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

1. Preface

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation:

Table 1-1. Documentation Conventions

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

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

1.2 Recommended Reading

This user’s guide describes the use and features of the MPLAB XC8 C Compiler when building for PIC 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 Legacy User’s Guide
This version of the compiler's user's guide is for legacy projects that use the old `xc8` command line driver or the ISO/IEC 9899:1990 Standard (C90) for programming languages. When operating in this mode, the compiler uses a different front end with different specifications and output.

MPLAB® XC8 C Compiler User’s Guide for AVR® MCU
This version of the compiler's user's guide is for projects that target 8-bit AVR devices.

MPLAB® XC8 C Compiler Release Notes for PIC® 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.

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.

The compiler supports all 8-bit PIC® and AVR® microcontrollers; however, this document describes the use of the `xc8-cc` driver for programs that target only Microchip PIC devices, and additionally, for programs that are built against the C99 Standard. See the *MPLAB® XC8 C Compiler User's Guide for AVR® MCU* (DS50002750), for information on using this compiler when targeting Microchip AVR devices. If you are using the C90 Standard or the legacy compiler driver (`xc8`), see the *MPLAB® XC8 C Legacy Compiler User's Guide* (DS50002053) document.

Note: Features described as being part of MPLAB XC8 in this document assume that you are using a Microchip PIC device and are building for the C99 C standard. These features may differ if you choose to instead compile for a Microchip AVR device or for the C90 standard.

When compiling for the C99 standard, this compiler utilizes the Clang compiler front end. The older CPP/P1 front end is used when building for C90 projects.

2.1 Device Description

This guide describes the MPLAB XC8 C Compiler's support for all 8-bit Microchip PIC devices with baseline, enhanced baseline, mid-range, enhanced mid-range and PIC18 cores. Check the `ARCH` field in the device's INI file (`pic/dat/ini` directory of your compiler installation directory) to confirm the core architecture used by the compiler when building code. The following descriptions indicate the distinctions within those device cores:

The baseline core uses a 12-bit-wide instruction set and is available in PIC10, PIC12 and PIC16 part numbers. (`ARCH` value of `PIC12`)

The enhanced baseline core also uses a 12-bit instruction set, but this set includes additional instructions. Some of the enhanced baseline chips support interrupts and the additional instructions used by interrupts. These devices are available in PIC12 and PIC16 part numbers. (`ARCH` value of `PIC12E`, or `PIC12IE` for those with interrupt support)

The mid-range core uses a 14-bit-wide instruction set that includes more instructions than the baseline core. It has larger data memory banks and program memory pages, as well. It is available in PIC12, PIC14 and PIC16 part numbers. (`ARCH` value of `PIC14`)

The Enhanced mid-range core also uses a 14-bit-wide instruction set but incorporates additional instructions and features. There are both PIC12 and PIC16 part numbers that are based on the Enhanced mid-range core. (`ARCH` value of `PIC14E` or `PIC14EX`)

The PIC18 core instruction set is 16 bits wide and features additional instructions and an expanded register set. PIC18 core devices have part numbers that begin with PIC18. Some PIC18 devices implement extended data memory and a vectored interrupt controller module with support for one or more interrupt vector tables, rather than fixed-location, dual priority vectors. (`ARCH` value of `PIC18`, or `PIC18XV` for those with the extended data memory and the vectored interrupt controller module)

The compiler takes advantage of the target device's instruction set, addressing modes, memory, and registers whenever possible.

See 4.6.2.8 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.

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 PIC 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 4.6 Option Descriptions. Accordingly, “compiler options” commonly refers to command-line driver options.

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

2.5 Compatible Development Tools

The compiler works with many other Microchip tools, including:

- The MPLAB X IDE (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 PIC devices

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

3. How Tos

This section contains help and references for situations that are frequently encountered when building projects for Microchip 8-bit PIC devices. Click the links at the beginning of each section to assist in finding the topic relevant to your question. Some topics are indexed in multiple sections.

Start here:

- Installing and Activating the Compiler
- Invoking the Compiler
- Writing Source Code
- Getting My Application to Do What I Want
- Understanding the Compilation Process
- Fixing Code That Does Not Work

3.1 Installing and Activating the Compiler

This section details questions that might arise when installing or activating the compiler.

- How Do I Install and Activate My Compiler?
- How Can I Tell if the Compiler has Activated Successfully?
- Can I Install More Than One Version of the Same Compiler?

3.1.1 How Do I Install and Activate My Compiler?

Installation of the compiler and activation of the license are performed simultaneously by the XC compiler installer. The guide Installing and Licensing MPLAB XC C Compilers (DS52059) is available on https://www.microchip.com/compilers, under the **Documentation** tab. It provides details on single-user and network licenses, as well as how to activate a compiler for evaluation purposes.

3.1.2 How Can I Tell if the Compiler has Activated Successfully?

If you think the compiler has not installed correctly or is not activated, it is best to verify its operation outside of the MPLAB X IDE to isolate any potential compiler or IDE problems.

The `xclm` application, which is shipped with the compiler, can be queried to determine the status of your compiler. For example, from your DOS-prompt, type the following line, using the appropriate compiler path.

```
"C:\Program Files (x86)\Microchip\xc8\v2.00\bin\xclm" -status
```

This will show the licenses installed on the machine, allowing you to see if the compiler was activated successfully. The status of your compiler license can also be checked under the MPLAB X IDE under **Tools** > **LicensesLicense** > **Status**.

3.1.3 Can I Install More Than One Version of the Same Compiler?

Yes, the compilers and installation process has been designed to allow you to have more than one version of the same compiler installed, and you can easily move between the versions by changing options in MPLAB X IDE; see 3.2.4 How Can I Select Which Compiler I Want to Build With?

Compilers should be installed into a directory whose name is related to the compiler version. This is reflected in the default directory specified by the installer. For example, the 1.44 and 1.45 MPLAB XC8 compilers would typically be placed in separate directories.

```
C:\Program Files (x86)\Microchip\xc8\v1.44\
C:\Program Files (x86)\Microchip\xc8\v1.45\
```

3.2 Invoking the Compiler

This section discusses how the compiler is run, on the command-line or from the MPLAB X IDE. It includes information about how to get the compiler to do what you want it to do, in terms of options and the build process itself.

- How Do I Compile From Within MPLAB X IDE?
- How Do I Compile on the Command-line?
- How Do I Compile Using a Make Utility?
- How Can I Select Which Compiler I Want to Build With?
- How Do I Build Libraries?
- How Do I Know What Compiler Options Are Available and What They Do?
- What is Different About an MPLAB X IDE Debug Build?

See also the following linked information in other sections.

- What Do I Need to Do When Compiling to Use a Debugger?
- How Do I Use Library Files in My Project?
- How Do I Place a Function Into a Unique Section?
- What Optimizations Are Employed by the Compiler?

3.2.1 How Do I Compile From Within MPLAB X IDE?

MPLAB X IDE User's Guide and online help provide directions for setting up a project in the MPLAB X integrated development environment.

Alternatively, download the MPLAB® XC8 User's Guide for Embedded Engineers (DS50002400) from the Documentation tab on the Compilers' web page or open the MPLAB® XC8 Getting Started Guide (DS50002173) from the compiler's docs directory.

3.2.2 How Do I Compile On The Command-line?

The compiler driver is called `xc8-cc` for all 8-bit devices; e.g., in Windows, it is named `xc8-cc.exe`. This application should be invoked for all aspects of compilation. It is located in the `bin` directory of the compiler distribution. Avoid running the individual compiler applications (such as the assembler or linker) explicitly. You can compile and link in the one command, even if your project is made up of multiple source files.

The driver command format is introduced in 4.1 Invoking The Compiler. See the 3.2.4 How Can I Select Which Compiler I Want to Build With? section to ensure you are running the correct driver if you have more than one installed. The command-line options to the driver are detailed in 4.6 Option Descriptions. The files that can be passed to the driver are listed and described in 4.1.3 Input File Types.

3.2.3 How Do I Compile Using a Make Utility?

When compiling using a make utility (such as `make`), the compilation is usually performed as a two-step process: first generating the intermediate files, then the final compilation and link step to produce one binary output (as described in 4.2.3 Multi-Step C Compilation).

The MPLAB XC8 C Compiler uses a unique technology called OCG that uses an intermediate file format that is different than traditional compilers (including XC16 and XC32). The intermediate file format used by XC8 for C source files is a p-code file (`.p1` extension), not an object file.

3.2.4 How Can I Select Which Compiler I Want to Build With?

The compilation and installation process has been designed to allow you to have more than one compiler installed at the same time. You can create a project in MPLAB X IDE and then build this project with different compilers by simply changing a setting in the project properties.

To select which compiler is actually used when building a project under MPLAB X IDE, go to the **Project Properties** dialog. Select the **Configuration** category (**Conf: [default]**where **default** represents the name of the project configuration). A list of MPLAB XC8 compilers is shown in the **Compiler Toolchain**, on the far right. Select the compiler that you require.

Once selected, the controls for that compiler are then shown by selecting the **XC8 Global options**, **MPLAB XC8 Compiler** and **MPLAB XC8 Linker** categories. These reveal a pane of options on the right. Note that each category has several panes which can be selected from a pull-down menu that is near the top of the pane.

3.2.5 How Do I Build Libraries?

Use the librarian, `xc8-ar`, to build libraries from p-code (`.p1`) or object (`.o`) modules. Libraries can contain any mix of these module types, but C source code to be placed into a library should always be pre-built into a p-code module. Object modules should only be used for assembler source. See 4.2.3 Multi-Step C Compilation for information on building p-code files and 8.1 Archiver/Librarian for the librarian options.

For example:

```
xc8-cc -mcpu=16f877a -c lcd.c utils.c io.c
xc8-ar -r myLib.a lcd.p1 utils.p1 io.p1
```

creates a library file called `myLib.a`.

The MPLAB X IDE allows you to create a library project that will build a library file as the final output. Select Library Project from the Microchip Embedded Category when creating a new project to create a library of your source code.

Note that if you intend to step through your library code at a C level in MPLAB X IDE, you will need to place the library source files so that the relative path between their location and the project that is using them is the same as the relative path between where the library build command was executed and where the source files were located when they were built.

3.2.6 How Do I Know What Compiler Options Are Available and What They Do?

A list of all compiler options can be obtained by using the `--help` option on the command line. Alternatively, all options are listed in 4.6 Option Descriptions of this user's guide.

3.2.7 What is Different About an MPLAB X IDE Debug Build?

In MPLAB X, there are distinct build buttons and menu items to build (production build) a project and to debug (debug build) a project.

When performing a debug build, the IDE will also set the configuration bit to allow debugging of the project by a debug tool, such as the MPLAB ICD 4.

In MPLAB X IDE, memory is reserved for your debugger (if selected) only when you perform a debug build. See 3.4.4 What Do I Need to Do When Compiling to Use a Debugger?

Another difference is the setting of a preprocessor macro called `__DEBUG`, which is assigned 1 by the MPLAB X IDE when performing a debug build. This macro is not defined for production builds. You can make code in your source conditional on this macro using `#ifdef` directives, etc., (see 5.13.1 Preprocessor Directives); so your program will behave differently when in a development cycle.

3.3 Writing Source Code

This section presents issues that pertain to the source code you write. It has been subdivided into the sections listed below.

- C Language Specifics
- Device-Specific Features
- Memory Allocation
- Variables
- Functions
- Interrupts
- Assembly Code

3.3.1 C Language Specifics

This section discusses commonly asked source code issues that directly relate to the C language itself.

- When Should I Cast Expressions?
- Can Implicit Type Conversions Change the Expected Results of My Expressions?
- How Do I Enter Non-English Characters Into My Program?
- How Can I Use a Variable Defined in Another Source File?
- How Can I Use a Function Defined in Another Source File?

3.3.1.1 When Should I Cast Expressions?

Expressions can be explicitly cast using the cast operator -- a type in round brackets, e.g., `(int)`. In all cases, conversion of one type to another must be done with caution and only when absolutely necessary.

Consider the example:

```
unsigned long l;
unsigned int i;
i = l;
```

Here, a `long` type is being assigned to an int type and the assignment will truncate the value in l. The compiler will automatically perform a type conversion from the type of the expression on the right of the assignment operator (`long`) to the type of the value on the left of the operator (`int`).This is called an implicit type conversion. The compiler typically produces a warning concerning the potential loss of data by the truncation.

A cast to type `int` is not required and should not be used in the above example if a `long` to `int` conversion was intended. The compiler knows the types of both operands and performs the conversion accordingly. If you did use a cast, there is the potential for mistakes if the code is later changed. For example, if you had:

```
i = (int)l;
```

the code works the same way; but if in the future, the type of `i` is changed to a `long`, for example, then you must remember to adjust the cast, or remove it, otherwise the contents of `l` will continue to be truncated by the assignment, which cannot be correct. Most importantly, the warning issued by the compiler will not be produced if the cast is in place.

Only use a cast in situations where the types used by the compiler are not the types that you require. For example, consider the result of a division assigned to a floating point variable:

```
int i, j;
float fl;
fl = i/j;
```

In this case, integer division is performed, then the rounded integer result is converted to a float format. So if `i` contained 7 and `j` contained 2, the division yields 3 and this is implicitly converted to a `float` type (3.0) and then assigned to `fl`. If you wanted the division to be performed in a float format, then a cast is necessary:

```
fl = (float)i/j;
```

(Casting either `i` or `j` forces the compiler to encode a floating-point division.) The result assigned to `fl` now is 3.5.

An explicit cast can suppress warnings that might otherwise have been produced. This can also be the source of many problems. The more warnings the compiler produces, the better chance you have of finding potential bugs in your code.

3.3.1.2 Can Implicit Type Conversions Change The Expected Results Of My Expressions?

Yes!

The compiler will always use integral promotion and there is no way to disable this (see 5.5.1 Integral Promotion). In addition, the types of operands to binary operators are usually changed so that they have a common type, as specified by the C Standard. Changing the type of an operand can change the value of the final expression, so it is very important that you understand the type C Standard conversion rules that apply when dealing with binary operators. You can manually change the type of an operand by casting; see 3.3.1.1 When Should I Cast Expressions?

3.3.1.3 How Do I Enter Non-English Characters Into My Program?

The C standard (and accordingly, the MPLAB XC8 C compiler) does not support extended characters set in character and string literals in the source character set. See 5.3.7 Constant Types and Formats to see how these characters can be entered using escape sequences.

3.3.1.4 How Can I Use A Variable Defined In Another Source File?

Provided the variable defined in the other source file is not specified `static` or `auto`, then adding a declaration (as opposed to a definition) for that variable into the current file will allow you to access it. A declaration consists of the keyword `extern` in addition to the correct type and the exact name of the variable specified in its definition, e.g.,

```
extern int systemStatus;  // declare systemStatus for use here
```

This storage-class specifier is part of the C language and your favorite C textbook will provide more information on its usage.

The position of the declaration in the current file determines the scope of the variable. That is, if you place the `extern` declaration inside a function, it will limit the scope of the variable to within that function. If you place it outside of a function, it allows access to the variable in all functions for the remainder of the current file.

Declarations are often placed in header files and then `#include`d into the C source code (see 5.13.1 Preprocessor Directives).

3.3.1.5 How Can I Use A Function Defined In Another Source File?

Provided the function defined in the other source file is not `static`, then adding a declaration (as opposed to a definition) for that function into the current file will allow you to call it. A declaration optionally consists of the keyword `extern` in addition to the exact function prototype. You can omit the names of the parameters in the declaration. If you include the parameter names, they should match the definition. For example:

```
extern int readStatus(int);  // declare readStatus for use here
```

This storage-class specifier is part of the C language, and your favorite C textbook will provide more information on its usage.

Often, declarations are placed in header files and then they are `#include`d into the C source code (see 5.13.1 Preprocessor Directives).

3.3.2 Device-Specific Features

This section discusses the code that needs to be written to set up or control a feature that is specific to Microchip PIC devices.

- How Do I Set the Configuration Bits?
- How Do I Use the PIC Device's ID Locations?
- How Do I Determine the Cause of Reset on Mid-range Parts?
- How Do I Access SFRs?
- How Do I Place a Function Into a Unique Section?
- How Do I Find The Names Used to Represent SFRs and Bits?

See also the following linked information in other sections.

