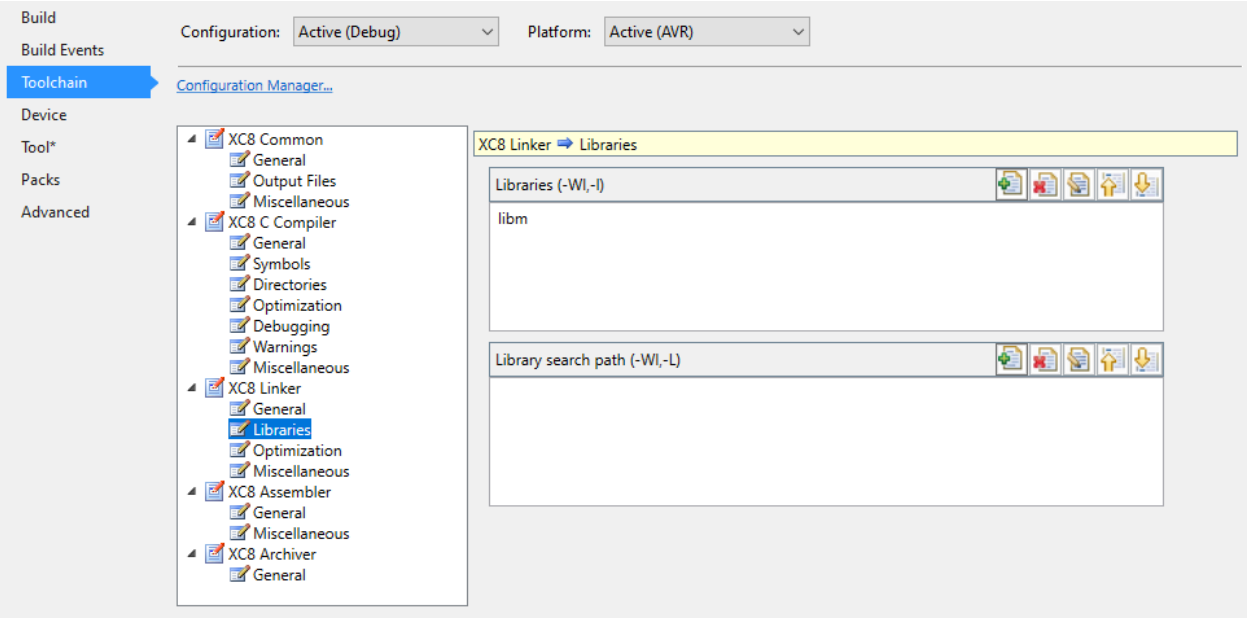
Figure 3-20. XC8 Linker - General



Do not use default libraries	Selecting this checkbox will prevent the standard system libraries being linked into the project. See Nodefaultlibs Option.
Generate MAP file	Select this checkbox to have a map file generated after linking. See Mapped Linker Options.
Use vprintf library	This option forces additional libraries to be linked, allowing floating-point format specifiers to work with avr-libc.

3.8.1.12 XC8 Linker - Libraries

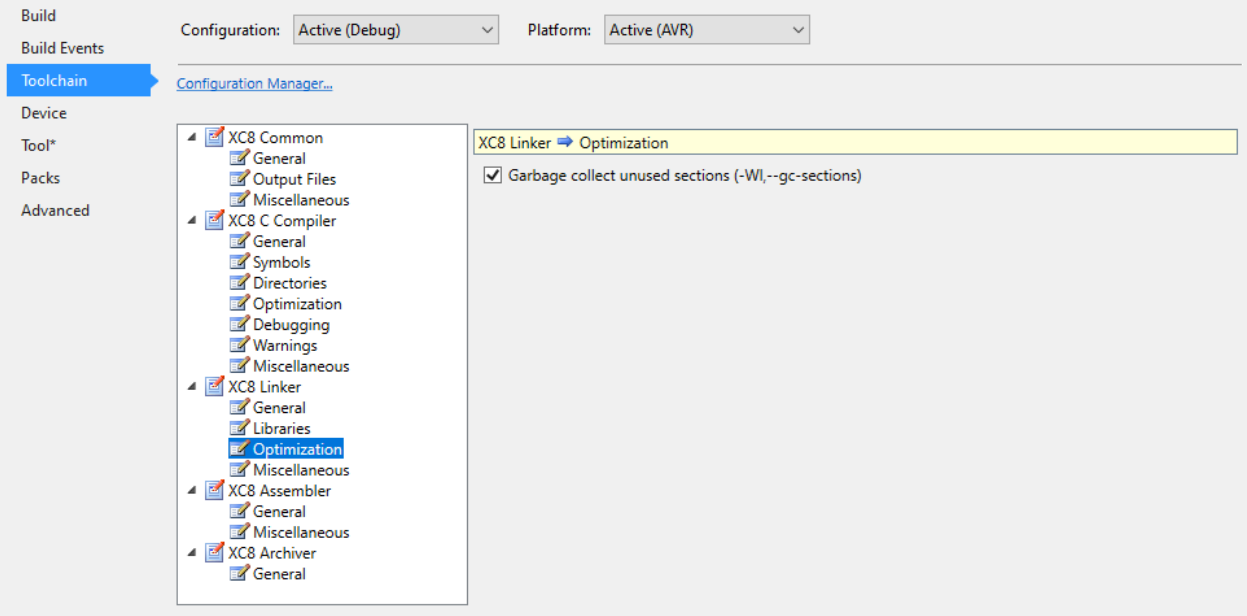
Figure 3-21. XC8 Linker - Libraries



Libraries	These controls allow you to specify additional libraries to be scanned for unresolved symbols when linking. See L: Specify Library File Option.
Library search path	These controls allow you to specify additional directories to be searched for library file. See L: Specify Library Search Path Option.

3.8.1.13 XC8 Linker - Optimization

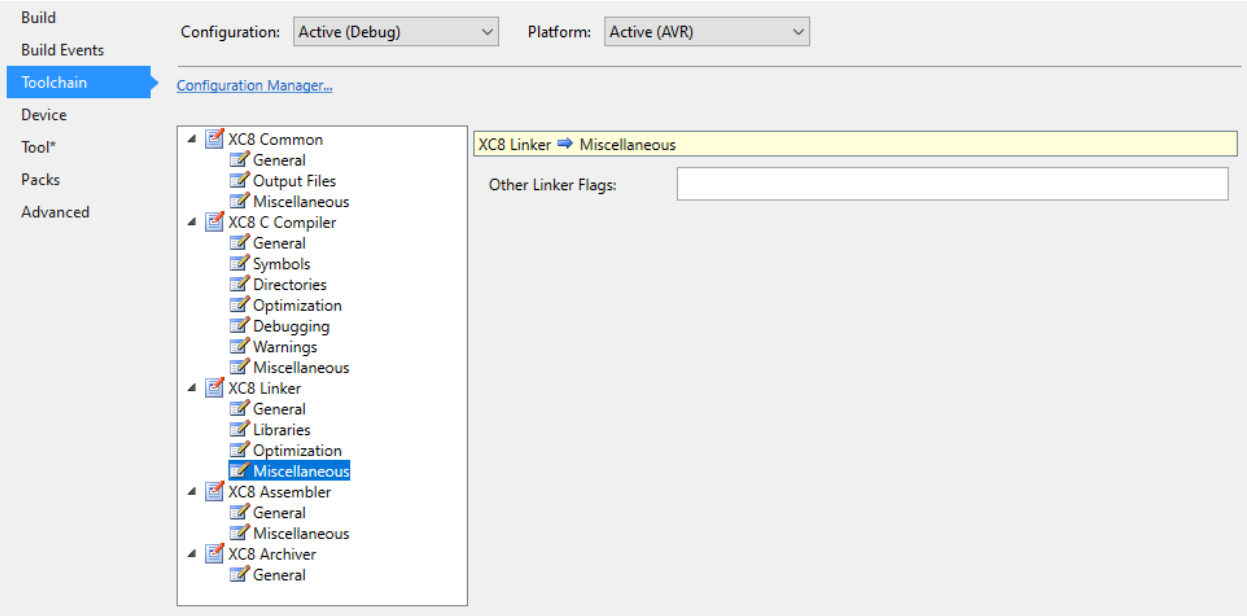
Figure 3-22. XC8 Linker - Optimization



Garbage collect unused sections	Select this checkbox to have unused sections removed from the program output. See Mapped Linker Options.

3.8.1.14 XC8 Linker - Miscellaneous

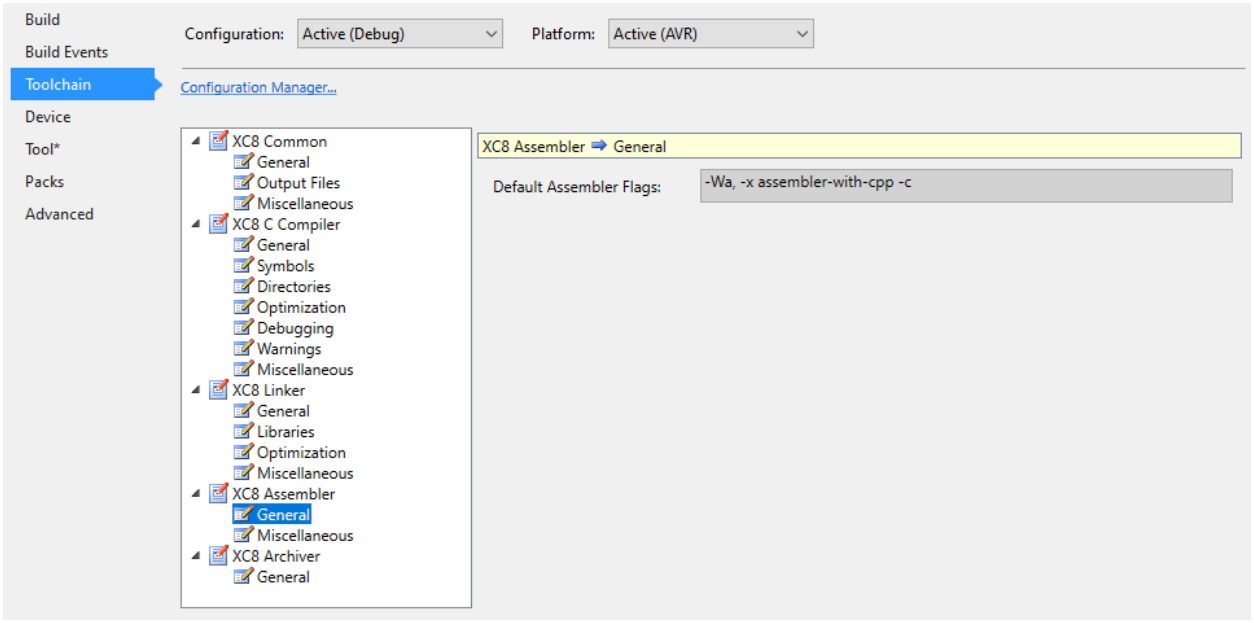
Figure 3-23. XC8 Linker - Miscellaneous



Other linker flags Enter additional options in this field to affect the link stage of compilation.

3.8.1.15 XC8 Assembler - General

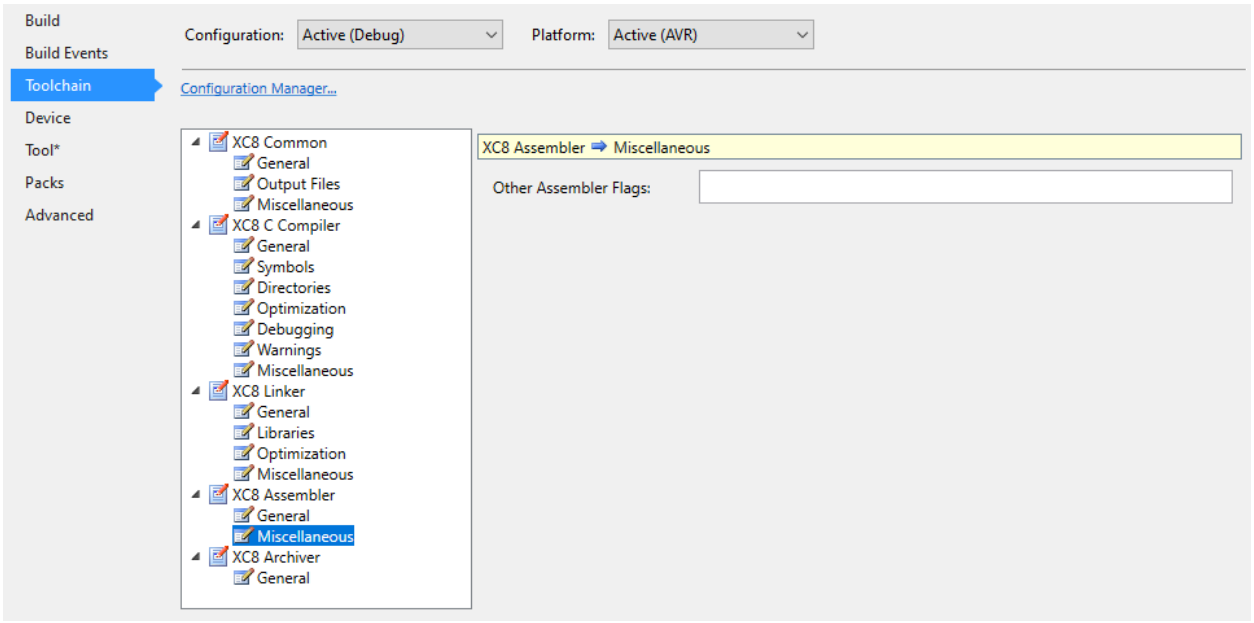
Figure 3-24. XC8 Assembler - General



Default Assembler Flags This is a summary of the default options that will be passed to the assembler.

3.8.1.16 XC8 Assembler - Miscellaneous

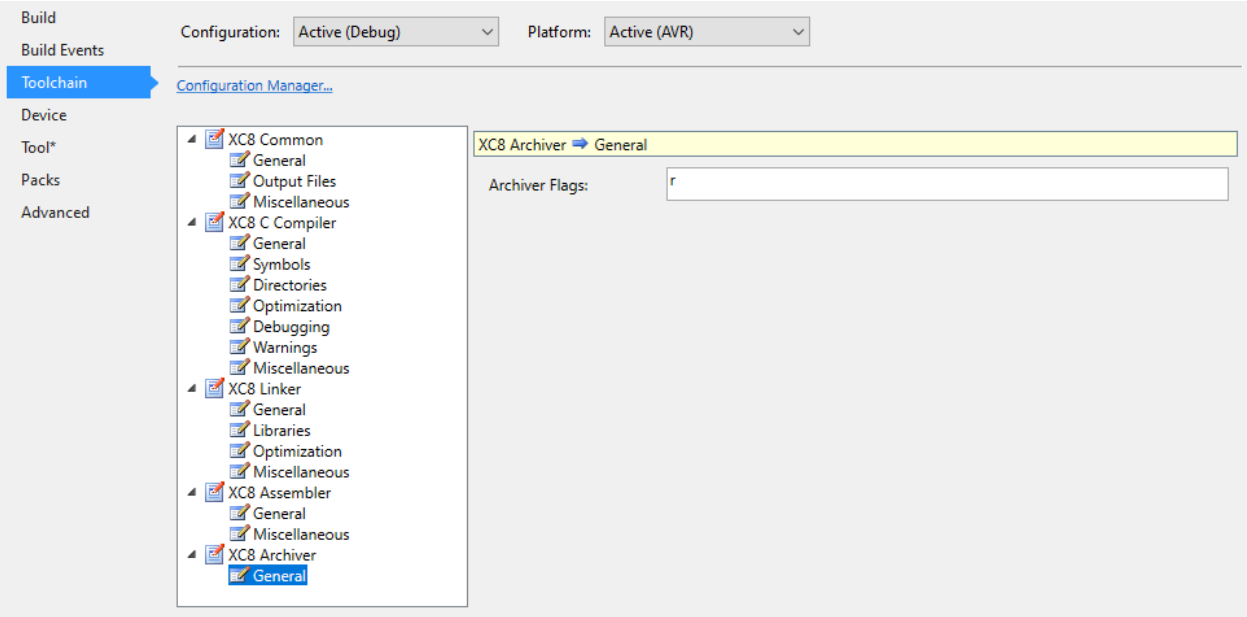
Figure 3-25. XC8 Assembler - Miscellaneous



Other assembler flags Enter additional options in this field to affect the assembler stage of compilation.

3.8.1.17 XC8 Archiver - General

Figure 3-26. XC8 Archiver - General



Archiver flags Enter additional options in this field to affect the archive stage of compilation. See Using the Archiver/Librarian

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 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 (CCI) standard 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 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.

4.1.2.2 Floating-point Types

The specification of the floating-point types, especially the `double` and `long double` types can vary from that mandated by the C standard. For example, the `DBL_DIG`/`LDBL_DIG` macro values specified by `<float.h>` are smaller than that indicated by the standard. Similarly the `DBL_EPSILON`/`LDBL_EPSILON` macro values are larger than that indicated by the standard.

These macros are listed in the *Microchip Unified Standard Library Reference Guide*.