- What Do I Need to Do When Compiling to Use a Debugger?

3.3.2.1 How Do I Set The Configuration Bits?

These should be set in your code using the `config` pragma. You can have the MPLAB X IDE generate the appropriate pragmas for you, but this code must still be copied into a source file that is part of your project. See 5.2.5 Configuration Bit Access, for details about how these bits are set.

3.3.2.2 How Do I Use The PIC Device's ID Locations?

The `config` pragma allows the ID location values to be programmed (see 5.2.6 ID Locations).

3.3.2.3 How Do I Determine The Cause Of Reset On Mid-range Parts?

The TO and PD bits in the STATUS register allow you to determine the cause of a Reset. However, these bits are quickly overwritten by the runtime startup code that is executed before `main()`; see 5.9.2 Runtime Startup Code for more information. You can have the STATUS register saved into a location that is later accessible from C code, so that the cause of Reset can be determined by the application after it is running again (see 5.9.2.4 Status Register Preservation).

3.3.2.4 How Do I Access SFRs?

The compiler ships with header files (see 5.2.3 Device Header Files) that define variables that are mapped over memory-mapped SFRs. Since these are C variables, they can be used like any other variable and no new syntax is required to access these registers.

Bits within SFRs can also be directly accessed via structures. Within the structure are bit-field members that allow access to individual elements in the register. Individual single-bit variables are also defined that are mapped over the bits in the SFR, but these should only be used in legacy projects. (see 5.2.7 Using SFRs From C Code).

The name assigned to the variable is usually the same as the name specified in the device data sheet. See 3.3.2.5 How Do I Find The Names Used To Represent SFRs And Bits? if these names are not recognized.

3.3.2.5 How Do I Find The Names Used To Represent SFRs And Bits?

Special function registers, as well as the bits within them, are accessed via special variables that map over the register; however, the names of these variables sometimes differ from those indicated in the data sheet for the device you are using.

If required, you can examine the `<xc.h>` header file to find the device-specific header file that is relevant for your device. This file will define the variables that allow access to these special variables. However, an easier way to find these variable names is to look in any of the preprocessed files left behind from a previous compilation. Provided the corresponding source file included `<xc.h>`, the preprocessed file will show the definitions for all the SFR variables and bits for your target device.

If you are compiling under MPLAB X IDE for a configuration called `default`, the preprocessed file(s) are left under the`build/default/production` directory of your project for regular builds, or under `build/default/debug` for debug builds. They are typically left in the source file directory if you are compiling on the command line. These files have an `.i` extension.

3.3.3 Memory Allocation

Here are questions relating to how your source code affects memory allocation.

- How Do I Position Variables at an Address I Nominate?
- How Do I Place a Variable Into a Unique Section?
- How Do I Position a Variable Into an Address Range?
- How Do I Position Functions at an Address I Nominate?
- How Do I Place Variables in Program Memory?
- How Do I Place a Function Into a Unique Section?
- How Do I Position a Function Into an Address Range?
- How Do I Stop the Compiler From Using Certain Memory Locations?

See also the following linked information in other sections.

- Why Are Some Objects Positioned Into Memory That I Reserved?
- How Do I Use High-Endurance Flash for Data, Not Code?

3.3.3.1 How Do I Position Variables At An Address I Nominate?

The easiest way to do this is to make the variable absolute by using the `__at(`*`address`*`)` construct (see 5.4.4 Absolute Variables), but you might consider also placing the variable in a unique section (see 5.14.3 Changing and Linking the Allocated Section). Absolute variables use the address you nominate in preference to the variable's symbol in generated code.

3.3.3.2 How Do I Place A Variable Into A Unique Section?

Use the `__section()` specifier to have the variable positioned in a new section (psect). After this has been done, the section can be linked to the desired address by using the `-Wl` driver option. See 5.14.3 Changing and Linking the Allocated Section for examples of both these operations.

3.3.3.3 How Do I Position A Variable Into An Address Range?

You need to move the variable into a unique section (psect), define a memory range and then place the new section in that range.

Use the `__section()` specifier to have the variable positioned in a new section. Use the `-Wl` driver option to define a memory range and to place the new section in that range. See 5.14.3 Changing and Linking the Allocated Section for examples of all these operations.

3.3.3.4 How Do I Position Functions At An Address I Nominate?

The easiest way to do this is to make the functions absolute by using the `__at(`*`address`*`)` construct, (see 5.7.3 Changing the Default Function Allocation), but you might consider also placing the variable in a unique section (see 5.14.3 Changing and Linking the Allocated Section). This means that the address you specify is used in preference to the function's symbol in generated code.

3.3.3.5 How Do I Place Variables In Program Memory?

The `const` qualifier implies that the qualified variable is read-only. Since these objects cannot be written, they are typically placed in program memory, thus freeing valuable data RAM. The exceptions to this are `const`-qualified parameters and some `const`-qualified auto objects. See 5.1.2.4 Const Auto Objects and 5.4.3 Objects in Program Space for more information. Variables that are qualified `const` can also be made absolute, so they can be positioned at an address you nominate (see 5.4.4.2 Absolute Objects In Program Memory).

3.3.3.6 How Do I Place A Function Into A Unique Section?

Use the `__section()` specifier to have the function positioned into a new section (psect). When this has been done, the section can be linked to the desired address by using the `-Wl` driver option. See 5.14.3 Changing and Linking the Allocated Section for examples of both these operations.

3.3.3.7 How Do I Position A Function Into An Address Range?

Having one or more functions located in a special area of memory might mean that you can ensure they are code protected, for example. To do this, you need to move the function into a unique section (psect), define a memory range, and then place the new section in that range.

Use the `__section()` specifier to have the function positioned into a new section. Use the `-Wl` driver option to define a memory range and to place the new section into that range. See 5.14.3 Changing and Linking the Allocated Section for examples of all these operations.

3.3.3.8 How Do I Stop The Compiler From Using Certain Memory Locations?

Memory can be reserved when you build. The `-mreserve` option allows you to adjust the ranges of data and program memory when you build (see 4.6.1.18 Reserve Option). By default, all the available on-chip memory is available for use. However, these options allow you to reserve parts of this memory.

3.3.4 Variables

This sections examines questions that relate to the definition and usage of variables and types within a program.

- Why Are My Floating-point Results Not Quite What I Am Expecting?
- How Can I Access Individual Bits of a Variable?

See also the following linked information in other sections.

- How Do I Share Data Between Interrupt and Main-line Code?
- How Do I Position Variables at an Address I Nominate?
- How Do I Place Variables in Program Memory?
- How Do I Place Variables in the PIC18 Device's External Program Memory?
- How Can I Rotate a Variable?
- How Do I Utilize/Allocate the RAM Banks on My Device?
- How Do I Utilize the Linear Memory on Enhanced Mid-range PIC Devices?
- How Do I Find Out Where Variables and Functions Have Been Positioned?

3.3.4.1 Why Are My Floating-point Results Not Quite What I Am Expecting?

The size of the floating point type can be adjusted for both `float` and `double` types only when building to the C90 standard (see 4.6.14.2 Short Float Option and 4.6.14.1 Short Double Option).

Since floating-point variables only have a finite number of bits to represent the values they are assigned, they only hold an approximation of their assigned value (see 5.3.4 Floating-Point Data Types). A floating-point variable can only hold one of a set of discrete real number values. If you attempt to assign a value that is not in this set, it is rounded to the nearest value. The more bits used by the mantissa in the floating-point variable, the more values can be exactly represented in the set, and the average error due to the rounding is reduced.

Whenever floating-point arithmetic is performed, rounding also occurs. This can also lead to results that do not appear to be correct.

3.3.4.2 How Can I Access Individual Bits Of A Variable?

There are several ways of doing this. The simplest and most portable way is to define an integer variable and use macros to read, set, or clear the bits within the variable using a mask value and logical operations, such as the following.

```
#define  testbit(var, bit)   ((var) & (1 <<(bit)))
#define  setbit(var, bit)    ((var) |= (1 << (bit)))
#define  clrbit(var, bit)    ((var) &= ~(1 << (bit)))
```

These, respectively, test to see if bit number, *bit*, in the integer, *var*, is set; set the corresponding bit in *var*; and clear the corresponding bit in *var*. Alternatively, a union of an integer variable and a structure with bit-fields (see 5.3.5.2 Bit-fields In Structures) can be defined, e.g.,

```
union both {
  unsigned char byte;
  struct {
    unsigned b0:1, b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1;
  } bitv;
} var;
```

This allows you to access byte as a whole (using `var.byte`), or any bit within that variable independently (using `var.bitv.b0` through `var.bitv.b7`).

Note that the compiler does support `bit` variables (see 5.3.2.1 Bit Data Types And Variables), as well as bit-fields in structures.

3.3.5 Functions

This section examines questions that relate to functions.

- What is the Optimum Size For Functions?
- How Do I Stop An Unused Function Being Removed?
- How Do I Make a Function Inline?

See also the following linked information in other sections.

- How Can I Tell How Big a Function Is?
- How Do I Position Functions at an Address I Nominate?
- How Do I Know Which Resources Are Being Used by Each Function?
- How Do I Find Out Where Variables and Functions Have Been Positioned?
- How Do I Use Interrupts in C?

3.3.5.1 What Is The Optimum Size For Functions?

Generally speaking, the source code for functions should be kept small, as this aids in readability and debugging. It is much easier to describe and debug the operation of a function that performs a small number of tasks. And small functions typically have fewer side effects, which can be the source of coding errors. In the embedded programming world, however, a large number of small functions, as well as the calls necessary to execute them, can result in excessive memory and stack usage, so a compromise is often necessary.

The PIC10/12/16 devices employ pages in the program memory that are used to store and execute function code. Although you are able to write C functions that will generate more than one page of assembly code, functions of such a size should be avoided and split into smaller routines where possible. (see 5.7.2 Allocation of Executable Code) The assembly call and jump sequences to locations in other pages are much longer than those made to destinations in the same page. If a function is so large as to cross a page boundary, then loops (or other code constructs that require jumps within that function) can use the longer form of jump on each iteration.

PIC18 devices are less affected by internal memory paging and the instruction set allows for calls and jumps to any destination with no penalty. But you should still endeavor to keep functions as small as possible.

Interrupt functions must be written so that they do not exceed the size of a memory page. They cannot be split to occupy more than one page.

With all devices, the smaller the function, the easier it is for the linker to allocate it to memory without errors.

3.3.5.2 How Do I Stop An Unused Function Being Removed?

If a C function's symbol is referenced in hand-written assembly code, the function will never be removed, even if it is not called or has never had its address taken in C code.

To include a reference, create an assembly source file and add this file to your project. You only have to reference the symbol in this file; so the file can contain the following

```
GLOBAL _myFunc
```

where `myFunc` is the C name of the function in question (note the leading underscore in the assembly name, see 5.11.3.3 Equivalent Assembly Symbols). This is sufficient to prevent the function removal optimization from being performed.

3.3.5.3 How Do I Make A Function Inline?

You can ask the compiler to inline a function by using the `inline` specifier (see 5.7.1.2 Inline Specifier) or `#pragma inline`. This is only a suggestion to the compiler and cannot always be obeyed. Do not confuse this specifier/pragma with the `intrinsic` pragma(1) (see 5.13.3.4 The #pragma Intrinsic Directive), which is for functions that have no corresponding source code and which will be specifically expanded by the code generator during compilation.

3.3.6 Interrupts

Interrupt and interrupt service routine questions are discussed in this section.

- How Do I Use Interrupts in C?

See also the following linked information in other sections.

- How Can I Make My Interrupt Routine Faster?
- How Do I Share Data Between Interrupt and Main-line Code?

3.3.6.1 How Do I Use Interrupts In C?

Be aware of what sort of interrupt hardware is available on your target device. Most baseline PIC devices do not implement interrupts at all; baseline devices with interrupts and mid-range devices utilize a single interrupt vector. PIC18 devices implement two separate interrupt vector locations and use a simple priority scheme. Some PIC18 devices use a vectored interrupt controller to invoke multiple interrupt functions.

In C source code, a function can be written to act as the interrupt service routine (see 5.8.1 Writing an Interrupt Service Routine). Such functions save and/or restore program context before and/or after executing the function body code and a different return instruction is used (see 5.8.4 Context Switching).

Code inside the interrupt function can do anything you like, but see 3.5.7 How Can I Make My Interrupt Routine Faster? for suggestions to enhance real-time performance.

Prior to any interrupt occurring, your program must ensure that peripherals are correctly configured and that interrupts are enabled (see 5.8.5 Enabling Interrupts). On PIC18 devices, you must specify the priority of interrupt sources by writing the appropriate SFRs.

3.3.7 Assembly Code

This section examines questions that arise when writing assembly code as part of a C project.

- How Should I Combine Assembly and C Code?
- What Do I Need Other than Instructions in an Assembly Source File?
- How Do I Access C Objects from Assembly Code?

[1] This specifier was originally named `in-line` but was changed to avoid confusion.

- How Can I Access SFRs from Within Assembly Code?
- What Things Must I Manage When Writing Assembly Code?

3.3.7.1 How Should I Combine Assembly And C Code?

Ideally, any hand-written assembly should be written as separate routines that can be called. This offers some degree of protection from interaction between compiler-generated and hand-written assembly code. Such code can be placed into a separate assembly module that can be added to your project (see 5.11.1 Integrating Assembly Language Modules).

If necessary, assembly code can be added in-line with C code (see 5.11.2 Inline Assembly). The code added in-line should ideally be limited to instructions such as `nop`, `sleep` or `clrwdt`. Macros are already provided which in-line all these instructions (see, for example, 9.2.19 NOP Macro). More complex in-line assembly that changes register contents and the device state can cause code failure if precautions are not taken, and such assembly should be used with caution. See 5.6 Register Usage for those registers used by the compiler.

3.3.7.2 What Do I Need Other Than Instructions In An Assembly Source File?

Assembly code typically needs assembler directives as well as the instructions themselves. The operation of all the directives are described in the subsections of 6.1.9 Assembler Directives. Common directives required are mentioned below.

All assembly code must be placed in a psect (section) so it can be manipulated as a whole by the linker and placed in memory. See 5.14.1 Compiler-Generated Psects for general information on psects; see 6.1.9.36 Psect Directive for information on the directive used to create and specify psects.

The other commonly used directive is `GLOBAL`, defined in 6.1.9.21 Global Directive which is used to make symbols accessible across multiple source files.

3.3.7.3 How Do I Access C Objects From Assembly Code?

Most C objects are accessible from assembly code. There is a mapping between the symbols used in the C source and those used in the assembly code generated from this source. Your assembly should access the assembly-equivalent symbols which are detailed in 5.11.3.3 Equivalent Assembly Symbols.

Instruct the assembler that the symbol is defined elsewhere by using the `GLOBAL` assembler directive (see 6.1.9.21 Global Directive). This is the assembly equivalent of a C declaration, although no type information is present. This directive is not needed and should not be used if the symbol is defined in the same module as your assembly code.

Any C variable accessed from assembly code will be treated as if it were qualified `volatile`. Specifically specifying the `volatile` qualifier in C code is preferred as it makes it clear that external code can access the object.

3.3.7.4 How Can I Access SFRs From Within Assembly Code?

The safest way to gain access to SFRs in assembly code is to have symbols defined in your assembly code that equate to the corresponding SFR address. Header files are provided with the compiler so that you do not need to define these yourselves (detailed in 5.11.3.4 Accessing Registers From Assembly Code).

There is no guarantee that you will be able to access symbols generated by the compilation of C code, even the code that accesses the SFR that you require.

3.3.7.5 What Things Must I Manage When Writing Assembly Code?

When writing assembly code by hand, you assume responsibility for managing certain features of the device and formatting your assembly instructions and operands. The following list describes some of the actions you must take.

- Whenever you access a RAM variable, you must ensure that the bank of the variable is selected before you read or write the location. This is done by one or more assembly instructions. The exact code is based on the device you are using and the location of the variable. Bank selection is not be required if the object is in common memory, (which is called the access bank on PIC18 devices) or if you are using an instruction that takes a full address (such as the `movff` instruction on PIC18 devices). Check your device data sheet to see the memory architecture of your device, as well as the instructions and registers which control bank selection. Failure to select the correct bank will lead to code failure.
  The `BANKSEL` pseudo instruction can be used to simplify this process (see 6.1.1.2 Bank And Page Selection).
- You must ensure that the address of the RAM variable you are accessing has been masked so that only the bank offset is being used as the instruction's file register operand. This should not be done if you are using an

instruction that takes a full address (such as the `movff` instruction on PIC18 devices). Check your device data sheet to see what address operand instructions requires. Failure to mask an address can lead to a fixup error (see 3.6.8 How Do I Fix a Fixup Overflow Error?) or code failure.
The `BANKMASK` macro can truncate the address for you (see 5.11.3.4 Accessing Registers From Assembly Code).

- Before you call or jump to any routine, you must ensure that you have selected the program memory page of this routine using the appropriate instructions. You can either use the `PAGESEL` pseudo instruction (see 6.1.1.2 Bank And Page Selection), or the `fcall` or `ljmp` pseudo instructions (not required on PIC18 devices) (see 6.1.1.7 Long Jumps And Calls) which will automatically add page selection instructions, if required.
- You must ensure that any RAM used for storage has memory reserved. If you are only accessing variables defined in C code, then reservation is already done by the compiler. You must reserve memory for any variables you only use in the assembly code using an appropriate directive such as `DS` or `DABS` (see 6.1.9.13 Ds Directive or 6.1.9.8 Dabs Directive). It is often easier to define objects in C code rather than in assembly code.
- You must place any assembly code you write in a psect (see 6.1.9.36 Psect Directive for the directive to do this, and 5.14.1 Compiler-Generated Psects for general information about psects). A psect that you define may need flags (options) to be specified. Take particular notice of the `delta`, `space`, `reloc` and `class` flags (see 6.1.9.36.4 Delta Flag and 6.1.9.36.18 Space Flag 6.1.9.36.16 Reloc Flag and 6.1.9.36.3 Class Flag). If these are not set correctly, compile errors or code failure will almost certainly result. If the psect specifies a class and that psect can be placed anywhere in the memory range defined by that class (see 7.1.1 A: Define Linker Class), you do not need to specify any options for it to be linked; otherwise, you will need to link the psect using a linker option (see 7.1.17 P: Position Psect for the usual way to link psects and 4.6.11.8 Wl: Pass Option To The Linker, Option which indicates how you can specify this option without running the linker directly).
Assembly code that is placed in-line with C code will be placed in the same psect as the compiler-generated assembly and you should not place this into a separate psect.
- You must ensure that any registers you write to in assembly code are not already in used by compiler-generated code. If you write assembly in a separate module, then this is less of an issue because the compiler will, by default, assume that all registers are used by these routines (see 5.6 Register Usage). No assumptions are made for in-line assembly (although the compiler will assume that the selected bank was changed by the assembly, see 5.11.2 Inline Assembly) and you must be careful to save and restore any resources that you use (modify) and which are already in use by the surrounding compiler-generated code.

3.4 Getting My Application To Do What I Want

This section provides programming techniques, applications and examples. It also examines questions that relate to making an application perform a specific task.

3.4.1 What Can Cause Glitches on Output Ports?

In most cases, this is caused by using ordinary variables to access port bits or the entire port itself. These variables should be qualified `volatile`.

The value stored in a variable mapped over a port (hence the actual value written to the port) directly translates to an electrical signal. It is vital that the values held by these variables only change when the code intends them to and that they change from their current state to their new value in a single transition (see 5.3.8.2 Volatile Type Qualifier). The compiler attempts to write to `volatile` variables in one operation.

3.4.2 Where Am I Allowed To Manually Link Psects?

It is recommended that the linker options for compiler-generated psects (sections) are not modified. If these must be changed or if there are user-defined psects that need special allocation, there might be device- or compiler-imposed restrictions on where these can be placed in memory.

Try to link psects in a suitable compiler linker class (as shown in 5.14.2 Default Linker Classes) as the definitions for these memory ranges take into consideration any restrictions. Define your own linker class, if necessary (see 3.3.3.3 How Do I Position A Variable Into An Address Range?). Refer to 5.14.1 Compiler-Generated Psects to see the memory placement restrictions that apply to compiler-generated psects which hold similar content to your psect.

Most limitations relate to psects straddling some memory boundary, such as a data bank or program memory page. One typical limitation is that all psects holding executable code cannot straddle a device page boundary. Compiler-generated psects holding variables must also be typically linked within the data bank for which they were created. These boundaries, therefore, impose limits on the size to which the psect can grow.

3.4.3 How Do I Link Bootloaders and Downloadable Applications?

Exactly how this is done depends on the device you are using and your project requirements, but the general approach when compiling applications that use a bootloader is to allocate discrete program memory space to the bootloader and application so they have their own dedicated memory. In this way the operation of one cannot affect the other. This will require that either the bootloader or the application is offset in memory. That is, the Reset and interrupt location are offset from address 0 and all program code is offset by the same amount.

On PIC18 devices, the application code is typically offset and the bootloader is linked with no offset so that it populates the Reset and interrupt code locations. The bootloader Reset and interrupt code merely contains code which redirects control to the real Reset and interrupt code defined by the application and which is offset.

On mid-range devices, this is not normally possible to perform when interrupts are being used. Consider offsetting all of the bootloader with the exception of the code associated with Reset, which must always be defined by the bootloader. The application code can define the code linked at the interrupt location. The bootloader will need to remap any application code that attempts to overwrite the Reset code defined by the bootloader.

The option `-mcodeoffset`, (see 4.6.1.3 Codeoffset Option), allows the program code (Reset and vectors included) to be moved by a specified amount for devices that do not use the VIC module. The option also restricts the program from using any program memory from address 0 (Reset vector) to the offset address. Always check the map file (see 7.3 Map Files), to ensure that nothing remains in reserved areas.

For devices with the VIC module (even those operating in legacy mode) you will need to adjust the IVTBASE register to move the hardware vector locations.

The contents of the HEX file for the bootloader can be merged with the code of the application by adding the HEX file as a project file, either on the command line, or in MPLAB X IDE. This results in a single HEX file that contains the bootloader and application code in the one image. HEX files are merged by the HEXMATE application (see 8.2 Hexmate). Check for warnings from this application about overlap, which can indicate that memory is in use by both bootloader and the downloadable application.

3.4.4 What Do I Need to Do When Compiling to Use a Debugger?

You can use debuggers, such as the MPLAB ICD4 or PICkit 4, to debug code built with the MPLAB XC8 compiler. These debuggers might use some of the data and program device memory for its own use. If this is the case, it is important that your code does not also use these resources.

In the MPLAB X IDE, the appropriate memory is reserved when you perform a debug build. All the device memory will be available when you perform a regular Build Project or Clean and Build. If you are building on the command-line, then you must communicate to the compiler that you will be using a debugger. The compiler's `-mdebugger` option will do this. (See 4.6.1.6 Debugger Option.)

The `ReservedResourcesByDeviceFamilyAndTool.html` file in the MPLAB X IDE's `docs/ReservedResources` directory lets you check the memory and other resources used by each device for different debugger tools and that will be reserved by the `-mdebugger` option.

When device memory is being used by a debugger, there is less memory available for your program and it is possible that your code or data might not fit in the device when a debugger is selected. If you are only using a debugger tool as a programmer, then you do not need to use the `-mdebugger` option, as no device memory will be required by the tool.

To verify that the resources reserved by the compiler match those required by the debugger, do the following. Compile your code with and without the debugger selected and keep a copy of the map file produced for both builds. Compare the linker options in the map files and look for changes in the `-A` options (see 7.1.1 A: Define Linker Class). For example, the memory defined for the `CODE` class with no debugger might be specified by this option:

```
-ACODE=00h-0FFh,0100h-07FFh,0800h-0FFFhx3
```

and with the ICD3 selected as the debugger, it becomes:

```
-ACODE=00h-0FFh,0100h-07FFh,0800h-0FFFhx2,01800h-01EFFh
```

This shows that a memory range from 1F00 to 1FFF has been removed by the compiler and cannot be used by your program (See also 3.5.16 Why Are Some Objects Positioned Into Memory That I Reserved?).

3.4.5 How Can I Have Code Executed Straight After Reset?

A special hook has been provided so you can easily add special “powerup” assembly code that will be linked to the Reset vector (see 5.9.3 The Powerup Routine). This code will be executed before the runtime startup code, which in turn is executed before the `main()` function (see 5.9 Main, Runtime Startup and Reset).

3.4.6 How Do I Share Data Between Interrupt and Main-line Code?

Variables accessed from both interrupt and main-line code can easily become corrupted or mis-read by the program. The `volatile` qualifier (see 5.3.8.2 Volatile Type Qualifier) tells the compiler to avoid performing optimizations on such variables. This will fix some of the issues associated with this problem.

The other issue relates to whether the compiler/device can access the data atomically. With 8-bit PIC devices, this is rarely the case. An atomic access is one where the entire variable is accessed in only one instruction. Such access is uninterruptible. You can determine if a variable is being accessed atomically by looking at the assembly code the compiler produces in the assembly list file (see 6.3 Assembly List Files). If the variable is accessed in one instruction, it is atomic. Since the way variables are accessed can vary from statement to statement it is usually best to avoid these issues entirely by disabling interrupts prior to the variable being accessed in main-line code, then re-enable the interrupts afterwards (see 5.8.5 Enabling Interrupts).

3.4.7 How Can I Prevent Misuse of My Code?

First, many devices with flash program memory allow all or part of this memory to be write protected. The device Configuration bits need to be set correctly for this to take place; see 5.2.5 Configuration Bit Access and your device data sheet.

Second, you can prevent third-party code being programmed at unused locations in the program memory by filling these locations with a value rather than leaving them in an unprogrammed state. You can choose a fill value that corresponds to an instruction or set all the bits so values cannot be further modified (consider what will happen if your program somehow reaches and starts executing from these filled values).

The compiler's HEXMATE utility (see 8.2 Hexmate) has the capability to fill unused locations and can be requested using a command-line driver option (see 4.6.11.10 Fill Option). As HEXMATE only works with HEX files, this feature is only available when producing HEX/COF file outputs (as opposed to binary, for example), which is the default operation.

And last, if you wish to make your library files or intermediate p-code files available to others but do not want the original source code to be viewable, then you can obfuscate the files using the `-mshroud` option (see 4.6.1.21 Shroud Option).

3.4.8 How Do I Use Printf to Send Text to a Peripheral?

The `printf()` function does two things:

1. Formats text based on the format string and conversion specifiers you specify.
2. Sends (prints) this formatted text to a destination (or stream).

See 5.10.2 Smart IO Routines and the *Microchip Unified Standard Library Reference Guide* for more information.

The `printf()` function performs all the formatting; then calls a helper function called `putch()`, to send each byte of the formatted text. By customizing the `putch()` function you can have `printf()` send data to any peripheral or location. You can choose the `printf()` output go to an LCD, SPI module, or USART for example.

A stub for the `putch()` function can be found in the compiler's `pic/sources` directory. Copy it into your project, then modify it to send the single byte parameter passed to it to the required destination. Before you can use `printf()`, peripherals that you use will need to be initialized in the usual way. Here is an example of `putch()` for a USART on a mid-range PIC device.

```
void putch(char data) {
  while( ! TXIF)   // check buffer
    continue;      // wait till ready
  TXREG = data;    // send data
}
```

You can get `printf()` to send to one of several destinations by using a global variable to indicate your choice. Have the `putch()` function send the byte to one of several destinations based on the contents of this variable.

3.4.9 How Do I Setup the Oscillator in My Code?

All PIC devices have several oscillator modes that must be selected by programming the device's configuration bits in your project. See your device data sheet for information on the modes and 5.2.5 Configuration Bit Access for assistance with programming the configuration bits.

Some devices have an OSCCON register, which further controls such runtime attributes as clock sources and internal clock frequencies. In C source, this register can be written to in the usual way, based on information in your device data sheet.

Some devices allow the internal oscillator to be tuned at runtime via the OSCTUNE register. Other devices allow for calibration of their internal oscillators using values pre-programmed into the device. The runtime startup code generated by the compiler, (see 5.9.2 Runtime Startup Code), will by default provide code that performs oscillator calibration. This can be disabled, if required, using an option (see 4.6.1.15 Osccal Option).

If you intend to use some of the compiler's built-in delay functions, you will need to set the `_XTAL_FREQ` macro, which indicates the system frequency to the compiler. This macro in no way affects the operating frequency of the device (see, for example 9.2.14 __delay_ms Builtin).

3.4.10 How Do I Place Variables in the PIC18 Device's External Program Memory?

If all you mean to do is place read-only variables in program memory, qualify them as `const` (see 5.4.3 Objects in Program Space). If you intend the variables to be located in the external program memory then use the `__far` qualifier (see 5.3.9.3 Far Type Qualifier) and specify the memory using the `-mram` option (see 4.6.1.17 Ram Option). The compiler will allow `__far`-qualified variables to be modified. Note that the time to access these variables will be longer than for variables in the internal data memory. The access mode to external memory can be specified with an option (see 4.6.1.9 Emi Option).

3.4.11 How Can I Implement a Delay in My Code?

If an accurate delay is required, or if there are other tasks that can be performed during the delay period, then using a timer to generate an interrupt is the best way to proceed.

If these are not issues in your code, then you can use the compiler's in-built delay pseudo-functions: `_delay()`, `__delay_ms()` or `__delay_us()` (see 9.2.9 _delay Builtin). These all expand into in-line assembly instructions or a (nested) loop of instructions that will consume the specified number of cycles or time. The delay argument must be a constant expression (i.e. it cannot contain variables or function calls) and evaluate to less than 50,463,240. To use the `__delay_ms()` or `__delay_us()` versions of the delay, the preprocessor macro `_XTAL_FREQ` must be correctly defined to match the device clock frequency.

Note that these code sequences will only use the `nop` instruction and/or instructions which form a loop. The alternate watchdog versions of these pseudo-functions, e.g., `_delaywdt()`, can use the `clrwdt` instruction as well.

3.4.12 How Can I Rotate a Variable?

The C language does not have a rotate operator, but rotations can be performed using the shift and bitwise OR operators. Since the PIC devices have a rotate instruction, the compiler will look for code expressions that implement rotations (using shifts and ORs) and use the rotate instruction in the generated output wherever possible (see 5.5.2 Rotation).

3.4.13 How Can I Stop Variables Being Cleared at Startup?

Use the `__persistent` qualifier (see 5.3.9.5 Persistent Type Qualifier), which will place the variables in a different psect that is not cleared by the runtime startup code.

3.4.14 How Do I Use High-Endurance Flash for Data, Not Code?

For devices that implement High-endurance Flash memory in the program memory space, the memory will need to be reserved so that the compiler does not use it for executable code (see 3.3.3.8 How Do I Stop The Compiler From Using Certain Memory Locations?). Your device data sheet will indicate if your device has this memory implemented.

3.5 Understanding the Compilation Process

This section tells you how to find out what the compiler did during the build process, how it encoded output code, where it placed objects, etc. It also discusses the features that are supported by the compiler.

- What's the Difference Between a licensed and unlicensed compiler?
- How Can I Make My Code Smaller?
- How Can I Reduce RAM Usage?
- How Can I Make My Code Faster?
- How Can I Speed Up Programming Times?
- How Does the Compiler Place Everything in Memory?
- How Can I Make My Interrupt Routine Faster?
- How Big Can C Variables Be?
- How Do I Utilize/Allocate the RAM Banks on My Device?
- How Do I Utilize the Linear Memory on Enhanced Mid-range PIC Devices?
- What Devices are Supported by the Compiler?
- How Do I Know What Code the Compiler Is Producing?
- How Can I Tell How Big a Function Is?
- How Do I Know Which Resources Are Being Used by Each Function?
- How Do I Find Out Where Variables and Functions Have Been Positioned?
- Why Are Some Objects Positioned Into Memory That I Reserved?
- How Do I Know How Much Memory Is Still Available?
- How Do I Use Library Files in My Project?
- What Optimizations Are Employed by the Compiler?
- Why Do I Get Out-of-memory Errors When I Select a Debugger?
- How Do I Know Which Stack Model the Compiler Has Assigned to a Function?
- How Do I Know What Value Has Been Programmed in the Configuration Bits or ID Location?
- How Do I Stop My Project's Checksum From Changing?

See also the following linked information in other sections.