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 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 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 can be managed by Microchip development environments.

4.2.2 Dual-Core Device Support

The MPLAB XC8 C Compiler supports dual-core devices in the AVR SD family.

These devices execute code on both cores simultaneously, comparing the results of each core to ensure consistent operation. The devices implement ECC (Error Correction Code) RAM, which can detect and correct some forms of data corruption. Multi-level traps indicate critical failures, such as a core execution mismatch, which will reset the device, as well as non-critical failures, such as invalid opcodes or ECC error, which can be masked by the programmer. A command line option is automatically set by the DFPs for dual-core devices, allowing the compiler to take into account all these special features when building project code.

Because of the devices' single-level pipelining and invalid opcode traps, there must always be a valid instruction after the last "real" instruction that is executed; otherwise fetching and decoding the next program memory location will cause a trap. For these devices, the linker automatically adds a `nop` instruction at the end of every code section that isn't immediately followed by another code section. It will add this instruction for all code section it allocates. To reduce the number of `nop` instructions required, the compiler will refrain from placing code and data in the same section, which might otherwise be performed, for example, when encoding jump tables.

As ECC errors can occur before program execution reaches the `main()` function, the startup code disables interrupts and masks non-critical traps immediately after Reset. This prevents user trap handling code executing before the runtime startup code has initialized the execution environment.

4.2.3 Instruction Set Support

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

4.2.4 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.

4.2.5 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.

4.2.6 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.

4.2.7 User Row Memory

Some devices have a User Row (USERROW) memory, which can be used for end-production data and is not affected by a device erase. It can be accessed as normal flash memory. Check your device data sheet to see if this memory is relevant for your selected device and for more information.

To place data into this memory, assign the data to the `.user_signatures` section, for example:

```
const unsigned char __attribute__((used, section(".user_signatures"))) x2[8] = {0x64, 0x28,
0x00, 0x01, 0xAB, 0x8A, 0xA8, 0x33};
```

This section is automatically placed at the correct address by the default linker script.

4.2.8 Boot Row Memory

Some devices have a Boot Row (BOOTROW) memory, which can be used for end-production data and is not affected by a device erase. The memory is accessible to the CPU both in locked and unlocked state, but only when the CPU is executing from the BOOT sector. Check your device data sheet to see if this memory is relevant for your selected device and for more information.

To place data into this memory, assign the data to any section prefixed with `.bootrow`, for example:

```
const unsigned char __attribute__((used, section(".bootrowinit"))) x2[8] = {0x64, 0x28, 0x00,
0x01, 0xAB, 0x8A, 0xA8, 0x33};
```

This section is automatically placed at the correct address by the default linker script.

4.2.9 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.

4.2.10 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 can 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();        // disable interrupts
  val = TCNT1;  // read timer value register; TEMP used internally
  SREG = sreg;  // restore state of interrupts

  return val;
}
```

4.2.11 Code Coverage

After purchase of the Analysis Tool Suite License (SW006027-2), 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. The preprocessor macro `__CODECOV` will be defined once the feature is enabled.

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 Microchip's Analysis Tool Suite License webpage for further information on the code coverage feature and other analysis tools.

4.2.12 Stack Guidance

Available with a PRO compiler license, the compiler's stack guidance feature can be used to estimate the maximum depth of any stack used by a program.

Runtime stack overflows cause program failure and can be difficult to track down, especially when the program is complex and interrupts are being used. The compiler's stack guidance feature constructs and analyzes the call graph of a program, determines the stack usage of each function,

and produces a report, from which the depth of stacks used by the program can be inferred. Monitoring a program's stack usage during its development will mitigate the possibility of stack overflow situations.

This feature is enabled by the `-mchp-stack-usage` command-line option.

Once enabled, the operation of the stack guidance feature is fully automatic. For command-line execution of the compiler, a report will be displayed directly to the console after a successful build. When building in the MPLAB X IDE, this same report will be displayed in the build view in the **Output** window.

A more detailed and permanent record of the stack usage information will be available in the map file, should one be requested using the `-Wl,-Map=`*`mapfile`* command-line option or the equivalent control in the MPLAB X IDE project properties.

4.2.12.1 Stack Guidance Information

The stack guidance features estimates the stack usage of the data stack used by programs compiled with the MPLAB XC8 C Compiler.

The following example shows a stack usage information summary that might be displayed after a successful build.

```
======================== STACK USAGE GUIDANCE ========================
In the call graph beginning at 'main',
   52 bytes of stack are required.

However, the following cautions exist:

1. Recursion has been detected:
   __fp_splitA
No stack usage predictions can be made.

2. The following labels are interrupt functions:
    __vector_18 uses 15 bytes
You must add stack allowances for those functions.

3. The following labels cannot be connected to the main call graph.
This is usually caused by some indirection:
   func1 uses 2 bytes
   func2 uses 0 bytes
   __fp_zero uses 0 bytes
You must add stack allowances for those functions.
======================================================================
```

The information will show the total estimated program stack usage.

In addition, a number of cautionary messages might be displayed. These clarify potential stack overflows reported by the above estimate and indicate additional stack usage that must be considered to determine a more accurate program stack usage. Due to factors that make it impossible to know when this additional memory might be used, these are not incorporated into the earlier summary.

The following cautions might be displayed in the described circumstances.

Recursive functions	Recursive function calls were detected in the main call graph. As the number of iterations of a recursive call cannot be predicted, the total software stack size cannot be determined, so no stack guidance is possible. The names of recursive functions detected are listed in this caution.
Disconnected functions	Functions that have not been directly called have been detected. Such functions might have been called, either indirectly in C code or by some other means, of they may not have been called at all. The stack usage of each function is indicated in this caution and need to be taken into account when considering the total software stack usage of your program.
Indeterminate calls	Indeterminate calls have been detected in the main call graph. These might be for example indirect calls or calls to a label. The number of bytes used by the calls is

	indicated in this caution and need to be taken into account when considering the total software stack usage of your program.
Indeterminate stack adjustments	Indeterminate stack adjustments have been detected in the main call graph. This might be the result of the use of variable-length arrays, the use of memory-allocation functions like `alloca` (which allocate memory on the stack rather than functions like `malloc`, which allocate on the heap), or the presence of functions without stack information available. No memory usage information regarding these situations can be reported. A static calculation of the additional bytes that are known to be used is indicated in this caution. These represent the minimum number of additional bytes used by your program.
Interrupt functions	Even though the size of the stack used by main-line and interrupt call graphs might be accurately known, interrupts can trigger at any time, and given this is the case, the compiler cannot reliably determine the program's total stack usage. This caution alerts you to the existence of interrupt functions and that the stack usage of these will need to be taken into account when considering the total software stack usage of your program.

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
`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 the declarations for <limits.h>, contained in the *Microchip Unified Standard Library Reference Guide*. The symbols presented there are preprocessor macros that 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 integer types. Integer values converted to a _Bool type result in 0 (false) if the integer 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 $(-1)^{sign} \times 2^{(exponent-127)} \times 1.mantissa$.

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:

$-1^{0}x2^{124}x1.302447676659$

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 the declarations for `<float.h>`, contained in the *Microchip Unified Standard Library Reference Guide*. The symbols presented there 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 an `int` type 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. A plain bit-field is unsigned; however the `-fsigned-bitfields` option can be used to change that behavior.

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 `__flash`*n*, 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 tinyAVR 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 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 128 KB. 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 Relax Option), which maintains 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 KB 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 Call-prologues Option)
- When generating switch/case dispatch tables (these can be inhibited by specifying the `-fno-jump-tables` option, 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 `l` 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, hence the result of the shift operation will be an `int`. Even though this result is assigned to the `long` variable, `result`, the result of the shift can never become larger than the size of an `int`, regardless of how much the constant is shifted. In this case, 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 Constants

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. They can alternatively be expressed as hexadecimal digits and a binary exponent initiated with either `p` or `P`. For example:

```
myFloat = -123.98E12;    // decimal representation
myFloat = 0xFFEp-22;     // hexadecimal 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 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 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 Absolute Objects In Program Memory.

Objects qualified with const are accessed as if they were qualified with __memx, see Memx Address Space Qualifier. If the definition of const-qualified objects also uses storage qualifiers, such as __flash or __flashn (see Flash Qualifier and 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 either 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 that 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 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 tinyAVR devices.

Pointers that use this qualifier can reference both data and program memory spaces in a similar way to pointers to const, see 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 64 KB segment. The section associated with the __flash qualifier is by default located in the lowest flash segment (segment 0), which occupies the lowest 64 KB 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 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 64 KB to 128 KB, with __flash2 qualified objects placed in a section that is linked to the segment from 128 KB to 196 KB, 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 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 tinyAVR core, for example the ATtiny40 device.

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

The `address(addr)` attribute is used to alias an object identifier to a peripheral address that may lie outside the IO address range. Such identifiers can then be used to address these memory-mapped peripherals. For example:

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

will internally equate `porta` to the address 0x600.

This attribute does not affect the allocation of any ordinary object in memory. To place objects at a specified address in the ordinary data memory, use the `__at()` specifier (see 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 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 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 Keep Attribute

The `keep` attribute prevents the linker from removing unused objects, regardless of the selected compiler optimization level or any garbage collection (`--gc-sections` linker option) performed by the linker.

4.3.10.8 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 Ext Option) a more portable macro, `__pack` (note the different spelling), is available.

4.3.10.9 Persistent Attribute

The `persistent` attribute is used to indicate that objects should not be cleared by the runtime startup code.

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 desirable for some data to be preserved across a Reset. The `persistent` attribute stores objects with static storage duration in a separate area of memory that is not altered by the runtime startup code.

For example, the following code ensures that the variables `intvar` and `mode` are not cleared at startup:

```
int __attribute__((persistent)) mode;

void test(void)
{
  static int __attribute__((persistent)) intvar;   /* must be static in this context */
  ...
}
```

If the CCI is enabled (see Ext Option) and the `<xc.h>` header is included, a more portable macro, `__persistent`, is available. For example, the following CCI-compliant code is similar to the above:

```
#include <xc.h>
__persistent int mode;

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

4.3.10.10 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.

4.3.10.11 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 Ext Option) a more portable macro, `__section(`*`section`*`)`, is available. See 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;
```

4.3.10.12 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 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 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 that are `static` only have their initial value assigned once during the program's execution. Thus, they generate more efficient code than initialized `auto` objects, which are assigned a value every time the block in which they are defined begins execution. Unlike `auto` objects, however, initializers for `static` objects must be constant expressions.

All `static` variables that 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 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 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 (see Section Attribute) after enabling the CCI (see Ext Option) to allocate them to a unique section, then link that section to the required address via an option. See 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 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 that have static storage duration and that are defined using `const` and any one of the `__flash`, `__flash`*n*, `__memx` qualifiers, or using `const` and the `progmem` attribute are always placed in program memory. When the compiler's const-in-progmem feature is enabled (the default state, or made a state explicit by using the `-mconst-data-in-progmem` option, see Const-data-in-progmem Option), objects defined using just the `const` type qualifier are also placed in program memory.

The `-mno-const-data-in-progmem` option disables the const-in-progmem feature and forces all objects defined using just the `const` type qualifier to instead be located in data memory.

The tinyAVR 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 `-mconst-data-in-config-mapped-progmem` option (see Const-data-in-config-mapped-progmem Option) can be used with those devices that support this memory mapping feature to have the linker place all `const`-qualified data in one 32 KB section and automatically initialize the relevant SFR register to ensure that these objects are mapped into data memory, where they can be accessed more efficiently.

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 `__flash`*n* qualifiers are limited to the memory available in one 64 KB segment of flash memory and must never cross a segment boundary.

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 Absolute Variables.

Objects in program memory can also be placed at specific positions by using the `__section()` specifier (see Section Attribute) after enabling the CCI (see Ext Option) to allocate them to a unique section, then link that section to the required address via an option. See Changing and Linking the Allocated Section for more information on changing the default linker options for sections.

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 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 that has static storage duration and file scope can be placed at an absolute address in program memory.

For example, the code:

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

will place the array, `settings`, at address 0x200 in 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 Register Usage).

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

4.4.7 Dynamic Memory Allocation

Dynamic memory allocation is a means for programs to manually request and release arbitrary-sized blocks of memory during program execution. It is available in some form for most devices supported by MPLAB XC compilers.

Programs can call the `malloc()` or `calloc()` library functions to allocate blocks of memory at runtime. The requested size does not need to be a constant expression, unlike with static or automatic allocation of objects (with the exception of variable-length automatic arrays, where supported). The contiguous region of memory spanning all the allocated memory blocks is collectively referred to as the heap.

An allocated block of memory that is no longer required can be released, or freed, by the program using the `free()` library function. Memory that has been freed can potentially be re-allocated at a later time. Failing to free memory no longer required can result in excessive memory use that could prevent the allocation of additional memory from the heap. If the size of an already allocated memory block needs to be changed, the `realloc()` function can be used.

Depending on your selected target device and options, MPLAB XC compilers may implement a full and/or simplified dynamic memory allocation scheme, or dynamic memory allocation may not be permitted at all. The exact operation of dynamic memory allocation with this compiler is described in the following section. The syntax and usage of the dynamic memory allocation functions is described in the *Microchip Unified Standard Library Reference Guide*.

4.4.7.1 Dynamic Memory Allocation for AVR Devices

The MPLAB XC8 C Compiler implements a unrestricted dynamic memory allocation scheme for all target devices; however, the smaller devices, such at tinyAVR devices, are unlikely to have enough program memory to accommodate the required library functions, nor have enough data memory to allow dynamic memory allocation to be useful.

The dynamic memory allocation scheme permits the use of all the standard memory allocation library functions. Initially, memory can be allocated as required, subject only to the maximum size reserved for the heap. Memory blocks that are freed after allocation are coalesced with neighboring

free memory and grouped into bins of similar sizes. This memory will be considered for re-allocation whenever additional memory is subsequently requested.

Although the allocation scheme attempts to utilize memory efficiently, fragmentation can always occur. Use caution when allocating and freeing memory. Prudent program designers will avoid using dynamic memory allocation, if at all possible.

Although successful calls to memory allocation functions will return a block with sufficient size to store the requested number of bytes, the total size of each allocated block may be larger than the requested size, and that size is dependent on whether the `free()` function is called in the program.

If `free()` is never called, then the memory allocation functions allocate a block with the requested memory size if possible and a pointer to that block returned. A request to allocate 1 byte of memory, such as `malloc(1)`, will in this case allocate a 1-byte block, provided enough free memory exists.

If `free()` is called in the program, then the compiler will include additional memory with each block allocated to manage the memory. In this case, each allocated block will include a 2-byte-wide header and an additional 2 bytes will be used as a free list pointer and thus, the minimum amount of memory that can be allocated for a block is 4 bytes. As the 2 bytes located before the memory whose address is returned after a successful allocation contain the header information, your program should never assume that this memory might contain allocated user data.

Regardless of whether the `free()` function is called, a request to allocate zero bytes of memory, such as `malloc(0)`, will be treated like a request to allocate 1 byte of memory, and if successful, will return a pointer to the memory allocated.

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

The variables `__malloc_heap_start` and `__malloc_heap_end` can be used to restrict the memory allocated by the `malloc()` and `calloc()` functions. These variables are statically initialized to point to `__heap_start` and `__heap_end`, respectively. The `__heap_start` variable is set to an address just beyond the last data memory address used by Best Fit Allocator after the allocation of static objects, and `__heap_end` is set to 0, which places the heap 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. Since these are addresses in RAM, the MSb is set in the specified addresses. The heap will extend up to address 0xffff.

```
-Wl,-Tdata=0x801100,--defsym=__heap_end=0x80ffff
```

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 remains inaccessible 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 allocate 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 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 Division, Shift and Modulo Operations

Some AVR EC family of devices implement a 32-bit Divide Accelerator (DIVA) module, which can perform 32-bit signed or unsigned integer hardware divisions and 32-bit shifts and rotates in hardware. The compiler can employ routines which set up, initiate, and obtain results from this module rather than call library routines that perform the same operation.

When building for devices that implement this module, its operation is automatically enabled, so that division, shift and modulo operations will invoke the DIVA routines. The `-mdiva` driver option makes this action explicit.

The `-mno-diva` option can be used to disable this operation, having these operations instead performed by conventional library routines that will be larger and slower than the DIVA routines. There is also a `__nodiva` function attribute, so that this functionality can be controlled on a per function basis.

The preprocessor macro `__AVR_HAVE_DIVA__` will be set if the device being targeted has this module and it is enabled.

Some floating-point operations also employ integer operations that might also be performed by the DIVA module when possible.

4.5.3 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
```

4.5.4 Switch Statements

By default, jump tables are used to optimize `switch()` statements. This can be made explicit by using the `-fjump-tables` option. The `-fno-jump-tables` option prevents jump tables from being used and will use sequences of compare sequences instead. Jump tables are usually faster to execute, but for `switch()` statements where most of the jumps are destined for the `default` label, they might produce excessive amounts of code.

Jump tables use the limited-range `lpm` assembler instruction. 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 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. This can increase the execution speed of a program.

Any function with internal linkage (those using a `static` specifier) can be an inline function, but the C99 language standard imposes restrictions on how functions with external linkage can use the `inline` specifier. A file scope declaration for a function using the `inline` function specifier (but not `extern`) is known as an inline definition and merely provides an alternative to an external definition of that function. You can provide an additional external definition of that function in others modules, or you could, for example, make the definition for the function external in the same module by providing an declaration using `extern`.

The following is an example of a function which has been made a candidate for in-lining.