- How Do I Find Out What a Warning/Error Message Means?
- What is Different About an MPLAB X IDE Debug Build?
- How Do I Stop An Unused Function Being Removed?
- How Do I Build Libraries?

3.5.1 What's the Difference Between a licensed and unlicensed compiler?

A license permits a higher level of optimization to be employed by the compiler, thus code produced by an unlicensed compiler will be larger and take longer to execute. Unlicensed compiler are not restricted in any other way, such as in allowable target devices or in the utilization of available device memory.

3.5.2 How Can I Make My Code Smaller?

There are a number of ways that this can be done, but results vary from one project to the next. Use the assembly list file (see 6.3 Assembly List Files) to observe the assembly code, produced by the compiler, to verify that the following tips are relevant to your code.

Use the smallest data types possible, as less code is needed to access these (this also reduces RAM usage). Note that a bit type and non-standard 24-bit integer type exists for this compiler. Avoid multi-bit bit-fields whenever possible. The code used to access these can be very large. See 5.3 Supported Data Types and Variables for all data types and sizes.

There are two sizes of floating-point type, with these being discussed in the same section as well. Avoid floating-point if at all possible. Consider writing fixed-point arithmetic code.

Use unsigned types, if possible, instead of signed types; particularly if they are used in expressions with a mix of types and sizes. Try to avoid an operator acting on operands with mixed sizes whenever possible.

Whenever you have a loop or condition code, use a “strong” stop condition, i.e., the following:

```
for(i=0; i!=10; i++)
```

is preferable to:

```
for(i=0; i<10; i++)
```

A check for equality (`==` or `!=`) is usually more efficient to implement than the weaker `<` comparison.

In some situations, using a loop counter that decrements to zero is more efficient than one that starts at zero and counts up by the same number of iterations. This is more likely to be the case if the loop index is a byte-wide type. So you might be able to rewrite the above as:

```
for(i=10; i!=0; i--)
```

There might be a small advantage in changing the order of function parameters so that the first parameter is byte sized. A register is used if the first parameter is byte-sized. For example consider:

```
char calc(char mode, int value);
```

over

```
char calc(int value, char mode);
```

Ensure that all optimizations are enabled (see 4.6.6 Options for Controlling Optimization). Be aware of what optimizations the compiler performs (see 5.12 Optimizations and 6.2 Assembly-Level Optimizations) so you can take advantage of them and don't waste your time manually performing optimizations in C code that the compiler already handles, e.g., don't turn a multiply-by-4 operation into a shift-by-2 operation as this sort of optimization is already detected.

3.5.3 How Can I Reduce RAM Usage?

Use the smallest data types possible (this also reduces code size as less code is needed to access these). Note that a bit type and non-standard 24-bit integer type (`__int24` and `__uint24`) exists for this compiler. See 5.3 Supported Data Types and Variables for all data types and sizes. There are two sizes of floating-point type that are discussed in the same section as well.

Consider using auto variables over global or static variables as there is the potential that these can share memory allocated to other auto variables that are not active at the same time. Memory allocation of auto variables is made on a compiled stack (described in 5.4.2.2 Automatic Storage Duration Objects).

Rather than pass large objects to or from, functions pass pointers which reference these objects. This is particularly true when larger structures are being passed, but there might be RAM savings to be made even when passing long variables.

Objects that do not need to change throughout the program can be located in program memory using the `const` qualifier (see 5.3.8.1 Const Type Qualifier and 5.4.2 Objects in Data Memory). This frees up precious RAM, but slows execution.

Ensure that all optimizations are enabled (see 4.6.6 Options for Controlling Optimization). Be aware of which optimizations the compiler performs (see 5.12 Optimizations), so that you can take advantage of them and don't waste your time manually performing optimizations in C code that the compiler already handles.

3.5.4 How Can I Make My Code Faster?

To a large degree, smaller code is faster code, so efforts to reduce code size often decrease execution time (see Section 2.6.2 "How Can I Make My Code Smaller?" and Section 2.6.7 "How Can I Make My Interrupt Routine Faster?"). However, there are ways some sequences can be sped up at the expense of increased code size.

When level O3 optimizations have been selected, the compiler favors optimizations that reduce the program's execution time (but which might increase the program's size), whereas level Os optimizations are designed to purely reduce code size (see 4.6.6 Options for Controlling Optimization).

Some library multiplication routines operate faster when one of their operands is a smaller value (see 5.2.9 Multiplication for more information on how to take advantage of this).

Generally, the biggest gains to be made in terms of speed of execution come from the algorithm used in a project. Identify which sections of your program need to be fast. Look for loops that might be linearly searching arrays and choose an alternate search method such as a hash table and function. Where results are being recalculated, consider if they can be cached.

3.5.5 How Can I Speed Up Programming Times?

The linker can allocate sections to both ends of program memory: some sections initially placed at a low address and built up through memory; other sections assembled at a high address and extended down. This does not affect code operation and makes linking easier, but it can produce a HEX file covering the entire device memory space. Programming this HEX file into the device may take a long time.

To reduce programming times in this situation, instruct the linker to not use all the device's program memory. Use the `-mreserve` option to reserve the upper part of program memory (see 4.6.1.18 Reserve Option).

3.5.6 How Does the Compiler Place Everything in Memory?

In most situations, assembly instructions and directives associated with both code and data are grouped into sections, called psects, which are then positioned into containers that represent the device memory. An introductory explanation into this process is given in 5.14.1 Compiler-Generated Psects. The exception is for absolute variables (see 5.4.4 Absolute Variables), which are placed at a specific address when they are defined and which are not placed in a psect.

3.5.7 How Can I Make My Interrupt Routine Faster?

Consider suggestions made in Section 3.5.2 "How Can I Make My Code Smaller?" (code size) for any interrupt code. Smaller code is often faster code.

In addition to the code you write in the ISR, there is the code the compiler produces to switch context. This is executed immediately after an interrupt occurs and immediately before the interrupt returns, meaning it must be included in the time taken to process an interrupt (see 5.8.4 Context Switching). This code is typically optimal, in that only registers used in the ISR will be saved by this code. Thus, the fewer registers that are used in your ISR means that potentially less context switch code will be executed. Register use increases with the complexity of code, so avoid complex statements and calls to functions that might also contain complex code. Use the assembly list file to see which registers are being used by the compiler in the interrupt code (see 6.3 Assembly List Files).

Mid-range devices have only a few registers that are used by the compiler and there is little context switch code. Some devices save context automatically into shadow registers, which further reduces (or eliminates entirely) the compiler-generated switch code (see 5.6 Register Usage).

Consider having the ISR simply set a flag and return. The flag can then be checked in main-line code to handle the interrupt. This has the advantage of moving the complicated interrupt-processing code out of the ISR so that it no longer contributes to its register usage. Always use the `volatile` qualifier (see 5.3.8.2 Volatile Type Qualifier for variables shared by the interrupt and main-line code; see Section 3.4.6 "How Do I Share Data Between Interrupt and Main-line Code?").

3.5.8 How Big Can C Variables Be?

This question specifically relates to the size of individual C objects, such as arrays or structures. The total size of all variables is another matter.

To answer this question you need to know in which memory space the variable will be located. Objects with static storage duration and that are qualified `const` will be located in program memory; other objects will be placed in data memory. Program memory object sizes are discussed in 5.4.3.1 Object Size Limitations. Objects in data memory are broadly grouped into autos and non-autos and the size limitations of these objects (see 5.4.2.1.2 Object Size Limits and 5.4.2.2.1 Object Size Limits).

3.5.9 How Do I Utilize/Allocate the RAM Banks on My Device?

The compiler will automatically use all the available RAM banks on the device you are programming. It is only if you wish to alter the default memory allocation that you need take any action. Special `bank()` qualifiers (see 5.3.9.1 Bank Type Qualifier) and an option (see 4.6.1.1 Addrqual Option) to indicate how these qualifiers are interpreted are used to manually allocate variables.

Note that there is no guarantee that all the memory on a device can be utilized as data and code is packed in sections, or psects.

3.5.10 How Do I Utilize the Linear Memory on Enhanced Mid-range PIC Devices?

The linear addressing mode is a means of accessing the banked data memory as one contiguous and linear block on enhanced mid-range devices (see 5.4.1 Address Spaces). Your device data sheet will indicate if this memory is implemented on your device and contain further operational details.

Use of the linear memory is fully automatic. Objects that are larger than a data bank can be defined in the usual way and will be accessed using the linear addressing mode (see 5.4.2.1.2 Object Size Limits). If you define absolute objects at a particular location in memory, you can use a linear address if you prefer, or the regular banked address (see 5.4.4.1 Absolute Objects In Data Memory).

3.5.11 What Devices are Supported by the Compiler?

Support for new devices usually takes place with each compiler release. There are several ways to find out whether a device is supported by your compiler (see also, 5.2.1 Device Support); two of which are as follows:

- HTML listings are provided in the compiler's `docs` directory. Open these in your favorite web browser. They are called `pic_chipinfo.html` and `pic18_chipinfo.html`.
- Run the compiler driver on the command line with the `-mprint-devices` option (see 4.6.2.8 Print-devices). A full list of all devices is printed to the screen.

3.5.12 How Do I Know What Code the Compiler Is Producing?

The assembly list file (see 6.3 Assembly List Files) shows the assembly output for almost the entire program, including library routines linked in to your program, as well a large amount of the runtime startup code (see 5.9.2 Runtime Startup Code). If you are using the command-line, the option `-Wa,-a` will produce this file for you (see 4.6.10 Mapped Assembler Options). The assembly list file will have a `.lst` extension.

The list file shows assembly instructions, some assembly directives and information about the program, such as the call graph, pointer reference graph and information for every function. Not all assembly directives are shown in the list file if the assembly optimizers are enabled (they are produced in the intermediate assembly file). Temporarily disable the assembly optimizers (see 4.6.6 Options for Controlling Optimization), if you wish to see all the assembly directives produced by the compiler.

3.5.13 How Can I Tell How Big a Function Is?

Information that includes the size of functions is presented in the map file. Look for the header "MODULE INFORMATION" near the bottom of the file. This information is discussed in 7.3.2.8 Module Information.

3.5.14 How Do I Know Which Resources Are Being Used by Each Function?

In the assembly list file there is information printed for every C function, including library functions (see 6.3 Assembly List Files). This information indicates what registers the function used, what functions it calls (this is also found in the call graph; see 6.3.6 Call Graph and how many bytes of data memory it requires. Note that auto, parameter and temporary variables used by a function can overlap with those from other functions as these are placed in a compiled stack by the compiler (see 5.4.2.2 Automatic Storage Duration Objects).

3.5.15 How Do I Find Out Where Variables and Functions Have Been Positioned?

You can determine where variables and functions have been positioned from either the assembly list file (see 6.3 Assembly List Files), or the map file (see 7.3 Map Files). Only symbols associated with objects with static storage duration are shown in the map file; all symbols (including those with automatic storage duration) are listed in the assembly list file, but only for the code represented by that list file. Each assembly module has its own list file.

There is a mapping between C identifiers and the symbols used in assembly code, which are the symbols shown in both of these files (see 5.11.3.3 Equivalent Assembly Symbols). The symbol associated with a variable is assigned the address of the lowest byte of the variable; for functions it is the address of the first instruction generated for that function.

3.5.16 Why Are Some Objects Positioned Into Memory That I Reserved?

The memory reservation options (see Section 2.4.3.6 "How Do I Place a Function Into a Unique Section?") will adjust the range of addresses associated with classes used by the linker. Most variables and function are placed into sections (see 5.14.1 Compiler-Generated Psects) that are linked anywhere inside these class ranges and so are affected by these reservation options.

Some sections are explicitly placed at an address rather than being linked anywhere in an address range, e.g., the sections that holds the code to be executed at Reset is always linked to address 0 because that is where the Reset location is defined to be for 8-bit devices. Such a section will not be affected by the `-mrom` option, even if you use it to reserve memory address 0. Sections that hold code associated with Reset and interrupts can be shifted using the `-mcodeoffset` option (see Section 4.6.1.3 "Codeoffset Option").

Check the assembly list file (see 6.3 Assembly List Files) to determine the names of sections that hold objects and code. Check the linker options in the map file (see 7.3 Map Files), to see if psects have been linked explicitly or if they are linked anywhere in a class. See also, the linker options `-p` (7.1.17 P: Position Psect) and `-A` (7.1.1 A: Define Linker Class).

3.5.17 How Do I Know How Much Memory Is Still Available?

Although the memory summary, printed by the compiler after compilation, and the memory display, available in MPLAB X IDE, both indicate the amount of memory used and the amount still available, neither feature indicates whether this memory is one contiguous block or broken into many small chunks. Small blocks of free memory cannot be used for larger objects and so out-of-memory errors can be produced even though the total amount of memory free is apparently sufficient for the objects to be positioned (see 3.6.6 How Do I Fix a "Can't find space..." Error?).

The 7.3.2.5 Unused Address Ranges section in the map file indicates exactly what memory is still available in each linker class. It also indicated the largest contiguous block in that class if there are memory bank or page divisions.

3.5.18 How Do I Use Library Files in My Project?

The compiler will automatically include any applicable standard library into the build process when you compile, so you never need to control these files. See 3.2.5 How Do I Build Libraries? for information on how you build your own library files.

To use one or more library files that were built by yourself or a colleague, include them in the list of files being compiled on the command line. The library files can be specified in any position in the file list relative to the source files, but if there is more than one library file, they will be searched in the order specified in the command line. For example:

```
xc8-cc -mcpu=16f1937 main.c int.c lcd.a
```

If you are using MPLAB X IDE to build a project, add the library file(s) to the Libraries folder shown in your project, in the order in which they should be searched. The IDE will ensure that they are passed to the compiler at the appropriate point in the build sequence.

3.5.19 What Optimizations Are Employed by the Compiler?

Optimizations are employed at both the C and assembly level of compilation. These are described in 5.12 Optimizations and 6.2 Assembly-Level Optimizations respectively. The options that control optimization are described in 4.6.6 Options for Controlling Optimization.

3.5.20 Why Do I Get Out-of-memory Errors When I Select a Debugger?

If you use a hardware tool debugger, such as the MPLAB REAL ICE or ICD 3, these might require some of the memory resources available on your PIC device for the on-board debug executive. When you select a debugger within an MPLAB X IDE project and perform a debug build, the memory required for debugging is removed from that available. In some situations, for programs that use a large amount of memory, this might be enough to trigger a 'can't find space' memory error.

3.5.21 How Do I Know Which Stack Model the Compiler Has Assigned to a Function?

Look in the function information section in the assembly list file (see 6.3.3 Function Information). The last line of this block will indicate whether the function uses a reentrant or non-reentrant model.

3.5.22 How Do I Know What Value Has Been Programmed in the Configuration Bits or ID Location?

Check the assembly list file for the output of the `#pragma config` directives. You will see the numerical value programmed to the appropriate locations. In the following example, the configuration value programmed is 0x1F. A breakdown of what this value means is also printed in this file.

```
;Config register CONFIG2H @ 0x300003
;       Watchdog Timer Enable bit
; ...
;       WDTPS = 0xF, unprogrammed default
psect   config
org 3145731
db  31
```

3.5.23 How Do I Stop My Project's Checksum From Changing?

The checksum that represents your built project, whether this is generated by the MPLAB X IDE or by tools such as Hexmate (see 8.2 Hexmate), is calculated from the generated output of the compiler. Indeed, the algorithms used to obtain the checksum are specifically designed so that even small changes in this output are almost guaranteed to produce a different checksum result. Checksums are not calculated from your project's source code. To ensure that your checksum does not change from build to build, you must ensure that the output of the compiler does not change.

The following actions and situations could cause changes in the compiled output and hence changes in your project's checksum.

- Changing the compiler version between builds.
- Changing the compiler options between builds.
- Changing the source code, header files, or library code used by the project between builds.
- Changing the order in which source files or libraries are compiled or linked between builds.
- Having source code that makes using of macros such as `__DATE__` and `__TIME__`, which produce output that is dependent on when the project was built.
- Moving the location of source files between builds, where those files use macros such as `__FILE__`, which produces output that is dependent on where the source file is located.

Note that the checksum algorithms used by tools such as Hexmate and the MPLAB X IDE can change, which can result in a different checksum for the same compiler output. Such changes are rare, but check the compiler and IDE release notes to see if the tools have been modified.