```
extern int combine(int x, int y);  // make this an external definition

inline int combine(int x, int y) {
```

```
  return 2 * x - y;
}
```

The compiler can encode a call to either the inline definition (in-lining the function) or the external definition, at its discretion. Your code should not make any assumption about whether in-lining 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.

All function calls to the inline definition 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 in-lining 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. As an alternative to in-lined functions, consider using a preprocessor macro.

4.7.1.2 Nopa Specifier

If the CCI is enabled (see Ext Option), 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.

4.7.2 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 Keep Attribute

The `keep` attribute prevents the linker from removing unused functions, regardless of the selected compiler optimization level or any garbage collection (`--gc-sections` linker option) performed by the linker.

4.7.2.2 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.

4.7.2.3 no_gccisr Attribute

The `no_gccisr` attribute disables any optimization of the context switch code associated with the interrupt function. See also ISR Prologues Option.

4.7.2.3.1 Nodiva Attribute

The `__nodiva` attribute indicates that the function should not use be encoded to use the DIVA module for and division, shift or modulus operations. Insteadm, these operations will be performed using software routines.

4.7.2.4 Nopa Attribute

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

4.7.2.5 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.

4.7.2.6 Section Attribute

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

For example, the following CCI-compliant code places the code associated with the `readInput` function in a unique section called `myText`:

```
int __section("myText") readInput(int port)
{ ... }
```

4.7.2.7 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 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 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 compiler can generate code for projects using the full AVR interrupt vector table, where each interrupt source has a corresponding interrupt number and entry in the vector table, and the hardware executes the instruction at the entry matching the interrupt source. Check your device data sheet to see how interrupts operate and can be configured.

Alternatively, for those devices that support the Compact Vector Table (CVT) feature, the project can be configured to use a four-entry vector table. This feature forces all level 0 interrupt sources to share the same interrupt vector number. Thus, the interrupts generated by all these sources share the same ISR, reducing the project's memory footprint. This feature is most suitable for devices with limited memory and applications using a small number of interrupt sources. Note, however, that since additional code might be needed in the ISR to determine the interrupt source, the response time of the ISR could be longer.

When available and once enabled, only four vectors are present in a compact interrupt vector table, those being for the following interrupts:

1. Reset
2. Non-maskable interrupts (NMI)
3. Priority Level 1 (LVL1) interrupts
4. Priority level 0 (LVL0) interrupts

4.8.1 Interrupt Service Routines

Observe the following information and guidelines when writing interrupt functions (also known as Interrupt Service Routines, or ISRs).

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 sometimes cleared in the course of processing the interrupt, either when the handler is invoked or by reading a particular hardware register; however, there are other instances when the flag must

be cleared manually by code. Failure to clear the flag might result in the interrupt triggering again as soon as the current ISR returns.

The interrupt 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.

The hardware globally disables interrupts when an interrupt is executed. Do not re-enable interrupts inside the interrupt function. This is performed automatically by a special `reti` return instruction used by the compiler to terminate the ISR execution.

Keep the ISR as simple as possible. Complex code will typically use many registers, which might increase the size of the context switch code.

The compiler processes interrupt functions differently from other functions, generating code to save and restore any registers that are used by the function and that are not saved by the device hardware. These functions additionally use the `reti` instruction, so they must not be called directly from C code, but they 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.

The following sections describe how interrupt functions should be written and incorporated into your project based on the vector table configuration.

4.8.1.1 Writing ISRs for a Full Vector Table

If your project is using the full vector table, observe the following guidelines when writing an ordinary ISR.

Enable the CCI (Ext Option). Write as many ISRs as required using the `__interrupt()` specifier, using `void` as the ISR return type and as the parameter list. Consider implementing default interrupt functions to handle accidental triggering of unused interrupts.

The `__interrupt()` specifier takes a numerical argument to indicate the interrupt number, hence the interrupt source. Macros whose names ends with `vect_num` are available once you have included `<xc.h>` in your program and can be used to indicate the interrupt number in a more readable way.

At the appropriate point in your code, enable the interrupt sources required and the global interrupt enable.

An example of an interrupt function is shown below. Ensure the CCI is enabled using the `-mext=cci` option.

```
#include <xc.h>
void __interrupt(SPI_STC_vect_num) spi_Isr(void) {
  process(SPI_ClientReceive());
  return;
}
```

More complex interrupt function definitions can be created using macros and attributes defined by `<avr/interrupt.h>`, 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 with the `ISR()` macro 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:

```
#include <avr/interrupt.h>
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:

```
#include <avr/interrupt.h>
ISR(IO_PINS_vect, ISR_NOBLOCK)
{ ... }
```

The `ISR_NOBLOCK` expands to `__attribute__((interrupt))`, which can be used instead.

If one ISR is to be used with more than one interrupt vector, then you can define that ISR in the usual way (using `__interrupt()` and enabling the CCI) for one vector then reuse that ISR for other vector definitions using the `ISR_ALIASOF()` argument.

```
#include <xc.h>
void __interrupt(PCINT0_vect_num)
{ ... }
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.

```
#include <avr/interrupt.h>
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.

4.8.1.2 Writing ISRs for a Compact Vector Table

If your project is using a compact vector table, the guidelines discussed for full vector table ISRs also apply; however, note the following differences.

Use the `-mcvt` option to inform the compiler that your program will use the compact vector table. By default, this feature is disabled. This option will set the appropriate SFR bits in the runtime startup code, as well as set up the vector table in the `.vectors` section.

Write one ISR for each of the non-maskable, Priority Level 1, and Priority Level 0 interrupts, as required by your program. There can be at most three ISRs. If the ISR for the level 0 interrupts is

handling multiple interrupt sources, it must check the relevant interrupt flags to see which source has triggered the interrupt. For example:

```
#Include <avr/interrupt.h>

void __interrupt(CVT_NMI_vect_num) {
    // process the NMI here
}
void __interrupt (CVT_LVL1_vect_num) {
    // process the level 1 interrupt
}

void __interrupt (CVT_LVL0_vect_num) {
    if (PORTA.INTFLAGS & 0x1) {
        // process PORTA bit #0 change of state here
    }
    // process all other level 0 interrupts here
}
```

When enabled, this feature will define a preprocessor macro called `__AVR_COMPACT_VECTOR_TABLE__`, which can be used to conditionally include interrupt fucntions into the program.

On devices supporting CVT, this feature will also be enabled if `-mno-interrupts` is specified. This flag can also be used to reduce context-saving code in prologue/epilogue in certain architectures, but it will enable compact vector table support as well.

4.8.2 Changing the Default Interrupt Function Allocation

You can use the `__at()` specifier (see 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 Writing ISRs for a Full Vector Table). 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 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 from that after a hardware Reset.

It is recommended that the `main()` function does not end. Place all or some of your code in `main()` within a loop construct (such as a `while(1)`) that 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 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).
`.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.

The initialization of objects can be disabled using -Wl,--no-data-init; however, code that relies on objects containing their initial value will fail.

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. Such objects are linked in a different area of memory and are not altered by the runtime startup code.

Related Links

Persistent Attribute

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.

The clearing of objects can be disabled using -Wl,--no-data-init; however, code that relies on objects containing their initial value will fail.

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.

Related Links
Persistent Attribute

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 `.initn` 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 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 provides C90- and C99-validated libraries of functions, macros, types, and objects that can assist with your code development.

The Standard C libraries are described in the separate *Microchip Unified Standard Library Reference Guide* document, whose content is relevant for all MPLAB XC C compilers.

The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping and math functions (trigonometric, exponential and hyperbolic).

In addition to the libraries supplied with the compiler, you can create your own libraries from source code you have written.

4.10.1 Smart IO Routines

The library code associated with the print and scan families of IO functions can be customized by the compiler with each compilation, based on compiler options and how you use these functions in your project. This can reduce the amount of redundant library code being linked into the program image, hence can reduce the program memory and data memory used by a program.

The smart output (print family) functions are:

`printf`	`fprintf`	`snprintf`	`sprintf`
`vfprintf`	`vprintf`	`vsnprintf`	`vsprintf`

The smart input (scan family) functions are:

`scanf`	`fscanf`	`sscanf`
`vfscanf`	`vscanf`	`vsscanf`

When this feature is enabled, the compiler analyzes your project's C source code every time you build, searching for calls to any of the smart IO functions. The conversion specifications present

in the format strings are collated across all calls, and their presence triggers inclusion of library routines with the associated functionality in the program image output.

For example, if a program contained only the following call:

```
printf("input is: %d\n", input);
```

when smart IO is enabled, the compiler will note that only the `%d` placeholder has been used by the `printf` function in the program, and the linked library routine defining `printf` will thus contain a basic functionality that can at least handle the printing of decimal integers. If the following call was added to the program:

```
printf("input is: %f\n", ratio);
```

the compiler will then see that both the `%d` and `%f` placeholders were used by `printf`. The linked library routine would then have additional functionality to ensure that all the requirements of the program can be met.

Specific details of how the smart IO feature operates for this compiler are detailed in the following section. The syntax and usage of all IO functions, which are part of the `<stdio.h>` header, are described in the *Microchip Unified Standard Library Reference Guide*.

4.10.1.1 Smart IO For AVR Devices

When using MPLAB XC8 C Compiler, multiple IO library variants, representing increasingly complex subsets of IO functionality, are available and are linked into your program based on the `-msmart-io` option and how you use the smart IO functions in your project's source code.

When the smart IO feature is disabled (`-msmart-io=0`), a full implementation of the IO functions will be linked into your program. All features of the IO library functions will be available, and these may consume a significant amount of the available program and data memory on the target device.

When the smart IO feature is enabled (`-msmart-io=1` or `-msmart-io`), the compiler will link in the least complex variant of the IO library that implements all of the IO functionality required by the program, based on the conversion specifications detected in the program's IO function format strings. This can substantially reduce the memory requirements of your program, especially if you can eliminate in your program the use of floating-point features in calls to smart IO functions. This is the default setting.

The compiler analyzes the usage of each IO function independently, so while the code for a particular program might require that the `printf` function be full featured, only a basic implementation of the `snprintf` function might be required, for example.

If the format string in a call to an IO function is not a string literal, the compiler will not be able to detect the exact usage of the IO function, and a full-featured variant of the IO library will be linked into the program image, even with smart IO enabled. In this instance, the `-msmart-io-format="`*`fmt`*`"` option can be used to specify those conversion specifications that have been used in non-literal format strings, allowing a more optimal library to be linked. You must ensure that your program only uses the indicated conversion specifications; otherwise, IO functions may not work as expected.

For example, consider the following four calls to smart IO functions.

```
vscanf("%d:%li", va_list1);
vprintf("%-s%d", va_list2);
vprintf(fmt1, va_list3); // ambiguous usage
vscanf(fmt2, va_list4);  // ambiguous usage
```

When processing the last two calls, the compiler cannot deduce any usage information from either of the format strings. If it is known that the format strings pointed to by `fmt1` and `fmt2` collectively use only the `%d`, `%i` and `%s` conversion specifiers, the `-msmart-io-format=fmt="%d%i%s"` option should be issued.

These options should be used consistently across all program modules to ensure an optimal selection of the library routines included in the program image.

4.10.2 Standard Libraries

The Microchip Unified Standard Library encompasses the functions defined by the standard C language headers provided by the C99 language specification, as well as any types and preprocessor macros needed to facilitate their use. This library is shipped with all Microchip C Compilers.

The behavior of and interface to the library functions as well as the intended use of the library types and macros is described in the *Microchip Unified Standard Library Reference Guide* document.

4.10.3 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 Archiver/Librarian).

Once built, user-defined libraries can be used on the command line along with the source files. Additional libraries can be added to your IDE project, or specified using an option.

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.

4.10.4 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 Library Example Code 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 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)

.....continued

Letter	Constraint	Range
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	
Cx*n*	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	
Y0*n*	Fixed-point or integer constant, where *n* ranges from 0 to 2	n
Ym*n*	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

.....continued

Mnemonic	Constraints	Mnemonic	Constraints
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
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 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.

The optimization level determines the available optimizations, which are listed in the table below.

Table 4-13. Optimization Level Sets

Level	Optimization sets available
`O0`	Rudimentary optimization
`O1`	Minimal optimizations
`O2`	All generic optimizations
`O3` (Licensed only)	All generic optimizations plus those targeting speed

.....continued	
Level	**Optimization sets available**
`Os` (Licensed only)	All generic optimizations plus those targeting space

The minimal code generator optimizations consist of the following.

- Simplification and folding of constant expressions to simplify expressions.
- Expression tree optimizations to ensure efficient assembly generation.
- Propagation of constants is performed where the numerical contents of a variable or expression can be determined at compile time. Variables which are not `volatile` and whose value can be exactly determined are replaced with the numerical value. Uninitialized global variables are assumed to contain zero prior to any assignment to them.
- Unreachable code is removed. C Statements that cannot be reached are removed before they generate assembly code. This allows subsequent optimizations to be applied at the C level.

The following is a list of more advanced C-level optimizations, which simplify C expressions or code produced from C expressions.

- Strength reductions and expression transformations are applied to all expression trees before code is generated. This involves replacing expressions with equivalent, but less costly operations.
- Unused variables in a program are removed. This applies to all variables. Variables removed will not have memory reserved for them, will not appear in any list or map file, and will not be present in debug information (will not be observable in the debugger). A warning is produced if an unused variable is encountered. Taking the address of a variable or referencing its assembly-domain symbol in hand-written assembly code also constitutes use of the variable.
- Redundant assignments to variables not subsequently used are removed, unless the variable is `volatile`. The assignment statement is completely removed, as if it was never present in the original source code. No code will be produced for it and you will not be able to set a breakpoint on that line in the debugger.
- Unused functions in a program are removed. A function is considered unused if it is not called, directly or indirectly, nor has had its address taken. The entire function is removed, as if it was never present in the original source code. No code will be produced for it and you will not be able to set a breakpoint on any line in the function in the debugger. Referencing a function's assembly-domain symbol in a separate hand-written assembly module will prevent it being removed. The assembly code need only use the symbol in the `.global` directive.
- Dereferencing pointers with only target can be replaced with direct access of the target object. This applies to data and function pointers.

The following is a list of more advanced assembly-level optimizations, which simplify the assembly code generated by the compiler. These optimizations do not apply to hand-written assembly code, whether that be placed in-line with C code or as separate assembly modules.

- Inlining of small routines is done so that a call to the routine is not required. Only very small routines (typically a few instructions) that are called only once will be changed so that code size is not adversely impacted. This speeds code execution without a significant increase in code size.
- Explicit inlining of functions that use the inline specifier (see Inline Specifier).
- Procedural abstraction is performed on assembly code sequences that appear more than once. This is essentially a reverse inlining process. The code sequences are abstracted into callable routines. A call to this routine will replace every instance of the original code sequence. This optimization reduces code size considerably, with a small impact on code speed. It can, however, adversely impact debugging.
- Replacement of long-form call and jump instructions with their shorter relative forms, either directly or via trampoline constructs.

- Peephole optimizations are performed on every instruction. These optimizations consider the state of execution at and immediately around each instruction – hence the name. They either alter or delete one or more instructions at each step.

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 E: Preprocess Only).

Assembler source files are preprocessed if the file uses a `.S` extension.

4.13.1 Preprocessor Directives

The XC8 accepts several specialized preprocessor directives, in addition to the standard directives. All of these are tabulated below.

Table 4-14. 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`

`#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 # operator to convert an argument to a string and the ## operator 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 might require further expansion. Remember that once a macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

Preprocessor macros can take a variable number of arguments, just like C functions. Use an ellipsis (...) after any named arguments to define a variable argument list. Use the special token `__VA_ARGS__` to expand this argument list in the macro replacement text. For example, the following PIC18 code places a variable number of words into program memory. Note that the arguments were converted to strings using the # operator so as to be usable in the `__asm()` statement.

```
#define PLACE_WORDS(...) \
__asm("PSECT cdata,class=CODE,reloc=2,noexec"); \
__asm("DW " #__VA_ARGS__)
```

Call this macro in the usual way, passing it as many arguments as are required, for example:

```
PLACE_WORDS(0x100, 1, 223, 0x30F0);
```

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 sometimes different from the C-domain result for the same sequence. The preprocessor processes unsigned and signed literal integer values in the controlling expression as if they had the respective type `uintmax_t` or `intmax_t`, defined in `<stdint.h>`. For the MPLAB XC8 C Compiler, the size of these types 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 4-15. 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.

.....continued

Symbol	Description
__AVR_ASM_ONLY__	Indicates that the selected device can only be programmed in assembly.
__AVR_COMPACT_VECTOR_TABLE__	Indicates that the compiler will configure the program to use compact interrupt vector tables.
__AVR_CONST_DATA_IN_CONFIG_MAPPED_PROGMEM__	Indicates that `const`-qualified objects will be placed in a 32 KB program section that will be mapped into data memory.
__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_DIVA__	Indicates that the device being targeted has the DIVA module and it is enabled.
__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 8 KB 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
__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 implements the avrtiny core. Note that some ATtiny devices do not use the avrtiny core.
__AVR_TINY_PM_BASE_ADDRESS__=*addr*	Deprecated; use __AVR_PM_BASE_ADDRESS__. Indicates the avrtiny core 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.

.....continued

Symbol	Description
__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
__CODECOV	When code coverage is enabled, with value __CC_RAM (1).
__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.
__LINE__	Indicates the source line number.
__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__	Indicates the current date, e.g., May 21 2004.
__FILE__	Indicates this source file being preprocessed.
__TIME__	Indicates 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 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 with commands inside 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 Wl: Pass Option To The 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 `.finin` 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.

4.14.2 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 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. To use the `__section()` specifier, the CCI (see Ext Option) must be enabled.

```
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. If this is not the case, then you might need to copy the default linker script, modify it appropriately, and specify it using the `-Tscript` driver option. The modification must define an output section using the same name as the section you have specified. The following, for example, will allocate any input sections called `myBss` to the output section with the same name.

```
myBss :
  {
    KEEP(*(myBss))
  }
```

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, use an option similar to:

```
-Wl,--section-start=myBss=0x800300
```

The `-Wl,-Tsection=addr` option can be used to position standard sections, like the `.text`, `.data` and `.bss` sections; however, since a Best Fit Allocator (BFA) is used with MPLAB XC8, these sections are not commonly used by the compiler to store objects or code.

Although the `-Wl,-Ttext=addr` option will move the `.text` section to the offset specified, this option also sets the starting address from which the linker begins program memory allocation for all code (even that handled by the BFA). Thus, when this option is used, the `.text` section will start at the exact offset specified; other code sections will be located after this. Similarly, the `-Wl,-Tdata=addr` option also sets the starting address from which the linker begins data (both initialized and uninitialized) allocation.

4.14.3 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 4-16. 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.

4.14.3.1 Linker Script Symbols

Several symbol are used by the compiler's linker scripts to locate the start and maximum end addresses of the memory regions in which sections are placed. While it is possible to redefine these symbols using the `-Wl,--defsym,`*`symbol=address`* option, there are typically compiler options that can be more easily used to achieve the desired goal.

The `__TEXT_REGION_LENGTH__` and `__DATA_REGION_LENGTH__` symbols, for example, define the maximum length (size) of the regions used to hold program code and RAM-based objects, respectively. Rather than adjust these directly, consider using the `-mreserve` option (Reserve Option), which can be used to remove memory ranges from consideration when linking. For example, to reduce the available program memory on a device with 8KB of program memory to only the lower 6 KB, use the option `-mreserve=rom@0x1800:0x2000`.

The `__TEXT_REGION_ORIGIN__` and `__DATA_REGION_ORIGIN__` symbols define the starting address of the regions used to hold program code and RAM-based objects, respectively. Rather than adjust these directly, consider using the `-Wl,-Ttext` and `-Wl,-Tdata` options, respectively (Changing and Linking the Allocated Section). For example, to have all code start at address 0x100, use the option `-Wl,-Ttext=0x100`.

4.14.4 Replacing Library Modules

For library functions that are weak (see 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 [file.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` *`modules`*	Delete module
`-m` *`modules`*	Re-order modules
`-p`	List modules
`-r` *`modules`*	Replace modules
`-t`	List modules with symbols
`-x` *`modules`*	Extract modules
`--target` *`device`*	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.

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 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	Implementation
"How a diagnostic is identified (3.10, 5.1.1.3)."	
	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*
"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)."	
	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	Implementation
"The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2)."	
"The name and type of the function called at program start-up in a freestanding environment (5.1.2.1)."	
	`int main (void);`
"The effect of program termination in a freestanding environment (5.1.2.1)."	
	Interrupts are disabled and the programs loops indefinitely
"An alternative manner in which the `main` function may be defined (5.1.2.2.1)."	
	`void main (void);`
"The values given to the strings pointed to by the argv argument to `main` (5.1.2.2.1)."	
	No arguments are passed to `main`. Reference to `argc` or `argv` is undefined.
"What constitutes an interactive device (5.1.2.3)."	
	Application defined.
"The set of signals, their semantics, and their default handling (7.14)."	
	Signals are not implemented.
"Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1)."	
	Signals are not implemented.
"Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program start-up (7.14.1.1)."	
	Signals are not implemented.
"The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5)."	
	The host environment is application defined.
"The manner of execution of the string by the `system` function (7.20.4.6)."	
	The host environment is application defined.

6.4 Identifiers

ISO Standard	Implementation
"Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2)."	
	None.
"The number of significant initial characters in an identifier (5.2.4.1, 6.4.2)."	
	All characters are significant.

6.5 Characters

ISO Standard	Implementation
"The number of bits in a byte (C90 3.4, C99 3.6)."	
	8.
"The values of the members of the execution character set (C90 and C99 5.2.1)."	
	The execution character set is ASCII.
"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)."	
	The execution character set is ASCII.
"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)."	
	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.
"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)."	
	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.
"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)."	
	The binary representation of the source character set is preserved to the execution character set.
"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)."	
	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.
"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)."	
	Multi-byte characters are not supported in source files.
"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)."	
	Multi-byte characters are not supported in source files.
"The current locale used to convert a wide string literal into corresponding wide character codes (C90 6.1.4, C99 6.4.5)."	
	Wide strings are not supported.
"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)."	
	Multi-byte characters are not supported in source files.

6.6 Integers

ISO Standard	Implementation
"Any extended integer types that exist in the implementation (C99 6.2.5)."	

.....continued

ISO Standard	Implementation
	The `__int24` and `__uint24` keywords designate a signed and unsigned, respectively, 24-bit integer type.
"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)."	
	All integer types are represented as two's complement and all bit patterns are ordinary values.
"The rank of any extended integer type relative to another extended integer type with the same precision (C99 6.3.1.1)."	
	There are no extended integer types with the same precision.
"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)."	
	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.
"The results of some bitwise operations on signed integers (C90 6.3, C99 6.5)."	
	The right shift operator (>> 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) of signed or unsigned values always clear the LSb of the result. Other bitwise operations act as if the operand was unsigned.

6.7 Floating-Point

ISO Standard	Implementation
"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)."	
	The accuracy is unknown.
"The rounding behaviors characterized by non-standard values of `FLT_ROUNDS` (C90 and C99 5.2.4.2.2)."	
	No such values are used.
"The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD` (C90 and C99 5.2.4.2.2)."	
	No such values are used.
"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)."	
	The integer is rounded to the nearest floating-point representation.
"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)."	
	A floating-point number is rounded down when converted to a narrow floating-point value.
"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)."	
	Not applicable; `FLT_RADIX` is a power of 2.
"Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (C99 6.5)."	
	The pragma is not implemented.
"The default state for the `FENV_ACCESS` pragma (C99 7.6.1)."	
	This pragma is not implemented.
"Additional floating-point exceptions, rounding modes, environments, classifications and their macro names (C99 7.6, 7.12)."	
	None supported.
"The default state for the `FP_CONTRACT` pragma (C99 7.12.2)."	
	This pragma is not implemented.
"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)."	
	The exception is not raised.
"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)."	

.....continued

ISO Standard	Implementation
	The exception is not raised.

6.8 Arrays and Pointers

ISO Standard	Implementation
"The result of converting a pointer to an integer or vice versa (C90 6.3.4, C99 6.3.2.3)."	
	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.
"The size of the result of subtracting two pointers to elements of the same array (C90 6.3.6, C99 6.5.6)."	
	The signed integer result will have the same size as the pointer operands in the subtraction.

6.9 Hints

ISO Standard	Implementation
"The extent to which suggestions made by using the `register` storage-class specifier are effective (C90 6.5.1, C99 6.7.1)."	
	The `register` storage class can be used to locate certain objects in a register (see Section on *Register Usage*).
"The extent to which suggestions made by using the `inline` function specifier are effective (C99 6.7.4)."	
	A function might be inlined if a 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	Implementation
"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)."	
	A plain `int` bit-field is treated as an unsigned integer. The `-fsigned-bitfields` option can be used to treat bit-fields as signed.
"Allowable bit-field types other than `_Bool`, `signed int`, and `unsigned int` (C99 6.7.2.1)."	
	All integer types are allowed.
"Whether a bit-field can straddle a storage unit boundary (C90 6.5.2.1, C99 6.7.2.1)."	
	A bit-field can straddle a storage unit.
"The order of allocation of bit-fields within a unit (C90 6.5.2.1, C99 6.7.2.1)."	
	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.
"The alignment of non-bit-field members of structures (C90 6.5.2.1, C99 6.7.2.1)."	
	No alignment is performed.
"The integer type compatible with each enumerated type (C90 6.5.2.2, C99 6.7.2.2)."	
	A `signed int` or `unsigned int` can be chosen to represent an enumerated type.

6.11 Qualifiers

ISO Standard	Implementation
"What constitutes an access to an object that has `volatile`-qualified type (C90 6.5.3, C99 6.7.3)."	
	Each reference to the identifier of a `volatile`-qualified object constitutes one access to the object.

6.12 Pre-Processing Directives

ISO Standard	Implementation
"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)."	
	The character sequence between the delimiters is considered to be a string which is a file name for the host environment.
"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)."	
	Yes.
"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)."	
	Yes.
"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)."	
	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`.
"How the named source file is searched for in an included " " delimited header (C90 6.8.2, C99 6.10.2)."	
	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.
"The method by which preprocessing tokens are combined into a header name (C90 6.8.2, C99 6.10.2)."	
	All tokens, including whitespace, are considered part of the header file name. Macro expansion is not performed on tokens inside the delimiters.
"The nesting limit for `#include` processing (C90 6.8.2, C99 6.10.2)."	
	No limit.
"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)."	
	No.
"The behavior on each recognized non-STDC `#pragma` directive (C90 6.8.6, C99 6.10.6)."	
	See the section on *Pragma Directives*.
"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)."	
	The date and time of translation are always available.

6.13 Library Functions

ISO Standard	Implementation
"Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1)."	
	See the compiler users guide relevant to your target device.
"The format of the diagnostic printed by the `assert` macro (7.2.1.1)."	
	`Assertion failed: expression (file: func: line)`
	`Assertion failed: (message), function function, file file, line line.\n`. The `function function` component is skipped if `__func__` is unavailable.
"The representation of floating-point exception flags stored by the `fegetexceptflag` function (7.6.2.2)."	
"Whether the `feraiseexcept` function raises the inexact exception in addition to the overflow or underflow exception (7.6.2.3)."	
	Unimplemented.
"Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1)."	
	None.
"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)."	

.....continued

ISO Standard	Implementation
	Unimplemented.
"Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1)."	
	None.
"The values returned by the mathematics functions on domain errors (7.12.1)."	
	The `errno` variable is set to `EDOM` on domain errors
"Whether the mathematics functions set `errno` to the value of the macro `ERANGE` on overflow and/or underflow range errors (7.12.1)."	
	Yes
"Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero (7.12.10.1)."	
	The first argument is returned.
"The base-2 logarithm of the modulus used by the `remquo` function in reducing the quotient (7.12.10.3)."	
	Unimplemented.
"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)."	
	Signals are not implemented.
"The null pointer constant to which the macro `NULL` expands (7.17)."	
	`((void*)0)`
"Whether the last line of a text stream requires a terminating new-line character (7.19.2)."	
	Streams are not implemented.
"Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2)."	
	Streams are not implemented.
"The number of null characters that may be appended to data written to a binary stream (7.19.2)."	
	Streams are not implemented.
"Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3)."	
	Streams are not implemented.
"Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3)."	
	Streams are not implemented.
"The characteristics of file buffering (7.19.3)."	
	File handling is not implemented.
"Whether a zero-length file actually exists (7.19.3)."	
	File handling is not implemented.
"The rules for composing valid file names (7.19.3)."	
	File handling is not implemented.
"Whether the same file can be open multiple times (7.19.3)."	
	File handling is not implemented.
"The nature and choice of encodings used for multibyte characters in files (7.19.3)."	
	File handling is not implemented.
"The effect of the remove function on an open file (7.19.4.1)."	
	File handling is not implemented.
"The effect if a file with the new name exists prior to a call to the rename function (7.19.4.2)."	
	File handling is not implemented.
"Whether an open temporary file is removed upon abnormal program termination (7.19.4.3)."	
	File handling is not implemented.
"What happens when the `tmpnam` function is called more than `TMP_MAX` times (7.19.4.4)."	

.....continued

ISO Standard	Implementation
	File handling is not implemented.
"Which changes of mode are permitted (if any) and under what circumstances (7.19.5.4)."	
	File handling is not implemented.
"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)."	
	NaN is printed as `nan`, with no char sequence printed. Infinity is printed as `[-/+]inf`.
"The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1)."	
	Functionally equivalent to `%lx`.
"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)."	
	Streams are not implemented.
"The set of sequences matched by the `%p` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2)."	
	Streams are not implemented.
"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)."	
	Streams are not implemented.
"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)."	
	No meaning is attached to the sequence.
"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)."	
	No.
"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)."	
	The requested size is bumped to 1 byte. If this can be successfully allocated, a pointer to the space is returned; otherwise `NULL` is returned.
"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)."	
	Streams are not implemented.
"The termination status returned to the host environment by the abort function (7.20.4.1)."	
	The host environment is application defined.
"The value returned by the system function when its argument is not a Null Pointer (7.20.4.5)."	
	The host environment is application defined.
"The local time zone and Daylight Saving Time (7.23.1)."	
	Application defined.
"The range and precision of times representable in `clock_t` and `time_t` (7.23)"	
	The `time_t` type is used to hold a number of seconds and is defined as a `long` type; `clock_t` is defined as an `unsigned long`.
"The era for the clock function (7.23.2.1)."	
	Application defined.
"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)."	
"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)."	
	No.
"Whether the functions in `<math.h>` honor the Rounding Direction mode (F.9)."	
	The rounding mode is not forced.

6.14 Architecture

ISO Standard	Implementation
"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)."	
	See the <float.h>, <limits.h> and <stdint.h> sections in the *Microchip Unified Standard Library Reference Guide*.
"The number, order and encoding of bytes in any object when not explicitly specified in the standard (C99 6.2.6.1)."	
	Little endian, populated from Least Significant Byte first.
"The value of the result of the `sizeof` operator (C90 6.3.3.4, C99 6.5.3.4)."	

7. Common C Interface

The Common C Interface (CCI) is available with all MPLAB XC C compilers and is designed to enhance code portability between these compilers. For example, CCI-conforming code would make it easier to port from a PIC18 MCU using the MPLAB XC8 C compiler to a PIC32 MCU using the MPLAB XC32 C/C++ Compiler.

The CCI assumes that your source code already conforms to the C90 or C99 Language Standard. If you intend to use the CCI, it is your responsibility to write code that conforms. Legacy projects will need to be migrated to achieve conformance. A compiler option must also be set to ensure that the operation of the compiler is consistent with the interface when the project is built.

7.1 Background - The Desire for Portable Code

All programmers want to write portable source code.

Portability means that the same source code can be compiled and run in a different execution environment than that for which it was written. Rarely can code be one hundred percent portable, but the more tolerant it is to change, the less time and effort it takes to have it running in a new environment.

Embedded engineers typically think of code portability as being across target devices, but this is only part of the situation. The same code could be compiled for the same target but with a different compiler. Differences between those compilers might lead to the code failing at compile time or runtime, so this must be considered as well.

You can only write code for one target device and only use one brand of compiler; but if there is no regulation of the compiler's operation, simply updating your compiler version can change your code's behavior.

Code must be portable across targets, tools, and time to be truly flexible.

Clearly, this portability cannot be achieved by the programmer alone, since the compiler vendors can base their products on different technologies, implement different features and code syntax, or improve the way their product works. Many a great compiler optimization has broken many an unsuspecting project.

Standards for the C language have been developed to ensure that change is managed and code is more portable. The International Standards Organization (ISO) publishes standards for many disciplines, including programming languages. Standards such as the ISO/IEC 9899:1999 are universally adopted for the C programming language.

7.1.1 The C Language Standard

A C Language Standard, such as ISO/IEC 9899:1999 Standard (C99), has to reconcile two opposing goals: freedom for compilers vendors to target new devices and improve code generation, against the known functional operation of source code for programmers. If both goals can be met, source code can be made portable.

The C Language Standards are implemented as a set of rules that detail not only the syntax that a conforming C program must follow, but the semantic rules by which that program will be interpreted. Thus, for a compiler to conform to the standard, it must ensure that a conforming C program functions as described by the standard.

Language Standards describe *implementation* as the set of tools and the runtime environment on which the code will run. If any of these change; for example, you build for and run on a different target device, or if you update the version of the compiler you use to build, then you are using a different implementation.

The standards uses the term *behavior* to mean the external appearance or action of the program. This has nothing to do with how a program is encoded.

Since the Language Standard is trying to achieve goals that could be construed as conflicting, some specifications appear somewhat vague. For example, C standards states that an `int` type must be able to hold at least a 16-bit value, but it does not go as far as saying what the size of an `int` actually is; and the action of right-shifting a signed integer can produce different results on different implementations; yet these different results are still compliant with the standard.

If a standard is too strict, device architectures cannot allow the compiler to conform. But if it is too weak, programmers would see wildly differing results within different compilers and architectures, causing the standard to lose its effectiveness.

For example, the mid-range PIC® microcontrollers do not have a data stack. Because a compiler targeting this device cannot implement recursion, it (strictly speaking) cannot conform to the C Language Standard. This example illustrates a situation in which the standard is too strict for mid-range devices and tools.

The standard organizes source code whose behavior is not fully defined into groups that include the following behaviors:

Implementation-defined behavior	This is unspecified behavior in which each implementation documents how the choice is made.
Unspecified behavior	The standard provides two or more possibilities and imposes no further requirements on which possibility is chosen in any particular instance.
Undefined behavior	This is behavior for which the standard imposes no requirements.

Code that strictly conforms to a Language Standard does not produce output that is dependent on any unspecified, undefined, or implementation-defined behavior. The size of an `int`, which was used as an example earlier, falls into the category of behavior that is defined by implementation. That is to say, the size of an `int` is defined by which compiler is being used, how that compiler is being used, and the device that is being targeted.

The MPLAB XC compilers are freestanding implementations that conform 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.

For freestanding implementations (or for what are typically called embedded applications), the standard allows non-standard extensions to the language, but obviously does not enforce how they are specified or how they work. When working so closely to the device hardware, a programmer needs a means of specifying device setup and interrupts, as well as utilizing the often complex world of small-device memory architectures. This cannot be offered by the standard in a consistent way.

While the C Language Standards provides a mutual understanding for programmers and compiler vendors, programmers need to consider the implementation-defined behavior of their tools and the probability that they may need to use extensions to the C language that are non-standard. Both of these circumstances can have an impact on code portability.

7.1.2 The Common C Interface

The Common C Interface (CCI) supplements the C Language Standard, such as ISO/IEC 9899:1999, and makes it easier for programmers to achieve consistent outcomes on all Microchip devices when using MPLAB XC C compilers.

It delivers the following improvements, all designed with portability in mind.

Refinement of the C Language Standard	The CCI documents specific behavior for some code in which actions are implementation-defined behavior under the C Language Standard. For example, the result of right-shifting a signed integer is fully defined by the CCI. Note that many implementation-defined items that closely couple with device characteristics, such as the size of an `int`, are not defined by the CCI.

Consistent syntax for non-standard extensions	The CCI non-standard extensions are mostly implemented using keywords with a uniform syntax. They replace keywords, macros and attributes that are the native compiler implementation. The interpretation of the keyword can differ across each compiler, and any arguments to the keywords can be device specific.
Coding guidelines	The CCI can indicate advice on how code should be written so that it can be ported to other devices or compilers. While you may choose not to follow the advice, it will not conform to the CCI.

7.2 Using the CCI

The CCI allows enhanced portability by refining implementation-defined behavior and standardizing the syntax for extensions to the language.

The CCI is something you choose to follow and put into effect, thus it is relevant for new projects, although you can choose to modify existing projects so they conform.

For your project to conform to the CCI, you must do the following things.

- **Enable the CCI**
  Select the MPLAB X IDE widget **Use CCI Syntax** in your project, or use the command-line option that is equivalent.
- **Include <xc.h> in every module**
  Some CCI features are only enabled if this header is seen by the compiler.
- **Ensure C Language Standard compliance**
  Code that does not conform to the C Language Standard does not conform to the CCI.
- **Observe refinements to the Language Standard by the CCI**
  Some implementation-defined behavior is defined explicitly by the CCI.
- **Use the CCI extensions to the language**
  Use the CCI extensions rather than the native language extensions.

The next sections detail specific items associated with the CCI. These items are segregated into those that refine the standard, those that deal with the C Language Standard extensions, and other miscellaneous compiler options and usage. Guidelines are indicated with these items.

If any implementation-defined behavior or any non-standard extension is not discussed in this document, then it is not part of the CCI. For example, GCC case ranges, label addresses and 24-bit `short long` types are not part of the CCI. Programs which use these features do not conform to the CCI. The compiler may issue a warning or error to indicate a non-CCI feature has been used and the CCI is enabled.

7.3 C Language Standard Refinement

The following topics describe how the CCI refines the implementation-defined behaviors outlined in the C Langauge Standard.

7.3.1 Source File Encoding

Under the CCI, a source file must be written using characters from the 7-bit ASCII set. Lines can be terminated using a *line feed* `(\n)` or *carriage return* `(\r)` that is immediately followed by a *line feed*. Escaped characters can be used in character constants or string literals to represent extended characters that are not in the basic character set.

Example

The following shows a string constant being defined that uses escaped characters.