3.6 Fixing Code That Does Not Work

This section examines issues relating to projects that do not build due to compiler errors, or those that build, but do not work as expected.

3.6.1 How Do I Find Out What an Warning/Error Message Means?

Each warning or error message has a description and possibly sample code that might trigger such an error, listed in the messages chapter (see 10. Error and Warning Messages).

3.6.2 How Do I Find the Code that Caused Compiler Errors or Warnings in My Program?

In most instances, when a syntax error occurs that relates to the source code, the message produced by the compiler indicates the offending line of code (see 4.5.1 Messaging Overview). If you are compiling in MPLAB X IDE, then you can double-click the message and have the editor take you to the offending line. But identifying the offending code is not always so easy.

In some instances, the error is reported on the line of code following the line that needs attention. This is because a C statement is allowed to extend over multiple lines of the source file. It is possible that the compiler cannot be able to determine that there is an error until it has started to scan to statement following. So in the following code

```
input = PORTB   // oops - forgot the semicolon
if(input>6)
// ...
```

The missing semicolon on the assignment statement will be flagged on the following line that contains the `if()` statement.

In other cases, the error might come from the assembler, not the code generator. If the assembly code was derived from a C source file, then the compiler will try to indicate the line in the C source file that corresponds to the assembly that is at fault. If the source being compiled is an assembly module, the error directly indicates the line of assembly that triggered the error. In either case, remember that the information in the error relates to some problem is the assembly code, not the C code.

Finally, there are errors that do not relate to any particular line of code at all. An error in a compiler option or a linker error are examples of these. If the program defines too many variables, there is no one particular line of code that is at fault; the program as a whole uses too much data. Note that the name and line number of the last processed file and source can be printed in some situations even though that code is not the direct source of the error.

To determine the application that generated the error or warning, check the message section of the manual, see 10. Error and Warning Messages. At the top of each message description, on the right in brackets, is the name of the application that produced this message. Knowing the application that produced the error makes it easier to track down the problem. The compiler application names are indicated in 4.2 The Compilation Sequence. If you need to see the assembly code generated by the compiler, look in the assembly list file (see 6.3 Assembly List Files). For information on where the linker attempted to position objects, see the map file discussed in 7.3 Map Files.

3.6.3 How Can I Stop Spurious Warnings From Being Produced?

Warnings indicate situations that could possibly lead to code failure. In many situations the code is valid and the warning is superfluous. Always check your code to confirm that it is not a possible source of error and in cases where this is so, there are several ways that warnings can be hidden.

- The warning level threshold can be adjusted so that only warnings of a certain importance are printed (see 4.5.3.1 Disabling Messages).
- All warnings with a specified ID can be inhibited.
- In some situations, a pragma can be used to inhibit a warning with a specified ID for certain lines of source code (see 5.13.3.11 The #pragma Warning Directive).

3.6.4 Why Can't I Even Blink an LED?

Even if you have set up the TRIS register and written a value to the port, there are several things that can prevent such a seemingly simple program from working.

- Make sure that the device's Configuration registers are set up correctly (see 5.2.5 Configuration Bit Access). Make sure that you explicitly specify every bit in these registers and don't just leave them in their default state. All the configuration features are described in your device data sheet. If the Configuration bits that specify the oscillator source are wrong, for example, the device clock cannot even be running.
- If the internal oscillator is being used, in addition to Configuration bits there can be SFRs you need to initialize to set the oscillator frequency and modes, see 5.2.7 Using SFRs From C Code and your device data sheet.
- Either turn off the Watchdog Timer in the Configuration bits or clear the Watch Dog Timer in your code so that the device does not reset. If the device is resetting, it can never reach the lines of code in your program that blink the LED. Turn off any other features that can cause device Reset until your test program is working.
- The device pins used by the port bits are often multiplexed with other peripherals. A pin might be connected to a bit in a port, or it might be an analog input, or it might the output of a comparator, for example. If the pin connected to your LED is not internally connected to the port you are using, then your LED will never operate as expected. The port function tables shown in your device data sheets will show other uses for each pin that will help you identify peripherals to investigate.
- Make sure you do not have a "read-modify-write" problem. If the device you are using does not have a separate "latch" register (as is the case with mid-range PIC devices) this problem can occur, particularly if the port outputs are driving large loads, such as an LED. You can see that setting one bit turns off another or other unusual events. Create your own latch by using a temporary variable. Rather than read and write the port directly, make modifications to the latch variable. After modifications are complete, copy the latch as a whole to the port. This means you are never reading the port to modify it. Check the device literature for more detailed information.

3.6.5 How Do I Know If the Hardware Stack Has Overflowed?

An 8-bit PIC device has a limited hardware stack that is used only for function (and interrupt function) return addresses (see 5.2.4 Stacks). If the nesting of function calls and interrupts is too deep, the stack will overflow (wraps around and overwrites previous entries). Code will then fail at a later point – sometimes much later in the call sequence – when it accesses the corrupted return address.

The compiler attempts to track stack depth and, when required, swap to a method of calling that does not need the hardware stack (PIC10/12/16 devices only). You have some degree of control over what happens when the stack depth has apparently overflowed, see 4.6.1.23 Stackcall Option for information on the `-mstackcall` option.

A call graph shows the call hierarchy and depth that the compiler has determined. This graph is shown in the assembly list file. To understand the information in this graph, see 6.3.6 Call Graph.

Since the runtime behavior of the program cannot be determined by the compiler, it can only assume the worst case and can report that overflow is possible even though it is not. However, no overflow should go undetected if the program is written entirely in C. Assembly code that uses the stack is not considered by the compiler and this must be taken into account.

3.6.6 How Do I Fix a "Can't find space..." Error?

There are a number of different variants of this message, but all essentially imply a similar situation. They all relate to there being no free space large enough to place a block of data or instructions. Due to memory paging, banking or other fragmentation, this message can be issued when seemingly there is enough memory remaining. See 10. Error and Warning Messages for more information on your particular error number.

3.6.7 How Do I Fix a "Can't generate code..." Error?

This is a catch-all message which is generated if the compiler has exhausted all possible means of compiling a C expression, see 10. Error and Warning Messages. It does not usually indicate a fault in your code. The inability to compile the code can be a deficiency in the compiler, or an expression that requires more registers or resources than are available at that point in the code. This is more likely to occur on baseline devices. In any case, simplifying the offending expression, or splitting a statement into several smaller statements, usually allows the compilation to continue. You may need to use another variable to hold the intermediate results of complicated expressions.

3.6.8 How Do I Fix a Fixup Overflow Error?

Fixup – the linker action of replacing a symbolic reference with an actual address – can overflow if the address assigned to the symbol is too large to fit in the address field of an assembly instruction. Most 8-bit PIC assembly instructions specify a file address that is an offset into the currently selected memory bank. If a full unmasked address is specified with these instructions, the linker will be unable to encode the large address value into the instruction and this error will be generated. For example, a mid-range device instruction only allows for file addresses in the range of 0 to 0x7F. However, if such a device has 4 data banks of RAM, the addresses of variables can range from 0 to 0x1FF.

For example, if the symbol of a variable that will be located at address 0x1D0 has been specified with one of these instructions, then when the symbol is replaced with its final value, this value will not fit in the address field of the instruction.

Many of the jump and call instructions also take a destination operand that is a truncated address. The PIC18 `call` and `goto` instructions work with a full address, but the branch and relative call instructions do not. If the destination label to any of these instructions is not masked, a fixup error can result.

The fixup process applies to the operands of assembler directives, as well as instructions; so if the operand to a directive overflows, a fixup error can also result. For example, if the symbol error is resolved by the linker to be the value 0x238, the directive:

```
DB error
```

which expects a byte value, will generate a fixup overflow error.

In most cases, fixup errors are caused by hand-written assembly code. When writing assembly, it is the programmer's responsibility to add instructions to select the destination bank or page, then mask the address being used in the instruction (see Section 2.4.7.5 "What Things Must I Manage When Writing Assembly Code?").

In some situations assembly code generated from C code can produce a fixup overflow message. Typically this will be related to jumps that are out of range. C switch statements that have become too large can trigger such a message. Changing how a compiler-generated psect is linked can also cause fixup overflow, as the new psect location may break an assumption made by the compiler.

It is important to remember that this is an issue with an assembly instruction, and that you need to find the instruction at fault before you can proceed. See the relevant error number in 10. Error and Warning Messages for specific details about how to track down the offending instruction.

3.6.9 What Can Cause Corrupted Variables and Code Failure When Using Interrupts?

This is usually caused by having variables used by both interrupt and main-line code. If the compiler optimizes access to a variable or access is interrupted by an interrupt routine, then corruption can occur. See 3.4.6 How Do I Share Data Between Interrupt and Main-line Code? for more information.

4. 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 and its command-line options, you may instead run the `xc8` driver application. It's use is described in its own user's guide, MPLAB® XC8 C Compiler User's Guide, which also covers the C90 aspect of compilation.

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.

4.1 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.

4.1.1 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=16F877A -O2 -o hello.elf hello.c
```

Throughout this manual, it is assumed that the compiler applications are in your console's search path. See 4.1.2 Driver Environment Variables for information on the environment variable that specifies the search locations. Alternatively, use the full directory path along with the driver name when executing the compiler.

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

Command-line options are case sensitive, with their format and description being supplied in 4.6 Option Descriptions. Many of the command-line options accepted by `xc8-cc` are common to all the MPLAB XC compilers, to allow greater portability between devices and compilers.

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

Note also that the base name of some of the output files is based on the base name of the first C source file listed on the `xc8-cc`command line, unless the name is specified using the `-o` option.

If you are building code using a make system, familiarity with the unique intermediate p-code file format (described in 4.2.3 Multi-Step C Compilation), is recommended. Object files are seldom used with the MPLAB XC8 C Compiler, unless assembly source modules are used in the project.

4.1.1.1 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=16F877A -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.

4.1.2 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.

4.1.3 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 4-1. Input File Types

Extension	File format
`.c`	C source file
`.i`	Preprocessed C source file
`.p1`	P-code intermediate file
`.s`	Assembler source file
`.S`	Assembly source file requiring preprocessing
`.as` or `.asm`	Assembly source file
`.o`	Relocatable object code file
`.a`	Archive (library) file
`.hex`	Intel HEX file

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 name of the IDE project that contains them.

MPLAB XC8 will accept assembly source files with the `.as` or `.asm` extensions, but these extensions are not accepted by other XC compilers. It is recommended that you use the `.s` extension for assembly source files. Alternatively, you can use the `.S` extension or use the `-x` option with source files using a `.s` extension for assembly source files that must be preprocessed before being assembled.

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.

4.2 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.

4.2.1 The Compiler Applications

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

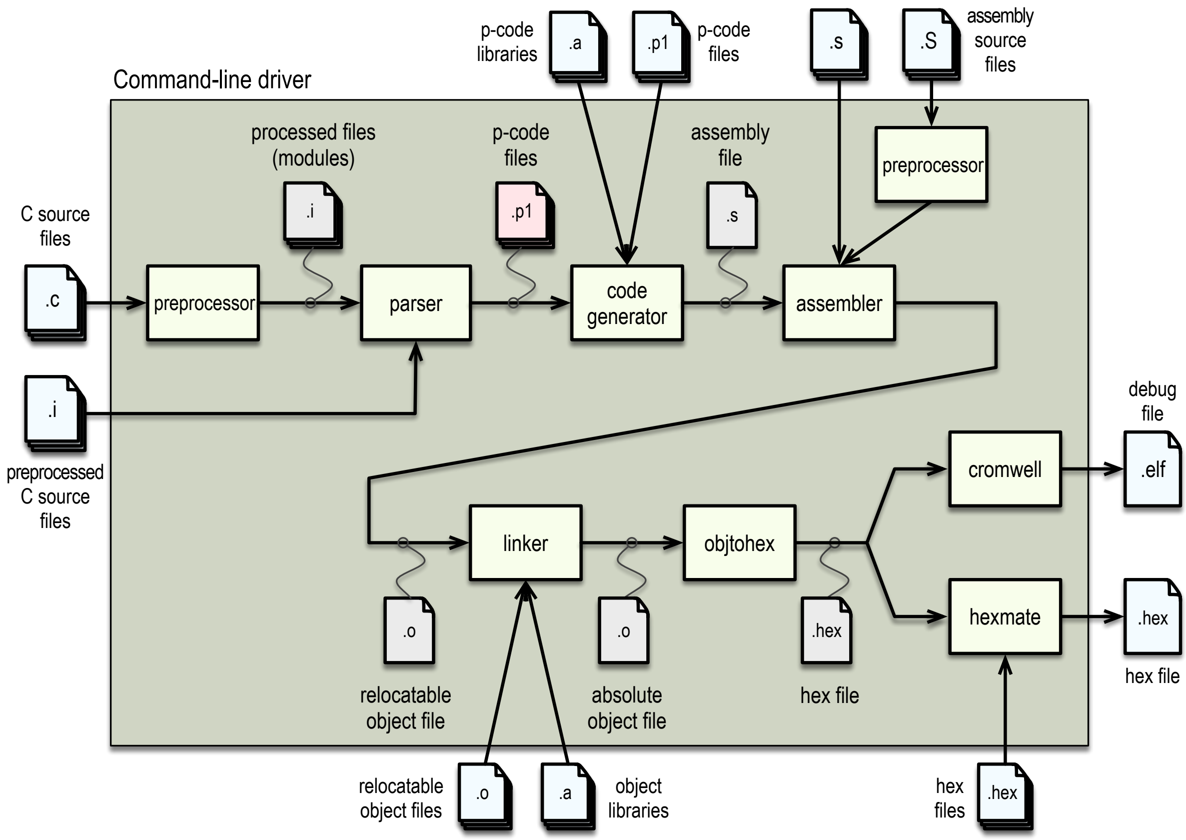
The large shaded box represents the compiler, which is controlled by the command line driver, `xc8-cc`. You might be satisfied just knowing that C source files (shown on the far left) are passed to the compiler and the resulting output files (shown here as a HEX and ELF debug file on the far right) are 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.

When compiling for the C99 standard, the Clang front end application performs the role of both the preprocessor and parser applications.

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 4-1. Compiler Applications And Files



It is recommended that only the `hexmate` and archiver (`xc8-ar`) internal applications be executed directly. Their command-line options are described in 8. Utilities.

4.2.2 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.

4.2.2.1 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=16F877A ex1.c
```

This command compiles the `ex1.c` source file for a 16F877A device and has the output written to `ex1.elf`, which may be used by your development environment.

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 4.2.3 Multi-Step C Compilation).

Unless otherwise specified, a HEX file and ELF file are produced as the final output.

The intermediate files remain after compilation has completed, but most other temporary files are deleted, unless you use the `-save-temps` option (see 4.6.5.5 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 4.6.2.3 O: Specify Output File).

4.2.2.2 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=16F877A 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.

4.2.3 Multi-Step C Compilation

A multi-step compilation method can be employed to build projects consisting of one or more C source files. Make utilities can use this feature, taking note of which source files have changed since the last build to speed up compilation. Incremental builds are also be performed by integrated development environments.

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

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

```
xc8-cc -mcpu=16F877A -c ex1.c
xc8-cc -mcpu=16F877A -c add.c
```

```
xc8-cc -mcpu=16F877A  ex1.p1 add.p1
```

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 resultant intermediate files are linked in the final step to create the final output `ex1.elf`. All the files that constitute the project must be present when performing the second stage of compilation.

Note: Always use p-code files (`.p1` extension) as the intermediate file format for C source.

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

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

Related Links

7. Linker

4.2.4 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=16F877A 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.

MPLAB XC8 builds assembly source files before C source files, so that information contained in the assembly code can be subsequently passed to the code generator, see 5.11.3 Interaction between Assembly and C Code.

The intermediate file format associated with assembly source files is an object file (`.o` extension). The `-c` option (see 4.6.2.1 C: Compile To Intermediate File) will halt compilation after the assembly step when building assembly source files, generating the object file.

4.3 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

4.3.1 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 9.1 Library Example Code. For more information on creating and using your own libraries, see 5.10 Libraries.

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

4.3.1.1 Location and Naming Convention

The compiler will search for standard libraries in the `pic/lib/c99` or `pic/lib/c90` directory, based on your language standard selection.

The standard libraries are named *`family-type-options`*`.a`, where the following apply.