```
const char myName[] = "Bj\370rk\n";
```

Differences

All compilers have used this character set.

Migration to the CCI

No action required.

7.3.2 The Prototype for `main`

The prototype for the `main()` function is:

```
int main(void);
```

Example

The following shows an example of how `main()` might be defined:

```
int main(void)
{
  while(1)
    process();
}
```

Differences

When targeting PIC MCUs using MPLAB XC8 a `void` return type for this function has been assumed.

Migration to the CCI

Each program has one definition for the `main()` function. Confirm the return type for `main()` in all projects previously compiled for 8-bit targets.

7.3.3 Header File Specification

Header file specifications that use directory separators do not conform to the CCI.

Example

The following example shows two conforming include directives.

```
#include <usb_main.h>
```

```
#include "global.h"
```

Differences

Header file specifications that use directory separators have been allowed in previous versions of all compilers. Compatibility problems arose when Windows-style separators "\" were used and the code was compiled under other host operating systems. Under the CCI, no directory separators should be used.

Migration to the CCI

Any `#include` directives that use directory separators in the header file specifications should be changed. Remove all but the header file name in the directive. Add the directory path to the compiler's include search path or MPLAB X IDE equivalent. This will force the compiler to search the directories specified with this option.

For example, the following code:

```
#include <inc/lcd.h>
```

should be changed to:

```
#include <lcd.h>
```

and the path to the `inc` directory added to the compiler's header search path in your MPLAB X IDE project properties, or on the command-line as follows:

```
-Ilcd
```

7.3.4 Include Search Paths

When you include a header file under the CCI, the file should be discoverable in the paths searched by the compiler that are detailed below.

Header files specified in angle bracket delimiters < > should be discoverable in the search paths that are specified by `-I` options (or the equivalent MPLAB X IDE option), or in the standard compiler `include` directories. The `-I` options are searched in the order in which they are specified.

Header files specified in quote characters " " should be discoverable in the current working directory or in the same directories that are searched when the header files are specified in angle bracket delimiters (as above). In the case of an MPLAB X project, the current working directory is the directory in which the C source file is located. If unsuccessful, the search paths should be to the same directories searched when the header file is specified in angle bracket delimiters.

Any other options to specify search paths for header files do not conform to the CCI.

Example

If including a header file, as in the following directive:

```
#include "myGlobals.h"
```

the header file should be locatable in the current working directory, or the paths specified by any `-I` options, or the standard compiler directories. A header file being located elsewhere does not conform to the CCI.

Differences

The compiler operation under the CCI is not changed. This is purely a coding guideline.

Migration to the CCI

Remove any option that specifies header file search paths other than the `-I` option (or the equivalent MPLAB X IDE option), and use the `-I` option in place of this. Ensure the header file can be found in the directories specified in this section.

7.3.5 The Number of Significant Initial Characters in an Identifier

At least the first 255 characters in an identifier (internal and external) are significant. This extends upon the requirement of the C Language Standard, which states a lower number of significant characters are used to identify an object.

Example

The following example shows two poorly named variables, but names that are considered unique under the CCI.

```
int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningFast;

int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningSlow;
```

Differences

When using MPLAB XC8 for PIC targets and in C90 mode, only 31 significant characters have been accepted by default, but an option allowed this to be extended. When building in C99 mode for these devices, there has been no limit on the number of significant characters. There has also been no character limit when using MPLAB XC8 to target AVR MCUs, or when using any other compiler.

Migration to the CCI

No action required. You can take advantage of the less restrictive naming scheme.

7.3.6 Sizes of Types

The sizes of the basic C types, for example `char`, `int` and `long`, are *not* fully defined by the CCI. These types, by design, reflect the size of registers and other architectural features in the target

device. They allow the device to efficiently access objects of this type. The C Language Standard does, however, indicate minimum requirements for these types, as specified in <limits.h>.

If you need fixed-size types in your project, use the types defined in <stdint.h>, for example, uint8_t or int16_t. These types are consistently defined across all XC compilers, even outside of the CCI.

Essentially, the C language offers a choice of two groups of types: those that offer sizes and formats that are tailored to the device you are using; or those that have a fixed size, regardless of the target.

Example

The following example shows the definition of a variable, native, whose size will allow efficient access on the target device; and a variable, fixed, whose size is clearly indicated and remains fixed, even though it may not allow efficient access on every device.

```
int native;
int16_t fixed;
```

Differences

This is consistent with previous types implemented by the compiler.

Migration to the CCI

If you require a C type that has a fixed size, regardless of the target device, use one of the types defined by <stdint.h>.

7.3.7 Plain char Types

The type of a plain char is unsigned char. It is generally recommended that all definitions for the char type explicitly state the signedness of the object.

Example

The following example:

```
char foobar;
```

defines an unsigned char object called foobar.

Differences

When targeting PIC MCUs, the MPLAB XC8 compiler has always treated plain char as an unsigned type. When targeting AVR MCUs, the compiler has used signed char as the default plain char type. Use of signed char has been true for MPLAB XC16, XC-DSC, and XC32. The –funsigned-char option on these compilers changes the default type to be unsigned char.

Migration to the CCI

Any definition of an object defined as a plain char needs review. Any plain char that was intended to be a signed quantity should be replaced with an explicit definition, for example:

```
signed char foobar;
```

which is the recommended method of defining such an object.

You can alternatively use the –funsigned-char option to change the type of plain char.

7.3.8 Signed Integer Representation

The value of a signed integer is determined by taking the two's complement of the integer.

Example

The following shows a variable, test, that is assigned the value -28 decimal.

```
signed char test = 0xE4;
```

Differences

All compilers have represented signed integers in the way described in this section.

Migration to the CCI

No action required.

7.3.9 Integer Conversion

When converting an integer type to a signed integer of insufficient size, the original value is truncated from the most-significant bit to accommodate the target size.

Example

The following shows an assignment of a value that is truncated.

```
signed char destination;
unsigned int source = 0x12FE;
destination = source;
```

Under the CCI, the value of `destination` after the alignment is -2 (that is, the bit pattern 0xFE).

Differences

All compilers have performed integer conversion in an identical fashion to that described in this section.

Migration to the CCI

No action required.

7.3.10 Bitwise Operations on Signed Values

Bitwise operations on signed values act on the two's complement representation, including the sign bit. See also Right-Shifting Signed Values.

Example

The following shows an example of a negative quantity involved in a bitwise AND operation.

```
signed char output, input = -13;
output = input & 0x7E;
```

Under the CCI, the value of output after the assignment is 0x72.

Differences

All compilers have performed bitwise operations in an identical fashion to that described in this section.

Migration to the CCI

No action required.

7.3.11 Right-Shifting Signed Values

Right-shifting a signed value will involve sign extension. This will preserve the sign of the original value.

Example

The following example shows a negative quantity involved in a right-shift operation.

```
signed char output, input = -13;

output = input >> 3;
```

Under the CCI, the value of `output` after the assignment is -2 (that is, the bit pattern 0xFE).

Differences

All compilers have performed right-shifting as described in this section.

Migration to the CCI

No action required.

7.3.12 Conversion of Union Member Accessed Using Member with Different Type

If a union defines several members of different types and you use one member identifier to try to access the contents of another (whether any conversion is applied to the result) is implementation-defined behavior in the standard. In the CCI, no conversion is applied and the bytes of the union object are interpreted as an object of the type of the member being accessed, without regard for alignment or other possible invalid conditions.

Example

The following shows an example of a union defining several members.

```
union {
    signed char code;
    unsigned int data;
    float offset;
} foobar;
```

Code that attempts to extract `offset` by reading `data` is not guaranteed to read the correct value.

```
float result;
result = foobbar.data;
```

Differences

All compilers have not converted union members accessed via other members.

Migration to the CCI

No action required.

7.3.13 Default Bit-field `int` Type

The type of a bit-field specified as a plain `int` is identical to that of one defined using `unsigned int`. This is quite different from other objects where the types `int`, `signed` and `signed int` are synonymous. It is recommended that the signedness of the bit-field be explicitly stated in all bit-field definitions.

Example

The following shows an example of a structure tag containing bit-fields that are unsigned integers and with the size specified.

```
struct OUTPUTS {
    int direction :1;
    int parity    :3;
    int value     :4;
};
```

Differences

When targeting PIC devices, the MPLAB XC8 compiler has issued a warning if an `int` type was specified for bit-fields, and it has instead implemented the bit-field with an `unsigned int` type. For other devices, this compiler has implemented bit-fields defined using `int` as having a `signed int` type, unless the option `-funsigned-bitfields` was specified. Use of a `signed int` type is also true for the MPLAB XC16, XC-DSC, and XC32 compilers.

Migration to the CCI

Any code that defines a bit-field with the plain `int` type should be reviewed. If the intention was for these to be signed quantities, then the type of these should be changed to `signed int`. In the following example:

```
struct WAYPT {
    int log          :3;
    int direction    :4;
};
```

the bit-field type should be changed to `signed int`, as in:

```
struct WAYPT {
    signed int log       :3;
    signed int direction :4;
};
```

7.3.14 Bit-Fields Straddling a Storage Unit Boundary

The standard indicates that implementations can determine whether bit-fields cross a storage unit boundary. In the CCI, bit-fields do not straddle a storage unit boundary; a new storage unit is allocated to the structure, and padding bits fill the gap.

Note that the size of a storage unit differs with each compiler and target device, as this is based on the size of the base data type (for example, `int`) from which the bit-field type is derived. For MPLAB XC8, this unit is 8-bits in size; for MPLAB XC16, it is 16 bits; and for MPLAB XC32, it is 32 bits in size. For MPLAB XC-DSC compiler, the size depends on the device used.

Example

The following shows a structure containing bit-fields being defined.

```
struct {
        unsigned first  : 6;
        unsigned second : 6;
} order;
```

Under the CCI and using MPLAB XC8, the storage allocation unit is byte sized. The bit-field `second` is allocated a new storage unit since there are only 2 bits remaining in the first storage unit in which `first` is allocated. The size of this structure, `order`, is 2 bytes.

Differences

This allocation is identical with that used by all previous compilers.

Migration to the CCI

No action required.

7.3.15 The Allocation Order of Bit-Field

The memory ordering of bit-fields into their storage unit is not specified by the C Language Standard. In the CCI, the first bit defined is the least significant bit (LSb) of the storage unit in which it is allocated.

Example

The following shows a structure containing bit-fields being defined.

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

The bit-field `lo` is assigned the least significant bit of the storage unit assigned to the structure `foo`. The bit-field `mid` is assigned the next 6 least significant bits, and `hi`, the most significant bit of that same storage unit byte.

Differences

This is identical with the previous operation of all compilers.

Migration to the CCI

No action required.

7.3.16 The NULL Macro

The NULL macro is defined by `<stddef.h>`; however, its definition is implementation-defined. Under the CCI, the definition of `NULL` is the expression `(0)`.

Example

The following shows a pointer being assigned a null pointer constant via the `NULL` macro.

```
int * ip = NULL;
```

The value of `NULL`, `(0)`, is implicitly converted to the destination type.

Differences

MPLAB XC32 has assigned `NULL` the expression `((void *)0)`. The same is true for MPLAB XC8 when targeting AVR MCUs.

Migration to the CCI

No action required.

7.3.17 Floating-Point Sizes

Under the CCI, floating-point types must not be smaller than 32 bits in size.

Example

The following shows the definition for `outY`, which is at least 32-bit in size.

```
float outY;
```

Differences

The MPLAB XC8 when targeting PIC MCUs and building with C90 language standard has allowed the use of 24-bit `float` and `double` types.

Migration to the CCI

When using MPLAB XC8, the `float` and `double` type will be made 32 bits in size once the CCI is enabled, regardless of target device or operating mode. Review any source code that assumes a `float` or `double` type is 24 bits in size.

No migration is required for other compilers.

7.4 C Language Standard Extensions

The following topics describe how the CCI provides device-specific extensions to the C Language Standard.

7.4.1 Generic Header File

A single header file `<xc.h>` must be used to declare all compiler- and device-specific types and SFRs. You *must* include this file into every module to conform with the CCI. Some CCI definitions depend on this header being seen.

Example

The following shows this header file being included, thus allowing conformance with the CCI, as well as allowing access to SFRs.