- *`family`* can be `pic18` for PIC18 devices, or `pic` for all other 8-bit PIC devices
- *`type`* indicates the sort of library functionality provided and can be `stdlib` for the standard library functions, or `intlib` for intrinsic functions, etc.
- *`options`* indicates hyphen-separated names to indicate variants of the library to accommodate different compiler options or modes, e.g., `htc` for the default flavor of C used by MPLAB XC8, `d32` for 32-bit doubles, `sp` for space optimizations etc.

For example, the standard library for baseline and mid-range devices using 24-bit double types is `pic-stdlib-d24-sz.a`.

4.3.2 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 5.9 Main, Runtime Startup and Reset.

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

4.3.2.1 Runtime Startup Code Generation

Rather than the traditional method of linking in generic, precompiled runtime startup routines, the MPLAB XC8 C Compiler determines what runtime startup code is required from the user's program and then generates this code each time you build your project.

The default operation of the driver is to keep the startup module; however, you can ask that it be deleted by using the option `-mno-keep-startup` (see 4.6.1.13 Keep Startup Option). If you are using the MPLAB X IDE to build, the file will be deleted unless you indicate otherwise in the **Project Properties** dialog.

If the startup module is kept, it will be called `startup.s` and will be located in the current working directory. If you are using an IDE to perform the compilation, the destination directory will be dictated by the IDE. Assuming you have the **default** configuration selected, MPLAB X IDE stores this file in either the `dist/default/production` or `dist/default/debug` directories (based on whether you perform a production or debug build) in your project directory.

Generation of the runtime startup code is an automatic process that does not require any user interaction; however, some aspects of the runtime code can be controlled, if required, using the `-Wl,--no-data-init` option. 4.6.12 Mapped Linker Options describes the use of this option.

The runtime startup code is executed before `main()`. However, if you require any special initialization to be performed immediately after Reset, you should use the powerup feature described later in 5.9.3 The Powerup Routine.

4.4 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.

Note: Throughout this manual, the term project name will refer to either the name of the project created in the IDE, or the base name (file name without extension) of the first C source file specified on the command line.

4.4.1 Output Files

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

Table 4-2. Common Output Files

Extension	File Type	How created
`.elf`	ELF (Executable and Linkable Format) with Dwarf debugging information	`-o` option
`.cof`	COFF (Common Object File Format)	`-gcoff` option
`.p1`	P-code file (Intermediate file)	`-c`
`.s`	Assembly file	`-S` option
`.i`	Preprocessed C file	`-E` and `-o` option

The default behavior of `xc8-cc` is to produce an ELF and Intel HEX output. Unless changed by the `-o` option (see 4.6.2.3 O: Specify Output File), the base names of these files will be the project name.

The default output can be changed by using the `-gcoff` option, which generates a COFF file (described in 4.6.5.2 G: Produce Debugging Information Option).

The ELF/DWARF file is used by debuggers to obtain debugging information about the project and allows for more accurate debugging compared to 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.p1` when the `-c` option is used.

4.4.2 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 4-3. 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.

There is one list file produced for the entire program, including library code.

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.

4.5 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.

Advisory Messages Conveys information regarding a situation the compiler has encountered or some action the compiler is about to take. The information is being displayed “for your interest” and typically requires no action to be taken. Compilation will continue as normal after such a message is issued.

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.

A list of warning and error messages, and descriptions can be found in this guide.

Related Links

10. Error and Warning Messages

4.5.1 Messaging Overview

A centralized messaging system is used by most applications to report on the validity of your program and the compilation process.

Note: The Clang front end, used when compiling for C99 projects, currently bypasses this system and is not controlled by the features described here. The front end is responsible for messages such as syntax errors in your C source code; however, all other applications always use the messaging system described here.

Messages are referenced by a unique number. The messaging system takes the message number requested by the application that needs to convey the information and determines the corresponding message type and string from one of several Message Description Files (MDF), stored in the `pic/dat` directory of the compiler’s installation directory. Some applications also include a built-in copy of this information should a MDF not be specified on the command line or the file cannot be found.

The user is able to set a threshold for being notified of message importance, so that only messages the user considers significant are displayed. In addition, messages with a particular number can be disabled. In some instances, a pragma can also be used to disable a particular message number within specific lines of code. These methods are explained in 4.5.3.1 Disabling Messages.

As well as the actual message string produced by the compiler, there are several other pieces of information that can be displayed when the message is triggered, such as the message number, the line number of the code that triggered the message, the name of the file that contains the offending code, and the application that issued the message, etc.

If a message is an error, an internal counter is incremented. After a specific number of errors has been reached, compilation of the current module will cease. The default number of errors that cause this termination can be adjusted by using the `-fmax-errors` option (see 4.6.4.1 Max Errors Option). This counter is reset for each internal compiler application, thus specifying a maximum of five errors will allow up to five errors from the parser, five from the code generator, five from the linker, five from the driver, etc.

Although the information in the MDF can be modified with any text editor, this is not recommended. Message behavior should only be altered using the options and pragmas described in the following sections.

4.5.2 Message Format

By default, messages are printed in a human-readable format. This format can vary from one internal application to another, since each application reports information about different file formats.

Some applications (for example, the parser) are typically able to pinpoint the area of interest down to a position on a particular line of C source code. Other applications (for example, the linker) can at best only indicate a module name and record number, which is less directly associated with any particular line of code. Some messages relate to issues in driver options that are in no way associated with the source code.

The format of messages produced by the Clang front end cannot be changed, but the following information is still relevant for the other compiler applications.

The compiler will use environment variables, if set, whose values are used as a template for all messages produced by all compiler applications. The names and effects of these environment variables are given in the table below.

Table 4-4. Messaging Environment Variables

Variable	Effect
`HTC_MSG_FORMAT`	All advisory messages
`HTC_WARN_FORMAT`	All warning messages
`HTC_ERR_FORMAT`	All error and fatal error messages

The value of these environment variables are strings that are used as templates for the message format. Printf-like placeholders can be placed within the string to allow the message format to be customized. The placeholders and what they represent are presented in the table below.

Table 4-5. Messaging Placeholders

Placeholder	Replacement
`%a`	Application name
`%c`	Column number
`%f`	Filename
`%l`	Line number
`%n`	Message number
`%s`	Message string (from MDF)

If these options are used in a DOS batch file, two percent characters will need to be used to specify the placeholders, as DOS interprets a single percent character as an argument and will not pass this on to the compiler. For example:

```
SET HTC_ERR_FORMAT="file %%f: line %%l"
```

4.5.3 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.

4.5.3.1 Disabling Messages

Each numbered warning message has a default number indicating a level of importance. This number is specified in the MDF and ranges from -9 to 9. The higher the number, the more important the warning.

Warning messages can be disabled by adjusting the warning level threshold using the `-mwarn` driver option, (see 4.6.4.2 Warn Option). Any warnings whose level is below that of the current threshold are not displayed.

The default threshold is 0 which implies that only warnings with a warning level of 0 or higher will be displayed by default. The information in this option is propagated to all compiler applications, so its effect will be observed during all stages of the compilation process.

When building for C99, the Clang front end is used. This application does not use the numbered message system. Warning messages coming from the parser indicate the option which allowed their generation, and this information can be used to disable the message. For example, if the following warning was issued:

```
init.c:8:11: warning: implicit conversion changes signedness: 'int' to 'unsigned int' [-Wsign-conversion]
```

the warning could be disabled by using the option `-Xparser -Wno-sign-conversion`. This uses the "no-" form of the option indicated in the square brackets in the warning message. The `-Xparser` driver option passes the option argument to the parser (Clang) application.

All warnings from all applications can be disabled using the `-w` option.

Note: Disabling error or warning messages in no way fixes the condition that triggered the message. Always use extreme caution when disabling warning messages.

4.5.3.2 Changing Message Types

It is also possible to change the type of some messages, for example changing what would normally be a warning to an error. This can only be done for messages generated by the parser or code generator. See 5.13.3.11 The #pragma Warning Directive for more information on this pragma.

4.6 Option Descriptions

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

Many options follow a GCC style; however, the compiler is not based on GCC and you cannot use any other GCC options that are not documented here. Many of the old-style `xc8` driver options (described in the MPLAB® XC8 C Compiler User's Guide DS50002053) can be used with `xc8-cc` if there is no GCC-style option equivalent available; however, their use may trigger a warning.

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.

4.6.1 Options Specific to PIC Devices

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

Table 4-6. Machine-specific Options

Option	Controls
`-maddrqual=`*action*	How the compiler will respond to storage qualifiers
`-mchecksum=`*specs*	The generation and placement of a checksum or hash
`-mcodeoffset=`*offset*	The offset applied to reset and interrupt vectors
`-m[no-]config`	Whether the device will be programmed with default configuration bit values
`-mcpu=`*device*	The target device that code will be built for
`-mdebugger=`*type*	Which debugger will be in use when executing the code
`-m[no-]default-config-bits`	See `-m[no-]config`
`-mdfp=`*path*	Which device family pack to use
`-m[no-]download`	How the final HEX file is conditioned

...........continued	
Option	**Controls**
`-m[no-]download-hex`	See `-m[no-]download`
`-memi=`*`mode`*	The external memory interface that will be used
`-merrata=`*`type`*	Which workarounds to errata will be applied by the compiler
`-mheap=`*`size`*	The maximum allowable size for the heap
`-m[no-]ivt=`*`address`*	The interrupt vector table selected at startup
`-m[no-]keep-startup`	Whether the runtime startup source is deleted after compilation
`-mmaxichip`	Use of a hypothetical device with full memory
`-m[no-]osccal`	Whether the oscillator will be calibrated
`-m[no-]oscval=`*`value`*	The oscillator calibration value
`-mram=`*`ranges`*	Data memory that is available for the program
`-mreserve=`*`ranges`*	What memory should be reserved
`-m[no-]resetbits`	Whether the device status bits should be preserved
`-mrom=`*`ranges`*	Program memory that is available for the program
`-m[no-]save-resetbits`	See `-m[no-]resetbits`
`-mshroud`	Whether the output file should obfuscate the source code
`-mstack=`*`model`*`[:`*`size`*`]`	Which data stack will be used by default
`-m[no-]stackcall`	Whether functions can be called via lookup tables
`-msummary=`*`types`*	What memory summary information is produced
`-mundefints=`*`action`*	How the compiler completes unimplemented interrupts
`-m[no-]use-ivt`	see `-m[no-]ivt`

4.6.1.1 Addrqual Option

The`-maddrqual=`*`action`* option indicates the compiler's response to non-standard memory qualifiers in C source code, as shown in the table below.

Table 4-7. Compiler Response To Memory Qualifiers

Action	Response
`require`	The qualifiers will be honored. If they cannot be met, an error will be issued.
`request`	The qualifiers will be honored, if possible. No error will be generated if they cannot be followed.
`ignore`	The qualifiers will be ignored and code compiled as if they were not used.
`reject`	If the qualifiers are encountered, an error will be immediately generated.

The `__near` qualifier is affected by this option. On PIC18 devices, this option affects the `__far` qualifier; and for other 8-bit devices, the `__bank(`*`x`*`)` qualifier. By default, these qualifiers are ignored; i.e., they are accepted without error, but have no effect. Using this option allows these qualifiers to be interpreted differently by the compiler.

For example, when using the option `-maddrqual=request`, the compiler will try to honor any non-standard qualifiers, but silently ignore them if they cannot be met.

4.6.1.2 Checksum Option

The `-mchecksum=specs` option will calculate a hash value (for example checksum or CRC) over the address range specified and stores the result in the hex file at the indicated destination address. The general form of this option is as follows.

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

The *start*, *end* and *destination* attributes are, by default, hexadecimal constants. The addresses defining the input range (`start` - `end`) are typically made multiples of the algorithm width. If this is not the case, bytes (with value 0) will pad any missing input word locations. The `destination` is where the hash result will be stored. This address cannot be within the range of addresses over which the hash is calculated.

The following specifications are appended as a comma-separated list to this option.

Table 4-8. Checksum Arguments

Argument	Description
`width=n`	Optionally specifies the decimal width of the result. Results can be calculated for byte-widths of 1 to 4 bytes for most algorithms, but it represents the bit width for SHA algorithms. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order. This width argument is not required with any Fletcher algorithm, as they have fixed widths, but it may be used to alter the default endianism of the result.
`offset=nnnn`	Specifies an initial hexadecimal value or offset to be used in the hash calculation. It is not used with SHA algorithms.
`algorithm=n`	The decimal argument selects one of the hash algorithms implemented in Hexmate. The selectable algorithms are described in Table 4-9. If unspecified, the default algorithm used is 8-bit checksum addition (algorithm 1).
`polynomial=nn`	Selects the polynomial value when using CRC algorithms
`code=nn.Base`	Selects a hexadecimal code that will trail each byte in the result. This can allow each byte of the hash result to be embedded within an instruction, for example `code=34` will embed each byte of the result in a `retlw` instruction on Mid-range devices, or `code=0000` will append two 0x00 bytes to each byte of the hash. This code sequence can be optionally followed by `.Base`, where *Base* is the number of bytes of the hash to be output before the trailing code sequence is appended. A specification of `code=1122.2`, for example, will output the bytes 0x11 and 0x22 after each two bytes of the hash result.
`revword=n`	Specifies a decimal word width. If this is non-zero, then bytes within each word are read in reverse order when calculating a hash value. Words are aligned to the addresses in the HEX file. At present, the width must be 0 or 2. A zero width disables the reverse-byte feature, as if the `revword` suboption was not present. This suboption should be used when using Hexmate to match a CRC produced by a PIC hardware CRC module that use the Scanner module to stream data to it.
`skip=n.Bytes`	Specifies a decimal word width. If this is non-zero, then the MSB within each word is skipped for the purposes of calculating a hash value. Words are aligned to the addresses in the HEX file. At present, the width must be 0 (which disables the skip feature, as if the `skip` suboption was not present) or greater than 1. This value can be optionally followed by `.Bytes`, where *Bytes* is a number representing the number of bytes to skip in each word, for example `skip=4.2` will skip the two most significant bytes in each 4-byte word.

Note that the reverse and skip features act on words that are aligned to the HEX file addresses, not to the position of the data in the sequence being processed. In other words, the alignment of the words are not affected by the `start` and `end` addresses specified with the option.

If an accompanying `--fill` option (4.6.11.10 Fill Option) has not been specified, unused locations within the specified address range will be automatically filled with 0xFFF for baseline devices, 0x3FFF for mid-range devices, or 0xFFFF for PIC18 devices. This is to remove any unknown values from the calculations and ensure the accuracy of the result.

For example:

```
-mchecksum=800-fff@20,width=1,algorithm=2
```

will calculate a 1-byte checksum from address 0x800 to 0xfff and store this at address 0x20. A 16-bit addition algorithm will be used. Table 4-9 shows the available algorithms and 8.2.3 Hash Value Calculations describes these in detail.

Table 4-9. Checksum Algorithm Selection

Selector	Algorithm description
`-5`	Reflected cyclic redundancy check (CRC)
`-4`	Subtraction of 32 bit values from initial value
`-3`	Subtraction of 24 bit values from initial value
`-2`	Subtraction of 16 bit values from initial value
`-1`	Subtraction of 8 bit values from initial value
`1`	Addition of 8 bit values from initial value
`2`	Addition of 16 bit values from initial value
`3`	Addition of 24 bit values from initial value
`4`	Addition of 32 bit values from initial value
`5`	Cyclic redundancy check (CRC)
`7`	Fletcher’s checksum (8 bit calculation, 2-byte result width)
`8`	Fletcher’s checksum (16 bit calculation, 4-byte result width)
`10`	SHA-2 (currently only SHA256 is supported)

The hash calculations are performed by the Hexmate application. The information in this driver option is passed to the Hexmate application when it is executed.

4.6.1.3 Codeoffset Option

The `-mcodeoffset=`*`offset`* option shifts the reset and interrupt vector locations up in memory, by the specified offset, preventing code and data from using memory up to this offset address. This operation is commonly required when writing bootloaders.

The address is assumed to be a hexadecimal constant. A leading `0x`, or a trailing `h` hexadecimal specifier can be used but is not necessary.

This option does not affect the location of the interrupt vector table for those devices that support vectored interrupts, nor does it affect the low- or high-priority interrupt vectors for the same devices operating in legacy mode. Adjust the IVTBASE register to perform either of these two tasks.

As an example, the option `-mcodeoffset=600` will move the reset vector from address 0 to 0x600; and move the interrupt vector from address 4 to 0x604, in the case of mid-range PIC devices, or to the addresses 0x608 and 0x618 for PIC18 devices. No code or data will be placed at the addresses 0 thru 0x5FF.

As the reset and interrupt vector locations are at fixed addresses in the PIC device, it is the programmer’s responsibility to provide code that can redirect control from the actual vector locations to the reset and interrupt routines in their offset location.

4.6.1.4 Config Option

The `-mconfig` option can be used to have the compiler program default values for those configuration bits that have not been specified in the code using the `config` pragma. The alternate form of this option is `-mdefault-config-bits`.

The default operation is to not program unspecified bits, and this can be made explicit using the `-mno-config` option (or its equivalent, `-mno-default-config-bits`).

4.6.1.5 Cpu Option

The `-mcpu=`*`device`* option must be used to specify the target device when building. This is the only option that is mandatory.

For example `-mcpu=18f6722` will select the PIC18F6722 device. To see a list of supported devices that can be used with this option, use the `-mprint-devices` option (4.6.2.8 Print-devices).

4.6.1.6 Debugger Option

The `-mdebugger=`*`type`* option is intended for use in compatibility with development tools that can act as a debugger. `xc8-cc` supports several debuggers and these are defined in the table below. Failing to select the appropriate debugger can lead to runtime failure.

Table 4-10. Selectable Debuggers

Type	Debugger selected
`none`	No debugger (default)
`icd2`	MPLAB® ICD 2
`icd3`	MPLAB ICD 3
`icd4`	MPLAB ICD 4
`pickit2`	PICkit™ 2
`pickit3`	PICkit 3
`pickit4`	PICkit 4
`realice`	MPLAB REAL ICE™ In-circuit Emulator
`snap`	MPLAB Snap

For example:

```
xc8-cc -mcpu=16F877AA -mdebugger=icd4 main.c
```

will ensure that none of the source code is allocated resourced that would be used by the debug executive for an MPLAB ICD 4.

When a debugger is set using this option, a preprocessor macro is defined, allowing your source code to change based on the debugger selection, see 5.13.2 Predefined Macros.

If the debugging features of the development tool are not to be used (for example, if the MPLAB ICD 4 is only being used as a programmer), then the debugger option can be set to `none`, because memory resources will not be used by the tool in that situation.

4.6.1.7 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 use the device-specific files provided in 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.

When using this option, the preprocessor will search for include files in the `<DFP>/xc8/pic/include/proc` and `<DFP>/xc8/pic/include` directories first, then search the standard search directories.

4.6.1.8 Download Option

The `-mdownload` option conditions the Intel HEX for use by bootloader. The `-mdownload-hex` option is equivalent in effect.

When used, this option will pad data records in the Intel HEX file to 16-byte lengths and will align them on 16-byte boundaries.

The default operation is to not modify the HEX file and this can be made explicit using the option `-mno-download`. (`-mno-download-hex`)

4.6.1.9 Emi Option

The `-memi=`*`mode`* option allows you to select the mode used by the external memory interface available on some PIC18 devices.

The interface can operate in a 16-bit word-write or byte-select mode; or in an 8-bit byte-write mode, and which are represented in the table below.

Table 4-11. External Memory Interface Modes

Mode	Operation
`wordwrite`	16-bit word-write mode (default)
`byteselect`	16-bit byte-select mode
`bytewrite`	8-bit byte-write mode

For example, the option `-memi=bytewrite` will select the 8-bit byte-write mode.

The selected mode will affect the code generated when writing to the external data interface. In word write mode, dummy reads and writes can be added to ensure that an even number of bytes are always written. In byte-select or byte-write modes, dummy reads and writes are not generated and can result in more efficient code.

Note that this option does not pre-configure the device for the selected mode. Your device data sheet will indicate the settings required in your code.

4.6.1.10 Errata Option

The `-merrata=`*`type`* option allows specification of software workarounds to documented silicon errata issues. A default set of errata apply to each device, but this set can be adjusted by using this option and the arguments presented in the table below.

Table 4-12. Errata Workarounds

Type	#	Avoided Issue
`4000`	0	Program memory accesses/jumps across 4000h address boundary
`fastints`	1	Fast interrupt shadow registers corruption
`lfsr`	2	Broken LFSR instruction
`minus40`	3	Program memory reads at -40 degrees
`reset`	4	`goto` instruction cannot exist at Reset vector

...........continued

Type	#	Avoided Issue
`bsr15`	5	Flag problems when BSR holds value 15
`daw`	6	Broken DAW instruction
`eedatard`	7	Read EEDAT in immediate instruction after RD set
`eeadr`	8	Don't set RD bit immediately after loading EEADR
`ee_lvd`	9	LVD must stabilize before writing EEPROM
`fl_lvd`	10	LVD must stabilize before writing Flash
`tblwtint`	11	Clear interrupt registers before `tblwt` instruction
`fw4000`	12	Flash write exe must act on opposite side of 4000h boundary
`resetram`	13	RAM contents can corrupt if async. Reset occurs during write access
`fetch`	14	Corruptible instruction fetch. – applies FFFFh (`nop`) at required locations
`clocksw`	15	Code corruption if switching to external oscillator clock source – applies switch to HFINTOSC high-power mode first
`branch`	16	The PC might become invalid when restoring from an interrupt during a `bra` or `brw` instruction — avoids branch instructions
`brknop2`	17	Hardware breakpoints might be affected by `bra` instruction — avoids branching to the following location
`nvmreg`	18	The program will access data flash rather than program flash memory after a reset — adjusts the NVMCON register
`bsr63`	19	Corrupted execution of `movff` instruction when the BSR holds 63

At present, workarounds are mainly employed for PIC18 devices, but the `clocksw` and `branch` errata are only applicable for some enhanced Mid-range devices.

To disable all software workarounds, use the following.

```
-merrata=none
```

To maintain all the default workarounds but disable the jump across 4000 errata, for example, use the following:

```
-merrata=default,-4000
```

The value assigned to the preprocessor macro `_ERRATA_TYPES` (see 5.13.2 Predefined Macros) indicates the errata applied. Each errata listed in Table 4-12 table represents one bit position in the macro's value, with the topmost errata in the table being the least significant. That bit position in the `_ERRATA_TYPES` macro is set if the corresponding errata is applied. The header file `<errata.h>` contains definitions for each errata value, for example `ERRATA_4000` and `ERRATA_FETCH`, which can be used with the compiler-defined `_ERRATA_TYPES` macro to determine which erratas are in effect.

4.6.1.11 Heap Option

The `-mheap=[`*`size`*`|auto]` option specifies the size of memory reserved for the heap.

The heap is used with the standard dynamic memory allocation functions, those being `malloc()`, `calloc()`, `realloc()`, and `free()`, whose operation is described in 5.4.7.1 Dynamic Memory Allocation for PIC Devices.

When this option is used with a *`size`* argument, being a decimal number of bytes, the compiler will attempt to reserve the requested amount of memory for the heap. Thus, at runtime, the heap will be able to grow to the specified size with no corruption of data. This memory is reserved unconditionally, even if no dynamic memory allocation functions are used in the program.

Alternatively, the argument may be specified as `auto`, in which case the compiler will calculate a maximum size for the heap after other static and stack allocations have been made. Memory with this calculated size will only be reserved, however, if the compiler detects calls to any of the dynamic memory allocation functions in the program. If no `-mheap` option is specified, then the compiler uses an auto setting, as if `-mheap=auto` was issued. Whenever acting with an auto setting, the compiler will issue a compiler advisory message if dynamic memory allocation functions have been detected and memory has been reserved as a result.

The size of software stacks used by functions compiled to use a reentrant model are also dynamic in nature, and memory must be reserved for their use. That memory is specified using the `-mstack` option (see 4.6.1.22 Stack Option). A size argument of `auto` can be used with both the `-mheap` and `-mstack` options, in which case the compiler will evenly distribute the free memory remaining between the heap and stacks after other statical allocations have been made.

4.6.1.12 Ivt Option

The `-mivt=`*`address`* option selects the interrupt vector table that will be used at the beginning of program execution for those PIC18 devices that implement interrupt vector tables.

The address argument specified is written to the IVTBASE register during startup, for example, `-mivt=0x200` will select the interrupt vector table whose base address is at 200h. The table at the address you specify must be populated by vectors in your source code. This is achieved using the `base` argument in the interrupt routine definitions you write (see 5.8.1 Writing an Interrupt Service Routine). The default operation is to leave the vector table at address 0x8 and this can be made explicit using the option `-mno-ivt`.

4.6.1.13 Keep Startup Option

The `-mkeep-startup` option prevents the deletion of the startup assembly module after compilation. By default, this file is not deleted; the `-mno-keep-startup` option can be used to delete the file after each build.

4.6.1.14 Maxichip Option

The `-mmaxichip` option tells the compiler to build for a hypothetical device with the same physical core and peripherals as the selected device, but with the maximum allowable memory resources permitted by the device family. You might use this option if your program does not fit in your intended target device and you wish to get an indication of the code or data size reductions needed to be able to program that device.

The compiler will normally terminate if the selected device runs out of program memory, data memory, or EEPROM. When using this option, the program memory of PIC18 and mid-range devices will be maximized to extend from address 0 to either the bottom of external memory or the maximum address permitted by the PC register, whichever is lower. The program memory of baseline parts is maximized from address 0 to the lower address of the Configuration Words.

The number of data memory banks is expanded to the maximum number of selectable banks as defined by the BSR register (for PIC18 devices), RP bits in the STATUS register (for mid-range devices), or the bank select bits in the FSR register (for baseline devices). The amount of RAM in each additional bank is equal to the size of the largest contiguous memory area within the physically implemented banks.

If present on the device, EEPROM is maximized to a size dictated by the number of bits in the EEADR or NVMADR register, as appropriate.

If required, check the map file (see 7.3 Map Files) to see the size and arrangement of the memory available when using this option with your device.

Note: When using the `-mmaxichip` option, you are not building for a real device. The generated code may not load or execute in simulators or the selected device. This option will not allow you to fit extra code into a device.

4.6.1.15 Osccal Option

The `-mosccal` option can be used to calibrate the oscillator for some PIC10/12/16 devices.

When using this option, the compiler will generate code which will calibrate the oscillator using the calibration constant preprogrammed in the device. The option `-mno-osccal` will omit the code that performs this initialization from the runtime startup code.

4.6.1.16 Oscval Option

The `-moscval=value` option allows you to specify the hexadecimal value that will be used to calibrate the oscillator for some PIC10/12/16 devices. The calibration value is usually preprogrammed into your device; however this option allows you to specify an alternate value, or the original value if it has been erased from the device.

The calibration value is stored at the top of program memory, encapsulated into a `movlw` instruction. The runtime startup code executes this instruction and stores the WREG into the calibration register. The option `-moscval=55` would ensure that the value 0x55 is loaded to the oscillator calibration register at startup (see 5.2.11 Oscillator Calibration Constants). Using the `-mno-oscval` form of this option (even with an argument), or not specifying any `-moscval` option, will result in the calibration register being programmed with whatever value is currently stored in the device.

4.6.1.17 Ram Option

The `-mram=ranges` option is used to adjust the data memory that is specified for the target device. Without this option, all the on-chip RAM implemented by the device is available, thus this option only needs be used if there are special memory requirements. Specifying additional memory that is not in the target device might result in a successful compilation, but can lead to code failures at runtime.

For example, to specify an additional range of memory to that already present on-chip, use:

```
-mram=default,+100-1ff
```

This will add the range from 100h to 1ffh to the on-chip memory. To only use an external range and ignore any on-chip memory, use:

```
-mram=0-ff
```

This option can also be used to reserve memory ranges already defined as on-chip memory in the relevant chipinfo file. To do this, supply a range prefixed with a minus character, `-`, for example:

```
-mram=default,-100-103
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 103h for allocation of RAM objects.

This option will adjust the memory ranges used by linker classes (see 7.1.1 A: Define Linker Class) . Any objects contained in a psect that do not use the classes affected by this option might be linked outside the valid memory specified by this option.

This option is also used to specify RAM for `far` objects on PIC18 devices. These objects are stored in the PIC18 extended memory. Any additional memory specified with this option whose address is above the on-chip program memory is assumed to be extended memory implemented as RAM.

For example, to indicate that RAM has been implemented in the extended memory space at addresses 0x20000 to 0x20fff, use the following option.

```
-mram=default,+20000-20fff
```

4.6.1.18 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.

This option performs a similar task to the `-mram` and `-mrom` options, but it cannot be used to add additional memory to that available for the program.

4.6.1.19 Resetbits Option

The `-mresetbits` option allows you to have the content of the status register preserved by the runtime startup code for PIC10/12/16 devices (described in 5.9.2.4 Status Register Preservation). The `-m[no-]save-resetbits` option is equivalent in effect.

When this option is in effect, the saved registers can be accessed in your program. The compiler can detect references to the saved STATUS register symbols an will automatically enable this option.

4.6.1.20 Rom Option

The `-mrom=`*`ranges`* option is used to change the default program memory that is specified for the target device. Without this option, all the on-chip program memory implemented by the device is available, thus this option only needs be used if there are special memory requirements. Specifying additional memory that is not in the target device might result in a successful compilation, but can lead to code failures at runtime.

For example, to specify an additional range of memory to that on-chip, use:

```
-mrom=default,+100-2ff
```

This will add the range from 100h to 2ffh to the on-chip memory. To only use an external range and ignore any on-chip memory, use:

```
-mrom=100-2ff
```

This option can also be used to reserve memory ranges already defined as on-chip memory in the chip configuration file. To do this supply a range prefixed with a minus character, `-`, for example:

```
-mrom=default,-100-1ff
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 1ffh for allocation of ROM objects.

This option will adjust the memory ranges used by linker classes (see 7.1.1 A: Define Linker Class) . Any code or objects contained in a psect that do not use the classes affected by this option might be linked outside the valid memory specified by this option.

Note that some psects must be linked above a threshold address, most notably some psects that hold `const`-qualified data. Using this option to remove the upper memory ranges can make it impossible to place these psects.

4.6.1.21 Shroud Option

The `-mshroud` option should be used in situations where either intermediate or library files are built from confidential source code and are to be distributed.

When using this option, C comments, which are normally included into these files, as well as line numbers and variable names will be removed, or obfuscated, so that the original source code cannot be reconstructed from the distributed files.

4.6.1.22 Stack Option

The `-mstack=`*`model`*`[:`*`size`*`|auto]` option selects the stack model to be used by a program, which will dictate where the stack-based variables for functions that do not use a stack specifier will be allocated memory. Additionally, it can specify the size of memory reserved for the stacks.

Functions can use stack specifiers (described in 5.7.1.3 Reentrant And Nonreentrant Specifiers) to indicate which stack should be used for that function's stack-based variables. For functions that do not use these specifiers, the `-mstack` option controls the stack type that the compiler will use.

The two data stacks available are called a compiled stack and a software stack (described in 5.2.4.2 Data Stacks). The stack models that can be used with this option are described in the table below.

Table 4-13. Stack Suboptions

Model	Default Allocation for Stack-based Variables
`compiled` or `nonreentrant`	Use the compiled stack for all functions not using a stack specifier; these functions will then be non-reentrant.
`software` or `reentrant`	Use the software stack for functions not using a stack specifier; these functions will then be reentrant.
`hybrid`	Use the compiled stack for functions not using a stack specifier and that are not called reentrantly; use the software stack for all other functions not using a stack specifier; functions are only reentrant if required.

If the `-mstack` option is not used, all functions not using a stack specifier will use the compiled stack and be non-reentrant.

The `software` (or `reentrant`) or `hybrid` models have no effect on projects targeting baseline and mid-range devices, as these devices only support a compiled stack. In addition, all interrupt functions for any device must use the compiled stack, but functions they call may use the software stack.

The hybrid model (`-mstack=hybrid`) will let the compiler choose how to encode each function based on how it is called. A function is compiled to use the software stack if it is called reentrantly in the program; otherwise, it will use a compiled stack. This model allows for reentrancy, when required, but takes advantage of the efficiency of the compiled stack for the majority of the program's functions.