```
#include <xc.h>
```

Differences

Although device-specific headers are shipped with the compilers, always include the top-level `<xc.h>` header.

Early compilers for 8-bit PIC devices used `<htc.h>` as the equivalent header. Other compilers used a variety of device-related headers to do the same job.

Migration to the CCI

Change for example:

```
#include <p32xxxx.h>
#include <p30fxxxx.h>
#include <pic16lf18324.h> /* or */
#include "p30f6014.h"
```

to:

```
#include <xc.h>
```

7.4.2 Absolute Addressing

Variables and functions can be placed at an absolute address by using the `__at()` construct. Stack-based (`auto` and parameter) variables cannot use the `__at()` specifier.

Example

The following shows two variables and a function being made absolute.

```
int scanMode __at(0x200);
const char keys[] __at(124) = { 'r', 's', 'u', 'd'};

__at(0x1000) int modify(int x) {
    return x * 2 + 3;
}
```

Differences

The legacy syntax used by MPLAB XC8 when targeting PIC MCUs has been an `@` symbol to specify an absolute address.

When targeting AVR MCUs, MPLAB XC8 has used the `address` attribute to specify an object's address. This attribute has also been used by the other compilers.

Migration to the CCI

Avoid making objects and functions absolute if possible.

If any source code uses the legacy MPLAB XC8 syntax, for example:

```
int scanMode @ 0x200;
```

change this to:

```
int scanMode __at(0x200);
```

When building for AVR MCUs with the MPLAB XC8 compiler or when using any other compiler, change code, for example, from:

```
int scanMode __attribute__((address(0x200)));
```

to:

```
int scanMode __at(0x200);
```

Caveats

If building for PIC device using MPLAB XC8 and the `__at()` and `__section()` specifiers have both been applied to an object, the `__section()` specifier is currently ignored.

The MPLAB XC32 compiler for PIC32C/SAM devices supports only 4-byte aligned absolute addresses.

7.4.3 Far Objects and Functions

The `__far` qualifier can be used to indicate that variables or functions are located in 'far memory'. Exactly what constitutes far memory is dependent on the target device, but it is typically memory that requires more complex code to access. Expressions involving far-qualified objects usually generate slower and larger code.

Use the native keywords discussed in the Differences section to look up information on the semantics of this qualifier.

Some devices may not have such memory implemented, in which case, use of this qualifier is ignored. Stack-based (`auto` and parameter) variables cannot use the `__far` specifier.

Example

The following shows a variable and function qualified using `__far`.

```
__far int serialNo;

__far int ext_getCond(int selector);
```

Differences

When targeting PIC devices, the MPLAB XC8 compiler has allowed the `far` specifier to be used with variables. This specifier has not been allowed with functions. When targeting AVR devices, MPLAB XC8 has not implemented any "far" memory.

The MPLAB XC16 and XC-DSC compilers have used the `far` attribute with both variables and functions.

The MPLAB XC32 compiler has used the `far` attribute with functions only.

Migration to the CCI

When targeting PIC devices with the MPLAB XC8 compiler, change any occurrence of the `far` qualifier to `__far`, for example, from:

```
far char template[20];
```

to:

```
__far char template[20];
```

When using MPLAB XC16, XC-DSC or XC32 compilers, change any occurrence of the `far` attribute to the `__far` specifier, for example, from:

```
void bar(void) __attribute__ ((far));

int tblIdx __attribute__ ((far));
```

to:

```
void __far bar(void);

int __far tblIdx;
```

Caveats

None.

7.4.4 Near Objects

The `__near` qualifier can be used to indicate that variables or functions are located in 'near memory'. Exactly what constitutes near memory is dependent on the target device, but it is typically memory that can be accessed with less complex code. Expressions involving near-qualified objects generally are faster and result in smaller code.

Use the native keywords discussed in the Differences section to look up information on the semantics of this qualifier.

Some devices may not have such memory implemented, in which case, use of this qualifier is ignored. Stack-based (`auto` and parameter) variables cannot use the `__near` specifier.

Example

The following shows a variable and function qualified using `__near`.

```
__near int serialNo;

__near int ext_getCond(int selector);
```

Differences

When targeting PIC devices, the MPLAB XC8 compiler has allowed the `near` specifier to be used with variables. This specifier has not been allowed with functions. When targeting AVR devices, MPLAB XC8 has not implemented any "near" memory.

The MPLAB XC16 and XC-DSC compilers have used the `near` attribute with both variables and functions.

The MPLAB XC32 compiler has used the `near` attribute for functions, only.

Migration to the CCI

When targeting PIC devices with the MPLAB XC8 compiler, change any occurrence of the `near` qualifier to `__near`, for example, from:

```
near char template[20];
```

to:

```
__near char template[20];
```

When using the other compilers, change any occurrence of the `near` attribute to the `__near` specifier, for example, from:

```
void bar(void) __attribute__ ((near));

int tblIdx __attribute__ ((near));
```

to

```
void __near bar(void);

int __near tblIdx;
```

Caveats

None.

7.4.5 Persistent Objects

The `__persistent` qualifier can be used to indicate that variables should not be cleared by the runtime startup code.

Use the native keywords discussed in the Differences section to look up information on the semantics of this qualifier.

Example

The following shows a variable qualified using `__persistent`.

```
__persistent int serialNo;
```

Differences

When targeting PIC devices, the MPLAB XC8 compiler has allowed use of the `persistent` specifier to indicate that variables should not be cleared at startup. When targeting AVR devices, this compiler has allowed the use of the `persistent` attribute with variables to indicate they were not to be cleared.

The MPLAB XC16, XC-DSC and XC32 compilers have also used the `persistent` attribute for this purpose.

Migration to the CCI

When building with the MPLAB XC8 compiler for PIC MCUs, change any occurrence of the `persistent` qualifier to `__persistent`, for example, from:

```
persistent char template[20];
```

to:

```
__persistent char template[20];
```

When building with MPLAB XC16, XC-DSC or XC32 compilers, or with the MPLAB XC8 compiler for AVR MCUs, change any occurrence of the `persistent` attribute to the `__persistent` specifier, for example, from:

```
int tblIdx __attribute__ ((persistent));
```

to

```
int __persistent tblIdx;
```

Caveats

None.

7.4.6 X and Y Data Objects

The `__xdata` and `__ydata` qualifiers can be used to indicate that variables are located in special memory regions. Exactly what constitutes X and Y memory is dependent on the target device, but it is typically memory that can be accessed independently on separate buses. Such memory is often required for some DSP instructions.

Use the native keywords discussed in the Differences section to look up information on the semantics of these qualifiers.

Some devices may not have such memory implemented; in which case, use of these qualifiers is ignored.

Example

The following shows a variable qualified using `__xdata`, as well as another variable qualified with `__ydata`.

```
__xdata char data[16];

__ydata char coeffs[4];
```

Differences

The MPLAB XC16 and XC-DSC compilers have used the `xmemory` and `ymemory` space attribute with variables.

Equivalent specifiers have never been defined for any other compiler.

Migration to the CCI

For MPLAB XC16 and XC-DSC compilers, change any occurrence of the `space` attributes `xmemory` or `ymemory` to `__xdata`, or `__ydata` respectively, for example, from:

```
char __attribute__((space(xmemory)))template[20];
```

to:

```
__xdata char template[20];
```

Caveats

None.

7.4.7 Banked Data Objects

The `__bank(num)` qualifier can be used to indicate that variables are located in a particular data memory bank. The number, *num*, represents the bank number. Exactly what constitutes banked memory is dependent on the target device, but it is typically a subdivision of data memory to allow for assembly instructions with a limited address width field.

Use the native keywords discussed in the Differences section to look up information on the semantics of these qualifiers.

Some devices may not have banked data memory implemented, in which case, use of this qualifier is ignored. The number of data banks implemented will vary from one device to another.

Example

The following shows a variable qualified using `__bank()`.

```
__bank(0) char start;

__bank(3) char stop;
```

Differences

When targeting PIC devices, the MPLAB XC8 compiler has used the four qualifiers `bank0`, `bank1`, `bank2` and `bank3` to indicate memory placement in a specific data bank.

Equivalent specifiers have never been defined for any other compiler.

Migration to the CCI

When building for PIC devices using MPLAB XC8, change any occurrence of the `bank`*num* qualifiers to `__bank(`*num*`)`, for example, from:

```
bank2 int logEntry;
```

to:

```
__bank(2) int logEntry;
```

Caveats

Only banks 0 through 3 are currently supported.

7.4.8 Alignment of Objects

The `__align(`*alignment*`)` specifier can be used to indicate that variables must be aligned on a memory address that is a multiple of the alignment specified. The alignment term must be a power of 2. Positive values request that the object's start address be aligned.

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

Example

The following shows variables qualified using `__align()` to ensure they end on an address that is a multiple of 8, and start on an address that is a multiple of 2, respectively.

```
__align(-8) int spacer;
__align(2) char coeffs[6];
```

Differences

The MPLAB XC16, XC-DSC and XC32 compilers have used the `aligned` attribute with variables. This has also been true when targeting AVR MCUs with MPLAB XC8 An alignment feature has never been available for PIC devices when using the MPLAB XC8 compiler.

Migration to the CCI

When building for AVR MCUs with the MPLAB XC8 compiler or when using any other compiler, change any occurrence of the `aligned` attribute to the `__align` specifier, for example, from:

```
char __attribute__((aligned(4)))mode;
```

to:

```
__align(4) char mode;
```

Caveats

This feature is not needed nor implemented when targeting PIC devices using MPLAB XC8.

When targeting PIC32C/SAM devices with MPLAB XC32 only positive alignment values are supported.

7.4.9 EEPROM Objects

The `__eeprom` qualifier can be used to indicate that variables should be positioned in EEPROM.

Use the native keywords discussed in the Differences section to look up information on the semantics of this qualifier.

Some devices may not implement EEPROM. Use of this qualifier for such devices generates a warning. Stack-based (`auto` and parameter) variables cannot use the `__eeprom` specifier.

Example

The following shows a variable qualified using `__eeprom`.

```
__eeprom int serialNos[4];
```

Differences

When targeting PIC devices, the MPLAB XC8 compiler has used the `eeprom` qualifier to indicate that objects should be placed in EEPROM.

The MPLAB XC16 and XC-DSC compilers have used the `space` attribute to allocate variables to the memory space used for EEPROM.

Migration to the CCI

When using MPLAB XC8 to build for PIC devices, change any occurrence of the `eeprom` qualifier to `__eeprom`, for example, from:

```
eeprom char title[20];
```

to:

```
__eeprom char title[20];
```

When using MPLAB XC16 or XC-DSC compilers, change any occurrence of the `eedata space` attribute to the `__eeprom` specifier, for example, from:

```
int mainSw __attribute__ ((space(eedata)));
```

to:

```
int __eeprom mainSw;
```

Caveats

The MPLAB XC8 compiler only permits the `__eeprom` specifier for those Baseline and Mid-range devices that support this memory. It is not permitted for other 8-bit devices.

7.4.10 Interrupt Functions

The `__interrupt(`*`type`*`)` specifier can be used to indicate that a function is to act as an interrupt service routine. The *`type`* is a comma-separated list of keywords that indicate information about the interrupt function.

The current interrupt types are shown in the following table.

Interrupt type	Description	Compiler
<empty>	Implement the default interrupt function.	
`low_priority`	The interrupt function corresponds to the low priority interrupt source.	MPLAB XC8 - PIC18 only
`high_priority`	The interrupt function corresponds to the high priority interrupt source.	MPLAB XC8
`save(symbol-list)`	Save on entry and restore on exit the listed symbols.	MPLAB XC16, MPLAB XC-DSC
`irq(`*`irqid`*`)`	Specify the interrupt vector associated with this interrupt.	MPLAB XC8, MPLAB XC16, MPLAB XC-DSC
`altirq(`*`altirqid`*`)`	Specify the alternate interrupt vector associated with this interrupt.	MPLAB XC16, MPLAB XC-DSC
`preprologue(`*`asm`*`)`	Specify assembly code to be executed before any compiler-generated interrupt code.	MPLAB XC16, MPLAB XC-DSC
`shadow`	Allow the ISR to utilize the shadow registers for context switching.	MPLAB XC16, MPLAB XC-DSC
`auto_psv`	The ISR will set the PSVPAG register and restore it on exit.	MPLAB XC16, MPLAB XC-DSC
`no_auto_psv`	The ISR will not set the PSVPAG register.	MPLAB XC16, MPLAB XC-DSC

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

Some devices may not implement interrupts. Use of this qualifier for such devices generates a warning. If the argument to the `__interrupt()` specifier does not make sense for the target device, a warning or error is issued by the compiler.

Example

The following shows a function qualified using `__interrupt` for an 8-bit PIC device.

```
__interrupt(low_priority) void getData(void) {
    if (TMR0IE && TMR0IF) {
        TMR0IF=0;
        ++tick_count;
    }
}
```

Differences

The legacy interrupt function syntax used by MPLAB XC8 when targeting PIC MCUs has been the `interrupt` specifier and optionally the `low_priority` specifier. When targeting AVR devices, the MPLAB XC8 compiler has used the `ISR()` macro to define interrupt functions.

The MPLAB XC16, XC-DSC, and XC32 compilers have used the `interrupt` attribute to define interrupt functions.

Migration to the CCI

When building with the MPLAB XC8 compiler for PIC MCUs, change any instance of the `interrupt` specifier to `__interrupt`, for example, from:

```
void interrupt low_priority tckI(void)
```

to:

```
void __interrupt(low_priority) tckI(void)
```

When building with the MPLAB XC8 compiler for AVR MCUs, change any instance of the `ISR()` macro, for example, from:

```
ISR(TIMER1_OVF_vect)
```

to:

```
void __interrupt(TIMER1_OVF_vect_num) spi_Isr(void)
```

When building with XC16 or XC-DSC compilers, change any occurrence of the `interrupt` attribute, for example, from:

```
void _attribute_((interrupt(auto_psv,irq(52)))) _T1Interrupt(void);
```

to:

```
void __interrupt(auto_psv,irq(52))) _T1Interrupt(void);
```

For MPLAB XC32, the `__interrupt()` keyword takes two parameters, the vector number and the (optional) IPL value. Change code that uses the `interrupt` attribute, similar to these examples:

```
void __attribute__((vector(0), interrupt(IPL7AUTO), nomips16)) myisr0_7A(void) {}

void __attribute__((vector(1), interrupt(IPL6SRS), nomips16)) myisr1_6SRS(void) {}

/* Determine IPL and context-saving mode at runtime */
void __attribute__((vector(2), interrupt(), nomips16)) myisr2_RUNTIME(void) {}
```

to:

```
void __interrupt(0,IPL7AUTO) myisr0_7A(void) {}

void __interrupt(1,IPL6SRS) myisr1_6SRS(void) {}

/* Determine IPL and context-saving mode at runtime */
void __interrupt(2) myisr2_RUNTIME(void) {}
```

Caveats

None.

7.4.11 Packing Objects

The `__pack` specifier can be used to indicate that structures should not use memory gaps to align structure members, or that individual structure members should not be aligned.

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

Some compilers cannot pad structures with alignment gaps for some devices, and use of this specifier for such devices is ignored.

Example

The following shows a structure qualified using __pack, as well as a structure where one member has been explicitly packed.

```
struct DATAPOINT {
    unsigned char type;
    int value;
} __pack x_point;
struct LINETYPE {
    unsigned char type;
    __pack int start;
    long total;
} line;
```

Differences

The __pack specifier is a new CCI specifier that can be used with PIC devices and MPLAB XC8. This specifier has no observable effect since the device memory is byte addressable for all data objects. When targeting AVR devices, the MPLAB XC8 compiler has used the packed attribute.

The MPLAB XC16, XC-DSC, and XC32 compilers have used the packed attribute to indicate that a structure member was not aligned with a memory gap.

Migration to the CCI

No migration is required for the MPLAB XC8 compiler when building for PIC MCUs. When building for AVR MCUs or when building with any other compiler, change any occurrence of the packed attribute to the __packed specifier, for example, from:

```
struct DOT
{
    char a;
    int x[2] __attribute__ ((packed));
};
```

to

```
struct DOT
{
    char a;
    __pack int x[2];
};
```

Alternatively, you can pack the entire structure, if required.

Caveats

None.

7.4.12 Indicating Antiquated Objects

The __deprecate specifier can be used to indicate that an object has limited longevity and should not be used in new designs. It is commonly used by the compiler vendor to indicate that compiler extensions or features can become obsolete, or that better features have been developed and should be used in preference.

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

Example

The following shows a function that uses the __deprecate keyword.

```
void __deprecate getValue(int mode)
{
//...
}
```

Differences

When targeting AVR MCUs using the MPLAB XC8 compiler or when using the other compilers, the `deprecated` attribute (note the different spelling to the CCI specifier) has been used to indicate that use of certain objects should be avoided, if possible.

Migration to the CCI

When building with MPLAB XC16, XC-DSC or XC32 compilers, or with the MPLAB XC8 compiler for AVR MCUs, change any occurrence of the `deprecated` attribute to `__deprecate`, for example, from:

```
int __attribute__(deprecated) intMask;
```

to:

```
int __deprecate intMask;
```

Caveats

None.

7.4.13 Assigning Objects to Sections

The `__section()` specifier can be used to indicate that an object should be located in the named section. This is typically used when the object has special and unique linking requirements that cannot be addressed by existing compiler features.

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

Example

The following shows a variable which uses the `__section` keyword.

```
int __section("comSec") commonFlag;
```

Differences

When targeting AVR MCUs using the MPLAB XC8 compiler or when using the other compilers, the `section` attribute has been used to indicate a different destination section name. The `__section()` specifier works in a similar way to the attribute.

Migration to the CCI

When building with MPLAB XC16, XC-DSC or XC32 compilers, or with the MPLAB XC8 compiler for AVR MCUs, change any occurrence of the `section` attribute, for example, from:

```
int __attribute__((section("myVars"))) intMask;
```

to:

```
int __section("myVars") intMask;
```

Caveats

None.

7.4.14 Specifying Configuration Bits

The `#pragma config` directive can be used to program the Configuration bits for a device. The pragma has the form:

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

where *`setting`* is a configuration setting descriptor (for example, `WDT`), *`state`* is a descriptive value (for example, `ON`) and *`value`* is a numerical value.

Use the native keywords discussed in the Differences section to look up information on the semantics of this directive.

Example

The following shows Configuration bits being specified using this pragma.

```
#pragma config WDT=ON, WDTPS = 0x1A
```

Differences

When targeting PIC MCUs, MPLAB XC8 has provided a legacy `__CONFIG()` macro, but more recently has accepted `#pragma config`. When targeting AVR devices, it has used a predefined `FUSES` structure to allow the configuration bits to be specified.

The MPLAB XC16 and XC-DSC compilers have used a number of macros to specify the configuration settings.

The MPLAB XC32 compiler has supported the use of `#pragma config`.

Migration to the CCI

When building for PIC MCUs with MPLAB XC8, change any occurrence of the `__CONFIG()` macro, for example,

```
__CONFIG(WDTEN & XT & DPROT)
```

to the `#pragma config` directive, for example:

```
#pragma config WDTE=ON, FOSC=XT, CPD=ON
```

When building for AVR MCUs with MPLAB XC8, change any occurrence of the `FUSES` structure, for example:

```
#include <avr/io.h>
FUSES = {
    .high = (FUSE_SPIEN)
};
```

to:

```
#pragma config SPIEN=SET
```

For the MPLAB XC16 and XC-DSC compilers, change any occurrence of the `_FOSC()` or `_FBORPOR()` macros attribute, for example, from:

```
_FOSC(CSW_FSCM_ON & EC_PLL16);
```

to:

```
#pragma config FCKSMEM = CSW_ON_FSCM_ON, FPR = ECIO_PLL16
```

No migration is required for 32-bit code.

Caveats

None.

7.4.15 Manifest Macros

The CCI defines the general form for macros that manifest the compiler and target device characteristics. These macros can be used to conditionally compile alternate source code based on the compiler or the target device.

The macros and macro families are details in the following table.

Table 7-1. Manifest Macros Defined by the CCI

Name	Meaning if defined	Example
__XC__	Compiled with an MPLAB XC compiler	__XC__
__CCI__	Compiler is CCI compliant and CCI enforcement is enabled	__CCI__
__XC#__	The specific XC compiler used (# can be 8, 16, _DSC or 32)	__XC32__
__DEVICEFAMILY__	The family of the selected target device	__dsPIC30F__
__DEVICENAME__	The selected target device name	__33ck256mp508__

Example

The following shows code that is conditionally compiled dependent on the device having EEPROM memory.

```
#ifdef __XC_DSC__
void __interrupt(__auto_psv__) myIsr(void)
#else
void __interrupt(low_priority) myIsr(void)
#endif
```

Differences

Some of these CCI macros are new (for example, __CCI__), and others have different names to previous symbols with identical meaning (for example, __33ck256mp508 is now __33ck256mp508__).

Migration to the CCI

Any code that uses compiler-defined macros needs review. Old macros continue to work as expected, but they are not compliant with the CCI.

Caveats

None.

7.4.16 In-Line Assembly

The asm() statement can be used to insert assembly code in-line with C code. The argument is a C string literal that represents a single assembly instruction. Obviously, the instructions contained in the argument are device specific.

Use the native keywords discussed in the Differences section to look up information on the semantics of this statement.

Example

The following shows a MOVLW 8-bit PIC MCU instruction being inserted in-line.

```
asm("MOVLW _foobar");
```

Differences

When targeting PIC devices with MPLAB XC8, the asm() or #asm ... #endasm constructs have been used to insert in-line assembly code.

The MPLAB XC16, XC-DSC and XC32 compilers, as well as the MPLAB XC8 compiler for AVR MCUs use the same syntax as the CCI.

Migration to the CCI

When building with the MPLAB XC8 compiler for PIC MCUs, change any instance of `#asm ... #endasm` so that each instruction in this `#asm` block is placed in its own `asm()` statement, for example, from:

```
#asm
    MOVLW        20
    MOVWF _i
    CLRF        Ii+1
#endasm
```

to:

```
asm("MOVLW        20");
asm("MOVWF _i");
asm("CLRF        Ii+1");
```

No migration is required for MPLAB XC8 when targeting AVR MCUs or when using the other compilers.

Caveats

None.

7.5 Compiler Features

The following item details the compiler options used to control the CCI.

7.5.1 Enabling the CCI

It is assumed that you are using the MPLAB X IDE to build projects that use the CCI. The location of the widget in the **Project Properties** window to enable CCI conformance is shown in the table below.

Table 7-2. Project Property Path to Enable CCI

Compiler	Category	Category Option
MPLAB XC8	XC8 Compiler	Preprocessing and messages > Use CCI Syntax
MPLAB XC16	xc16-gcc	Preprocessing and messages > Use CCI Syntax
MPLAB XC-DSC	xc-dsc-gcc	Preprocessing and messages > Use CCI Syntax
MPLAB XC32	xc32-gcc, xc32-g++	Preprocessing and messages > Use CCI Syntax

If you are not using this IDE, use the command-line option `-mext=cci`. For MPLAB XC16, MPLAB XC-DSC and MPLAB XC32, you may also use the option `-mcci`.

Differences

This option has never been implemented previously.

Migration to the CCI

Enable the option.

Caveats

None.

8. Library Functions

The MPLAB XC8 C Compiler is distributed with several libraries, containing functions, macros, and types to assist with program development.

The Standard C libraries are described in the separate *Microchip Unified Standard Library Reference Guide* document, whose content is relevant for all MPLAB XC C compilers.

MPLAB XC8-specific library functions and macros are described in this section.

8.1 Library Example Code

Example code is shown for most library functions. These examples illustrate how the functions can be called and might indicate other aspects of their usage, but they are not necessarily complete nor practical.

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

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

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

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

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

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

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

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

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

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

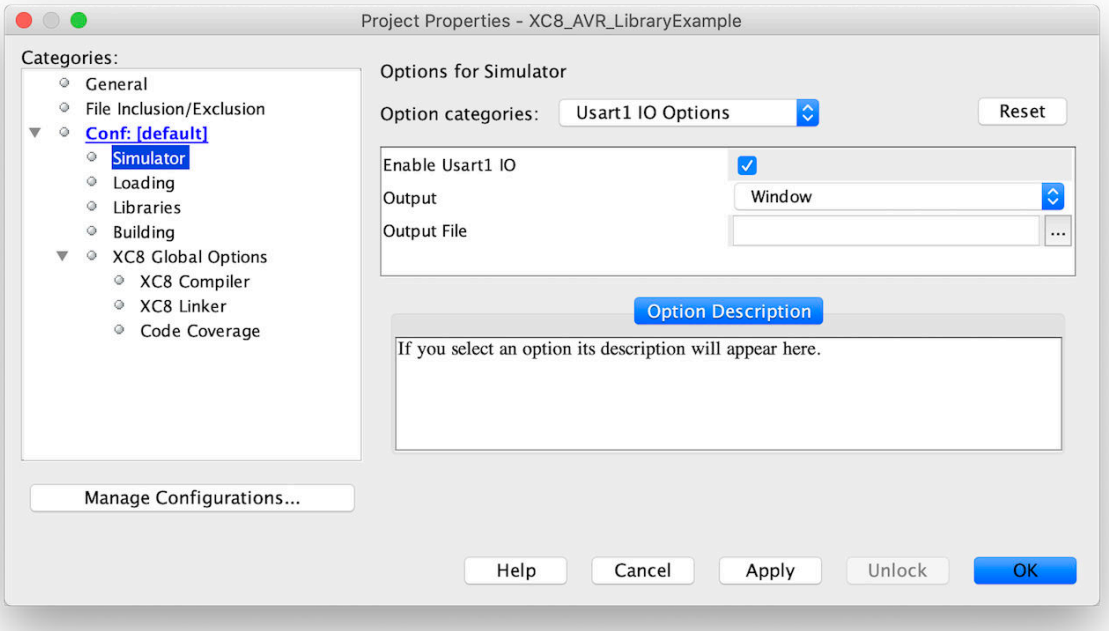
Compiler-specific implementations of the above are discussed in more detail in the following sections.

8.1.1 Example code for 8-bit AVR MCUs

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

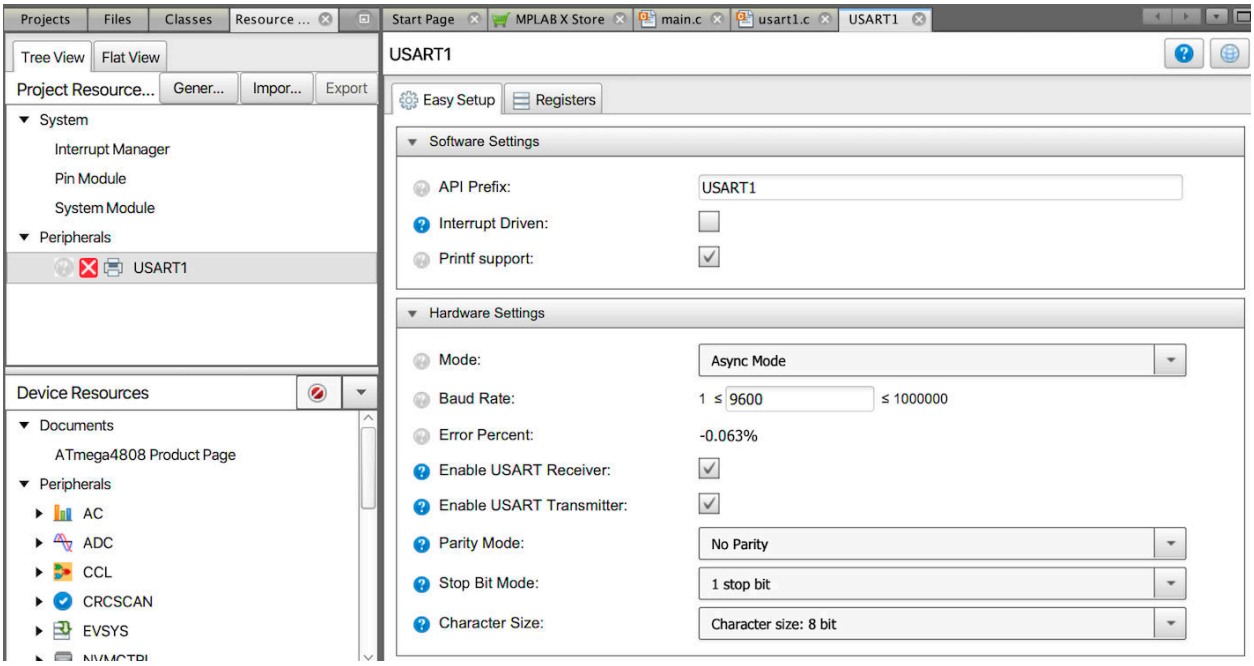
This feature is enabled from **Project properties > Simulator > USARTx IO Options**. You might have a choice of USARTs. Choose the USART that your code will write to. Output can be displayed in a window in the IDE or sent to a file on your host machine, based on the selections you make in the dialog.

Figure 8-1. Enabling the USART IO feature in the MPLAB X IDE



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

Figure 8-2. Initializing the USART using MCC



8.2 <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.

8.2.1 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();
}
```

8.2.2 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));
}
```

8.2.3 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));
}
```

8.2.4 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));
}
```

8.2.5 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);
}
```

8.2.6 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);
}
```

8.2.7 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);
}
```

8.2.8 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);
}
```

8.2.9 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);
}
```

8.2.10 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);
}
```

8.2.11 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();
}
```

8.2.12 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;
}
```

8.2.13 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;
}
```

8.2.14 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);
}
```

8.2.15 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();
}
```

8.2.16 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();
}
```

8.2.17 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();
}
```

8.2.18 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();
}
```

8.3 <cpufunc.h> CPU Related Functions

The header file `cpufunc.h` consists of functions that relate to instruction execution.

8.3.1 _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);
}
```

8.3.2 _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();
}
```

8.4 <delay.h> Delay Functions

The header file `delay.h` consists of functions that generate delays in program execution.

8.4.1 _delay_ms Function

Delay for the specified time.

Include

`<util/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 <util/delay.h>

int main(void)
{
  _delay_ms(20);  // delay for 20 milli seconds
}
```

8.4.2 _delay_us Function

Delay for the specified time.

Include

`<util/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 <util/delay.h>

int main(void)
{
  _delay_us(20);  // delay for 20 micro seconds
}
```

8.5 <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.

8.5.1 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);
}
```

8.5.2 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);
}
```

8.5.3 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);
}
```

8.5.4 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);
}
```

8.5.5 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);
}
```

8.5.6 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);
}
```

8.5.7 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);
}
```

8.5.8 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);
}
```

8.5.9 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);
}
```

8.5.10 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);
}
```

8.5.11 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);
}
```

8.5.12 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);
}
```

8.5.13 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);
}
```

8.5.14 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);
}
```

8.5.15 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);
}
```

8.5.16 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);
}
```

8.5.17 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");
}
```

8.6 <sfr_defs.h>

The header file `sfr_defs.h` consists of functions that assist with accessing special function registers.

8.6.1 _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
}
```

8.6.2 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;
}
```

8.6.3 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;
}
```

8.6.4 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();
}
```

8.6.5 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);
}
```

8.7 <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.

8.7.1 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();
}
```

8.7.2 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();
}
```

8.7.3 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();
}
```

8.7.4 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();
}
```

8.7.5 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();
}
```

8.7.6 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();
}
```

8.8 <xc.h>

The header file `xc.h` consists of macros specific to your target device.

8.8.1 di Macro

A macro that disables interrupts.

Include

`<xc.h>`

Prototype

```
void di(void);
```

Remarks

This macro inserts a `cli` instruction, which disables global interrupts.

Example

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

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

  di();                 // disable interrupts
  val = TCNT1;          // read timer value register; TEMP used internally
  ei();                 // re-enable interrupts
  return val;
}
```

8.8.2 ei Macro

A macro that enables interrupts.

Include

`<xc.h>`

Prototype

```
void ei(void);
```

Remarks

This macro inserts an `sei` instruction, which enables global interrupts.

Example

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

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

  di();                 // disable interrupts
  val = TCNT1;          // read timer value register; TEMP used internally
  ei();                 // re-enable interrupts
  return val;
}
```

8.8.3 NOP Macro

A macro that performs no operation.

Include

`<xc.h>`

Prototype

```
void NOP(void);
```

Remarks

This macro executes a `nop` instruction. A `nop` instruction can be used, for example, to introduce a small delay, to ensure safe execution of code when the device is in a certain state, or as a convenient place to attach a breakpoint for debugging purposes.

Example

```
#include <xc.h>

void delay10(void) {
  for (volatile char i=0; i!= 10; i++)
    NOP();
  return;
}
```

8.9 Built-in Functions

The compiler support a number of built-in functions, which can be used like regular library function calls, but which typically expand to assembly code sequences that are inserted in-line at the point of usage. As they do not require a call-return sequence, they are efficient to use but are more robust than a similarly defined preprocessor macro.

8.9.1 builtin_avr_cli Built-in Function

Inserts a `cli` instruction, which disables global interrupts.

Prototype

```
void __builtin_avr_cli(void);
```

Remarks

The `di()` macro, available once you have included the `<xc.h>` header in your code, performs a similar task.

Example

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

  __builtin_avr_cli(); // disable interrupts
  val = TCNT1;         // read timer value register; TEMP used internally
  __builtin_avr_sei(); // re-enable interrupts
  return val;
}
```

8.9.2 builtin_avr_delay_cycles Built-in Function

Inserts a simple delay instruction sequence.

Prototype

```
void __builtin_avr_delay_cycles(unsigned long tick)
```

Remarks

The built-in will produce code that delays for tick cycles. The tick operand must be a constant expression. It cannot take into account the effect of interrupts that might increase delay time.

Example

```
void toggle(void) {
  PORTB = 0x0F;
```

```
  __builtin_avr_delay_cycles(10);
  PORTB = 0x00;
}
```

8.9.3 builtin_avr_flash_segment Built-in Function

Return the flash segment of a full address.

Prototype

```
char __builtin_avr_flash_segment(const __memx void * mp)
```

Remarks

This built-in function returns the number of the flash segment (the 64 kB chunk) of the argument, which is assumed to be a 24-bit `__memx` byte address. If the address does not point to flash memory, the built-in function returns -1.

Example

```
const __memx int myObject = 22;

int main(void) {
  char segment;

  segment = __builtin_avr_flash_segment(&myObject);
}
```

8.9.4 builtin_avr_fmul Built-in Function

Inserts a `fmul` fractional multiplication instruction sequence.

Prototype

```
unsigned int __builtin_avr_fmul(unsigned char x, unsigned char y)
```

Remarks

The built-in will produce code that loads the operands to appropriate registers, performs the fractional unsigned multiplication of **x** and **y**, and stores the result.

Example

```
#include <stdint.h>

int main(void) {
  uint16_t result;
  uint8_t a, b;

  a = 128;
  b = 3;
  result = __builtin_avr_fmul(a, b);  // result will be assigned 768
}
```

8.9.5 builtin_avr_fmuls Built-in Function

Inserts a `fmuls` fractional multiplication instruction sequence.

Prototype

```
int __builtin_avr_fmuls(char x, char y)
```

Remarks

The built-in will produce code that loads the operands to appropriate registers, performs the fractional signed multiplication of **x** and **y**, and stores the result.

Example

```
#include <stdint.h>

int main(void) {
  int16_t result;
  int8_t a, b;

  a = 128;
  b = 3;
  result = __builtin_avr_fmuls(a, b);  // result will be assigned -768
}
```

8.9.6 builtin_avr_fmulsu Built-in Function

Inserts a `fmulsu` fractional multiplication instruction sequence.

Prototype

```
int __builtin_avr_fmulsu(char x, unsigned char y)
```

Remarks

The built-in will produce code that loads the operands to appropriate registers, performs the fractional signed-with-unsigned multiplication of **x** and **y**, and stores the result.

Example

```
#include <stdint.h>

int main(void) {
  int16_t result;
  int8_t a;
  uint8_t b;

  a = 128;
  b = 3;
  result = __builtin_avr_fmulsu(a, b);  // result will be assigned -768
}
```

8.9.7 builtin_avr_insert_bits Built-in Function

Inserts bits into `val` in a manner dictated by a mapping value and returns the resulting value.

Prototype

```
uint8_t __builtin_avr_insert_bits (uint32_t map, uint8_t bits, uint8_t val)
```

Remarks

Insert bits from `bits` into `val` in a manner dictated by `map`, and return the resulting value.

Each of the 8 nibbles (4 bits) of `map` controls a bit in the returned value whose bit position is the same as the nibble's nibble position in `map`. For example, the most-significant nibble of `map` controls insertion into the most-significant bit of the returned value, etc. If a nibble in `map` is 0xF, then the corresponding bit in the returned value is the corresponding bit of `val`, unmodified. Nibbles with a value of 0 through 7 indicate that the corresponding bit in the returned value is obtained from the bit with bit position within `bits`.

Example

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

int main(void) {
  uint8_t  result, myValue, myBits;
  volatile char dummy;
```

```
  myValue = 0xF6;
  myBits  = 0x3A;

  result = __builtin_avr_insert_bits(0x01234567, myBits, myValue);
  printf("result is 0x%X\n", result);
  result = __builtin_avr_insert_bits(0x5566FFFF, myBits, myValue);
  printf("result is 0x%X\n", result);
}
```

Example Output

```
result is 0x5C
result is 0xC6
```

Example Explanation

In the first usage of the built-in in the above example, the first nibble in the `map` value (0x0) indicates that the most-significant bit of the result will be derived from bit #0 in `myBits`. The next lower-significant bit will be derived from bit #1, etc. Given the map nibbles in the above example, the first execution of the built-in will return a value being that of `myBits` with its bits appearing in reverse order.

In the second usage of the built-in, the lower four nibble of `map` are 0xF, which implies that the lower four bits of the result value will be the lower four bits of `myValue`, unmodified. The two most-significant nibbles of `map` are 0x5, indicating that the two most-significant bits of the returned value will be bit #5 of `myBits` (which is 1). Similarly, the next two bits of the result will be bit #6 of `myBits` (which is 0).

8.9.8 builtin_avr_nop Built-in Function

Inserts a `nop` (no operation) instruction.

Prototype

```
void __builtin_avr_nop(void);
```

Remarks

Example

```
int spikeD(void)
{
  PORTD = 0x0F;
  __builtin_avr_nop();
  __builtin_avr_nop();
  PORTD = 0x00;
}
```

8.9.9 builtin_avr_sei Built-in Function

Inserts a `sei` instruction, which enables global interrupts.

Prototype

```
void __builtin_avr_sei(void);
```

Remarks

The `ei()` macro, available once you have included the `<xc.h>` header in your code, performs a similar task.

Example

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

```
  __builtin_avr_cli(); // disable interrupts
  val = TCNT1;         // read timer value register; TEMP used internally
  __builtin_avr_sei(); // re-enable interrupts
  return val;
}
```

8.9.10 builtin_avr_sleep Built-in Function

Inserts a `sleep` instruction, which places the device in sleep mode.

Prototype

```
void __builtin_avr_sleep(void);
```

Example

```
void operationMode(unsigned char channel)
{
  unsigned char val, status=GO;

  while(status) {
    val = readData(channel);
    __builtin_avr_wdr();
    status = processData(val);
    if(status == WAIT) {
      __builtin_avr_sleep();
      status = GO;
    }
    __builtin_avr_wdr();
  }
}
```

8.9.11 builtin_avr_swap Built-in Function

Inserts a `swap` nibble swap instruction sequence.

Prototype

```
unsigned char __builtin_avr_swap(unsigned char)
```

Remarks

The built-in will produce code that loads the operands to appropriate registers, performs the nibble swap, and stores the result.

Example

```
int main(void) {
  uint8_t a;
  uint8_t result;

  a = 0x81;
  result = __builtin_avr_swap(a);  // result will be 0x18
}
```

8.9.12 builtin_avr_wdr Built-in Function

Inserts a `wdr` instruction, which resets the watchdog timer.

Prototype

```
void __builtin_avr_wdr(void);
```

Example

```
void operationMode(unsigned char channel)
{
  unsigned char val, status=GO;
```

```
  while(status) {
    val = readData(channel);
    __builtin_avr_wdr();
    status = processData(val);
    if(status == WAIT) {
      __builtin_avr_sleep();
      status = GO;
    }
    __builtin_avr_wdr();
  }
}
```

9. 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

Revision D (September 2020)

- Updated and added new screen captures of MPLAB X IDE project properties dialogs
- Added screen captures of Microchip Studio project properties dialogs
- Added new `-Wl,--no-data-init` option to control new data initialisation feature
- Documented additional options, including: `-Werror`, `-fdata-sections`, `-ffunction-sections`, `-g`*`level`*

Revision E (August 2021)

- Removed standard C library functions; these are now described in a separate *Microchip Universal Standard Library Reference Guide* document
- Added information on Smart IO features
- Added new `-m[no-]gas-isr-prologues` option and `no_gccisr` attribute
- Added new `-mcall-isr-prologues` option
- Added information on the code coverage feature
- Added information on stack guidance feature
- Expanded and updated information linking sections
- Updated and added new screen captures of MPLAB X IDE project properties dialogs

Revision F (June 2022)

- Corrected code examples and expanded information relating to writing interrupt routines
- Added sections describing the `-msmart-io` and `-msmart-io-format` options, which control the feature set of library code that performs formatted IO

- Added section describing the new `-mreserve` option, which allows you to prevent regions of memory being populated by the linker
- Added sections describing the `-mno-pa-on-file` and `-mno-pa-on-function` options, as well as the new `-mno-pa-outline-calls` option, which all control procedural abstraction optimizations
- Added sections describing the `-f[no-]fat-lto-objects`, `-flto-partition`, `-fomit-frame-pointer`, and `-funroll-[all-]loops` optimzation options
- Added the mapped linker option, `-Wl,--section-start`
- Added in missing sections for the `-MF`, `-P` and `-U` driver options.
- Clarified information relating to the const-in-program-memory feature
- Added sections describing the supported built-in functions
- Adjusted description of how quoted strings can be passed using the `-D` option, which has changed in the compiler
- Made mention of the Analysis Tools Suite license (which replaces the Code Coverage license) required for the code coverage feature
- Added the `__CODECOV` preprocessor macro and indicated when it is defined
- Clarified that the `__nopa` specifier is only available when the CCI is enabled
- Added a section discussing linker script symbols

Revision G (December 2022)

- Updated the operation of the `-mrelax` option, which now performs additional optimizations
- Ensured that the `no-` forms of options are indicated in the option summary tables
- Added description for the `-fcommon` option
- Adjusted the placement of some option descriptions to be consistent with the placement used by GCC docs
- Updated the screen captures of MPLAB X IDE project properties dialogs and described new options
- Added a C99 language divergence relating to the floating-point type specification
- Expanded description of `-d` option
- Provided a summary of the optimizations performed by the compiler
- Added descriptions of the `ei()` and `di()` macros provided by `<xc.h>`
- Added description of the new `NOP()` macro provided by `<xc.h>`
- Corrected the operation summary for the `builtin_avr_cli` and `builtin_avr_insert_bits` built-in functions

Revision H (December 2023)

- Added section for the new `-mfuse-action` option, which can adjust linker options based on some configuration fuse settings
- Added the Common C Interface (CCI) chapter, present in some other compiler user's guides
- Clarified the operation of the `-mconst-data-in-config-mapped-progmem` option and its setting of the FLMAPLOCK bit
- Expanded information relating to the `-fcommon` option
- Expanded information relating to the `-fdata-sections` and `-ffunction-sections` options
- Added new checksum field to the MPLAB X IDE project properties dialogs
- General improvements to the descriptions of compiler features and options

Revision J (October 2024)

- Added section for the `-W[no-]msg` option, which can enable and disable warning and advisory messages
- Expanded the description of the `-Werror` option
- Added description for the `-finline-functions` option
- Added description for the `-mno-data-init` option, which had previously been shown as a mapped linker option
- Added description for the `-T` option, which had previously been shown as a mapped linker option
- Added section on dual-core device support
- Added section on how to program the User Row memory
- Added section on how to program the Boot Row memory
- Added sections on the `keep` object and function attribute
- Added section on the DIVA module routines and associated options, attribute and macro
- Rewrote the sections on writing interrupt code to describe the different strategies available, and added a section on writing interrupt functions using the compiler's new compact vector table feature

Microchip Information

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

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

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Product Identification System

To order or obtain information, e.g., on pricing or delivery, refer to the factory or the listed sales office.

PART NO.	[X](1)	-	X	/XX	XXX
Device	Tape and Reel Option		Temperature Range	Package	Pattern

Device:	PIC16F18313, PIC16LF18313, PIC16F18323, PIC16LF18323	
Tape and Reel Option:	Blank	= Standard packaging (tube or tray)
	T	= Tape and Reel(1)
Temperature Range:	I	= -40°C to +85°C (Industrial)
	E	= -40°C to +125°C (Extended)
Package:(2)	JQ	= UQFN
	P	= PDIP
	ST	= TSSOP
	SL	= SOIC-14
	SN	= SOIC-8
	RF	= UDFN
Pattern:	QTP, SQTP, Code or Special Requirements (blank otherwise)	

Examples:

- PIC16LF18313- I/P Industrial temperature, PDIP package
- PIC16F18313- E/SS Extended temperature, SSOP package

Notes:

1. Tape and Reel identifier only appears in the catalog part number description. This identifier is used for ordering purposes and is not printed on the device package. Check with your Microchip Sales Office for package availability with the Tape and Reel option.
2. Small form-factor packaging options may be available. Please check www.microchip.com/packaging for small-form factor package availability, or contact your local Sales Office.

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip product is strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for

additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP "AS IS". MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, CryptoMemory, CryptoRF, dsPIC, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

AgileSwitch, ClockWorks, The Embedded Control Solutions Company, EtherSynch, Flashtec, Hyper Speed Control, HyperLight Load, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, TimeCesium, TimeHub, TimePictra, TimeProvider, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, Clockstudio, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, EyeOpen, GridTime, IdealBridge, IGaT, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, IntelliMOS, Inter-Chip Connectivity, JitterBlocker, Knob-on-Display, MarginLink, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, mSiC, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, Power MOS IV, Power MOS 7, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SmartHLS, SMART-I.S., storClad, SQI, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, Trusted Time, TSHARC, Turing, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

ISBN: 978-1-6683-0279-8

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office	**Australia - Sydney**	**India - Bangalore**	**Austria - Wels**
2355 West Chandler Blvd.	Tel: 61-2-9868-6733	Tel: 91-80-3090-4444	Tel: 43-7242-2244-39
Chandler, AZ 85224-6199	**China - Beijing**	**India - New Delhi**	Fax: 43-7242-2244-393
Tel: 480-792-7200	Tel: 86-10-8569-7000	Tel: 91-11-4160-8631	**Denmark - Copenhagen**
Fax: 480-792-7277	**China - Chengdu**	**India - Pune**	Tel: 45-4485-5910
Technical Support:	Tel: 86-28-8665-5511	Tel: 91-20-4121-0141	Fax: 45-4485-2829
www.microchip.com/support	**China - Chongqing**	**Japan - Osaka**	**Finland - Espoo**
Web Address:	Tel: 86-23-8980-9588	Tel: 81-6-6152-7160	Tel: 358-9-4520-820
www.microchip.com	**China - Dongguan**	**Japan - Tokyo**	**France - Paris**
Atlanta	Tel: 86-769-8702-9880	Tel: 81-3-6880- 3770	Tel: 33-1-69-53-63-20
Duluth, GA	**China - Guangzhou**	**Korea - Daegu**	Fax: 33-1-69-30-90-79
Tel: 678-957-9614	Tel: 86-20-8755-8029	Tel: 82-53-744-4301	**Germany - Garching**
Fax: 678-957-1455	**China - Hangzhou**	**Korea - Seoul**	Tel: 49-8931-9700
Austin, TX	Tel: 86-571-8792-8115	Tel: 82-2-554-7200	**Germany - Haan**
Tel: 512-257-3370	**China - Hong Kong SAR**	**Malaysia - Kuala Lumpur**	Tel: 49-2129-3766400
Boston	Tel: 852-2943-5100	Tel: 60-3-7651-7906	**Germany - Heilbronn**
Westborough, MA	**China - Nanjing**	**Malaysia - Penang**	Tel: 49-7131-72400
Tel: 774-760-0087	Tel: 86-25-8473-2460	Tel: 60-4-227-8870	**Germany - Karlsruhe**
Fax: 774-760-0088	**China - Qingdao**	**Philippines - Manila**	Tel: 49-721-625370
Chicago	Tel: 86-532-8502-7355	Tel: 63-2-634-9065	**Germany - Munich**
Itasca, IL	**China - Shanghai**	**Singapore**	Tel: 49-89-627-144-0
Tel: 630-285-0071	Tel: 86-21-3326-8000	Tel: 65-6334-8870	Fax: 49-89-627-144-44
Fax: 630-285-0075	**China - Shenyang**	**Taiwan - Hsin Chu**	**Germany - Rosenheim**
Dallas	Tel: 86-24-2334-2829	Tel: 886-3-577-8366	Tel: 49-8031-354-560
Addison, TX	**China - Shenzhen**	**Taiwan - Kaohsiung**	**Israel - Hod Hasharon**
Tel: 972-818-7423	Tel: 86-755-8864-2200	Tel: 886-7-213-7830	Tel: 972-9-775-5100
Fax: 972-818-2924	**China - Suzhou**	**Taiwan - Taipei**	**Italy - Milan**
Detroit	Tel: 86-186-6233-1526	Tel: 886-2-2508-8600	Tel: 39-0331-742611
Novi, MI	**China - Wuhan**	**Thailand - Bangkok**	Fax: 39-0331-466781
Tel: 248-848-4000	Tel: 86-27-5980-5300	Tel: 66-2-694-1351	**Italy - Padova**
Houston, TX	**China - Xian**	**Vietnam - Ho Chi Minh**	Tel: 39-049-7625286
Tel: 281-894-5983	Tel: 86-29-8833-7252	Tel: 84-28-5448-2100	**Netherlands - Drunen**
Indianapolis	**China - Xiamen**		Tel: 31-416-690399
Noblesville, IN	Tel: 86-592-2388138		Fax: 31-416-690340
Tel: 317-773-8323	**China - Zhuhai**		**Norway - Trondheim**
Fax: 317-773-5453	Tel: 86-756-3210040		Tel: 47-72884388
Tel: 317-536-2380			**Poland - Warsaw**
Los Angeles			Tel: 48-22-3325737
Mission Viejo, CA			**Romania - Bucharest**
Tel: 949-462-9523			Tel: 40-21-407-87-50
Fax: 949-462-9608			**Spain - Madrid**
Tel: 951-273-7800			Tel: 34-91-708-08-90
Raleigh, NC			Fax: 34-91-708-08-91
Tel: 919-844-7510			**Sweden - Gothenberg**
New York, NY			Tel: 46-31-704-60-40
Tel: 631-435-6000			**Sweden - Stockholm**
San Jose, CA			Tel: 46-8-5090-4654
Tel: 408-735-9110			**UK - Wokingham**
Tel: 408-436-4270			Tel: 44-118-921-5800
Canada - Toronto			Fax: 44-118-921-5820
Tel: 905-695-1980			
Fax: 905-695-2078			