Note: Use the `software` (`reentrant`) setting with caution. The maximum runtime size of the software stack is not accurately known at compile time, so the compiler cannot predict an overflow, which could corrupt objects or registers. When all functions are forced to use the software stack, the stack size might increase substantially. Code to access objects on a software stack is typically larger than that used to access objects on a compiler stack.

In addition to the stack model, this option can be used to specify the maximum size of memory reserved by the compiler for the software stack. This option configuration only affects the software stack; there are no controls for the size of the compiled stack, which is fixed and known at compile time.

Distinct software stacks are created for use by main-line code and each interrupt function, but this is transparent at the program level. The compiler automatically manages the allocation of memory to each stack. If your program does not define any interrupt functions, all the available memory can potentially be made available to the software stack used by main-line code.

The maximum space the compiler will attempt to reserve for each area of the stack can be specified by following the `reentrant` stack type with a colon-separated list of decimal values, each being the size, in bytes, of the memory to be reserved. The sizes specified in the option correspond in this order to: the main-line code, the lowest priority interrupt through to the highest priority interrupt. (PIC18 devices have two separate interrupts; other devices which support interrupts have one.) For example the option:

```
-mstack=reentrant:200:0:40
```

when used with PIC18 projects will attempt to reserve 200 bytes for the main-line stack, zero bytes for the low-priority interrupt stack, and 40 bytes for the high-priority interrupt stack. At runtime, the stacks will be able to grow to the specified sizes with no corruption of data. This memory is reserved unconditionally, even if no reentrant functions are present in the program.

Alternatively, any size argument may be specified as `auto`, in which case the compiler will calculate a maximum size for this stack after other static, stack, and heap allocations have been made. Memory with this calculated size will only be reserved, however, if the compiler detects the presence of any function using a reentrant stack. If no `-mstack` option is specified, then the compiler uses an auto setting for every stack that can be potentially used by the program. Whenever acting with an auto stack size setting, the compiler will issue a compiler advisory message if reentrant functions have been detected and memory has been reserved as a result. For PIC18 devices, the following example:

```
-mstack=reentrant:auto:30:50
```

will arrange the stack starting locations so that the low-priority interrupt stack can grow to at most 30 bytes (before overflow); the high-priority interrupt stack can grow to at most 50 bytes (before overflow); and the main-line code stack can consume the remainder of the free memory after other static, stack, and heap allocations have been made. If you are not using an interrupt, it is recommended that you explicitly set the unused interrupt stack size to zero using this option.

If the maximum stack sizes are specified with this option, each size must be a numerical value or the `auto` token; do not leave a size field empty. If the sizes specified in this option would allocate more stack memory than is available, a warning will be issued and only the available memory will be utilized.

The size of the heap used by dynamic memory allocation functions is also dynamic in nature, and memory must be reserved for its use. That memory is specified using the `-mheap` option (see 4.6.1.11 Heap Option). A size argument of `auto` can be used with both the `-mstack` and `-mheap` options, in which case the compiler will evenly distribute the free memory remaining between the heap and stacks after statically allocated objects have been placed.

4.6.1.23 Stackcall Option

The `-mstackcall` option allows the compiler to use a table look-up method of calling functions.

Once the hardware function return address stack (5.2.4.1 Function Return Address Stack) has been filled, no further function nesting can take place without corrupting function return values. If this option is enabled, the compiler will revert to using a look-up table method of calling functions once the stack is full (see 5.7.7 Calling Functions).

4.6.1.24 Summary Option

The `-msummary=`*`type`* option selects the type of information that is included in the summary that is displayed once the build is complete. By default, or if the `mem` type is selected, a memory summary with the total memory usage for all memory spaces is shown.

A psect summary can be shown by enabling the `psect` type. This shows individual psects after they have been grouped by the linker and the memory ranges they cover. Table 4-20 shows what summary types are available. The default output printed corresponds to the `mem` setting.

SHA hashes for the generated hex file can also be shown using this option. These can be used to quickly determine if anything in the hex file has changed from a previous build.

Table 4-14. Summary Types

Type	Shows
`psect`	A summary of psect names and the addresses where they were linked will be shown.
`mem`	A concise summary of memory used will be shown (default).
`class`	A summary of all classes in each memory space will be shown.
`hex`	A summary of addresses and HEX files that make up the final output file will be shown.
`file`	Summary information will be shown on screen and saved to a file.
`sha1`	A SHA1 hash for the hex file.
`sha256`	A SHA256 hash for the hex file.
`xml`	Summary information will be shown on the screen, and usage information for the main memory spaces will be saved in an XML file.
`xmlfull`	Summary information will be shown on the screen, and usage information for all memory spaces will be saved in an XML file.

If specified, the XML files contain information about memory spaces on the selected device, consisting of the space's name, addressable unit, size, amount used and amount free.

4.6.1.25 Undefints Option

The `-mundefints=`*`action`* option allows you to control how the compiler responds to uninitialized interrupt vectors, including undefined legacy low- and high-priority vectors, and entries in the interrupt vector table.

A warning is generated by the compiler if any uninitialized vectors are detected. This warning can be disabled by using the `ignore` action with this option.

The full list of possible actions is shown in the table below for target devices that are using the Vectored Interrupt Controller (VIC) module and for all other situations (which includes devices that do not have the VIC, or those that do but where the vector tables are disabled and the device is running in legacy mode).

For example, to have a software breakpoint executed by any vector location that is not linked to an interrupt function, use the option `-mundefints:swbp`.

The default action for projects using the VIC is to program the address of a reset instruction (which will be located immediately after the vector table) into each unassigned vector location; for all other devices, it is to leave the locations unprogrammed and available for other use.

If the target device does not implement a reset instruction or software breakpoint instruction and the `reset` or `swpb` action has been specified with this option, an instruction that can jump to itself will be programmed instead.

Table 4-15. Unused Interrupt Suboptions

Action	Devices using the VIC	All other devices
`ignore`	No action; vector location available for program code.	No action; vector location available for program code (default).
`reset`	Program each unassigned vector with the address of a reset instruction (default).	Program a reset instruction at each unassigned vector.
`swbp`	Program each unassigned vector with the address of a software breakpoint instruction.	Program a software breakpoint instruction at each unassigned vector.

An interrupt function can be assigned to any otherwise unassigned vector location by using the default interrupt source when defining that function (see 5.8.1 Writing an Interrupt Service Routine).

The `-mundefints` option is ignored if the target device does not support interrupts.

4.6.2 Options for Controlling the Kind of Output

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

Table 4-16. Kind-of-output Control Options

Option	Produces
`-c`	An intermediate file
`-E`	A preprocessed file
`-o` *`file`*	An output file with the specified name
`-S`	An assembly file
`-v`	Verbose compilation
`-xassembler-with-cpp`	Output after preprocessing all source files
`--help`	Help information only
`-mprint-devices`	Chip information only
`--version`	Compiler version information

4.6.2.1 C: Compile To Intermediate File

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

In the case of C source files, compilation will halt after the parsing stage, leaving behind p-code files with a `.p1` extension. For assembly 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.

4.6.2.2 E: Preprocess Only

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

When this option is used, the build sequence will terminate after the preprocessing stage, leaving behind files with the same basename as the corresponding source file and with a `.i` extension.

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.

4.6.2.3 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.

Only the base name of the file specified with this option has significance. You cannot use this option to change the extension of an output file.

4.6.2.4 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=16F877A -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.

If the assembler optimizers are enabled, the resulting output file is optimized by the assembler; otherwise the output is the raw code generator output. Optimized assembly files have many of the assembler directives removed.

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.

4.6.2.5 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.

4.6.2.6 X: Specify Source Language Option

The `-x`*`language`* option allows you to specify that the source files that follow are written in the specified language, regardless of the extension they use.

The languages allowed by the compiler are tabulated below.

Table 4-17. Language options

Language	Description
`assembler`	Assembly source code

...........continued	
Language	**Description**
`assembler-with-cpp`	Assembly source code that must be preprocessed

For example, the command:

```
xc8-cc -mcpu=18f4520 -c -xassembler-with-cpp init.s
```

will tell the compiler to run the preprocessor over the assembly source file, even though the `init.s` file name does not use a `.S` extension.

4.6.2.7 Help

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

4.6.2.8 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.

4.6.2.9 Version

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

4.6.3 Options for Controlling the C Dialect

The options shown in the table below define the kind of C dialect used by the compiler and are discussed in the sections that follow.

Table 4-18. C Dialect Control Options

Option	Controls
`-ansi`	The C language standard
`-f[no-]signed-char` `-f[no-]unsigned-char`	The signedness of a plain `char` type
`-mext=`*`extension`*	Which language extensions is in effect
`-std=`*`standard`*	The C language standard

4.6.3.1 Ansi Option

The `-ansi` option is equivalent to `-std=c90`, and controls the C standard used.

4.6.3.2 Signed-char Option

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

By default, the plain `char` type is equivalent to `unsigned char`.

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

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

4.6.3.3 Ext Option

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

Table 4-19. 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.

4.6.3.4 Std Option

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

Note that MPLAB XC8 uses a different compiler front end for these two standards, thus you might see a change in compiler behavior when swapping between standards.

Table 4-20. Acceptable C Language Standards

Standard	Supports
`c89` or `c90`	ISO C90 (ANSI) programs (using the P1 front end for all devices)
`c99`	ISO C99 programs (using the Clang front end for PIC18 and Enhanced mid-range devices)

4.6.3.5 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 `unsigned char`.

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.

4.6.4 Options for Controlling Warnings and Errors

Warnings are diagnostic messages that report constructions that are not inherently erroneous, but that are risky or suggest there may have been an error.

The options shown in the table below control the messages produced by the compiler and are discussed in the sections that follow.

Table 4-21. Warning And Error Options

Option	Controls
`-fmax-errors=`*`n`*	How many errors are output before terminating compilation
`-mwarn=`*`level`*	The threshold at which warnings are output
`-w`	The suppression of all warning messages
`-Wpedantic`	The acceptance of non-standard language extensions

4.6.4.1 Max Errors Option

The `-fmax-errors=`*`n`* option sets the maximum number of errors each compiler application (excluding the Clang front end), as well as the driver, will display before execution is terminated.

By default, up to 20 error messages will be displayed by each application before the application terminates. The option `-fmax-errors=10`, for example, would ensure the applications terminate after only 10 errors.

See 4.5 Compiler Messages for full details of the messaging system employed by `xc8-cc`.

4.6.4.2 Warn Option

The `-mwarn=level` option is used to set the warning level threshold. Allowable warning levels range from `-9` to `9`. The warning level determines how pedantic the compiler is about dubious type conversions and constructs. Each warning has a designated warning level; the higher the warning level, the more important the warning message. If the warning message's warning level exceeds the set threshold, the warning is printed. The default warning level threshold is `0` and will allow all normal warning messages.

Use this option with care as some warning messages indicate code that is likely to fail during execution, or compromise portability.

The warnings from the Clang front end are not controlled by this option.

The Compiler Messages section has full information on the compiler's messaging system.

4.6.4.3 W: Disable All Warnings Option

The `-w` option inhibits all warning messages. Use this option with care as some warning messages indicate code that is likely to fail during execution or compromise portability.

4.6.4.4 Pedantic Option

The `-Wpedantic` option is used to enable strict ANSI C conformance of all special, non-standard keywords when building C89/90 conforming programs.

If this option is used, non-standard keywords must include two leading underscore characters (for example, `__persistent`) so as to strictly conform to the C standard.

4.6.5 Options for Debugging

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

Table 4-22. Debugging Options

Option	Controls
`-f[no-]instrument-functions`	Instrumentation of the generated output to provide function profiling information.
`-gformat`	The type of debugging information generated.
`-mchp-stack-usage`	The generation of stack usage information and warnings
`-mcodecov`	Instrumention of the generated output to provide code coverage information.
`-save-temps`	Whether intermediate files should be kept after compilation.

4.6.5.1 Instrument Functions Option

The `-finstrument-functions` option embeds diagnostic code into the output to allow for function profiling with the appropriate hardware. See 5.2.13 Function profiling for more information.

4.6.5.2 G: Produce Debugging Information Option

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

The support formats are tabulated below.

Table 4-23. Supported Debugging File Formats

Format	Debugging file format
`-gcoff`	COFF
`-gdwarf-3`	ELF/DWARF release 3
`-ginhx32`	Intel HEX with extended linear address records, allowing use of addresses beyond 64kB
`-ginhx032`	INHX32 with initialization of upper address to zero

By default, the compiler produces DWARF release 3 files.

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

4.6.5.3 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 5.2.15 Stack Guidance for more information on the stack guidance reports that are produced by the compiler.

4.6.5.4 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 5.2.14 Code Coverage for more information.

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

4.6.5.5 Save-temps Option

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

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

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

4.6.6 Options for Controlling Optimization

The options shown in the table below, control compiler optimizations and are described in the sections that follow. The table also indicates whether a compiler license is required to select the optimization level. See 5.12 Optimizations for a description of the sorts of optimizations possible.

Many of these controls specify a set of optimizations and that some sets disable certain optimizations, thus a 'select all options' approach will not produce the most compact output.

Table 4-24. General Optimization Options

Option	License	Builds with
`-O0`	No	Minimal optimizations (default)
`-O` `-O1`	No	Optimization level 1
`-O2`	No	Optimization level 2
`-O3`	Yes	Optimization level 3
`-Og`	No	Less optimizations for better debugging
`-Os`	Yes	Size-orientated optimizations
`-fasmfile`	No	Optimizations applied to assembly source files
`-fcacheconst`	Yes	Objects qualified `const` potentially in RAM
`-flocal`	No	Local optimization set only
`--nofallback`	No	Only the selected optimization level and with no license-imposed fall back to a lesser level

4.6.6.1 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.

4.6.6.2 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.

4.6.6.3 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.

4.6.6.4 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.

4.6.6.5 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.

4.6.6.6 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.

4.6.6.7 Asmfile Option

The `-fasmfile` option enables assembler optimizations on hand-written assembly source files, but has no effect on intermediate assembly output produced from C source by the code generator. Although this option is always selectable, the actual optimizations performed will depend on whether you have a licensed or unlicensed compiler. By default hand-written assembly source code is not optimized.

4.6.6.8 Cacheconst Option

The `-fcacheconst=[on|off|auto]` option enables optimizations that can cache objects qualified `const` in any unused data memory, accessing them from this location rather than from program memory. This feature is only available for licensed compilers.

Code that accesses objects in program memory is often slower and longer than that which accesses objects in data memory; however, lesser amounts of device data memory compared to program memory are present, and data memory is the only memory that can be used for writable objects. This optimization permits programs to use all the data memory required and then automatically utilize any unused portions of this data memory for read-only objects that would otherwise be placed in program memory.

If the option argument is specified as `off`, the optimization is completely disabled and no objects normally allocated to program memory will be cached in data memory.

If the optimization is enabled (the default state when using optimization level `3`, or when using the `on` argument at optimization levels `3` or `s`), the compiler will perform the usual allocation of objects to data memory. If there are unused data memory locations after this step, eligible objects that would have otherwise been stored in program memory will be considered for storage in this free data memory. The compiler will cache objects to data memory prioritized by a number a factors until no more objects can be allocated there. The compiler will generate code to access these objects in data memory, which will typically run faster than code that would have accessed them in program memory.

When the optimization is enabled in auto mode (the default state when using optimization level `s`, or when using the `auto` argument at optimization levels `3` or `s`), the preliminary steps performed by the compiler are the same as those performed when the `on` argument to this option has been specified; however, caching of each eligible object will only take place if the compiler can determine that there will be no code size increase as a consequence. The code to access an object in data memory will always be smaller than that which accesses program memory; however, if an object is located in data memory, its assigned value must be copied there by the runtime startup code (see 5.9.2.1 Initialization Of Objects). In most instances, the increase in size of the startup code to initialize the objects in data memory will be far less than the reduction in code size realized for each access of the object, but where that is not the case, the compiler will not redirect the object to data memory when this form of the option is used.

If the `-fcacheconst` option is not specified, the optimization will be disabled when selecting any of optimization levels `0`, `1`, or `2` (as if the `-fcacheconst=off` was specified); the optimization will be enabled when selecting optimization level `3` (as if `-fcacheconst=on` was specified); and the optimization will be set to its auto mode when using optimization level `s` (as if `-fcacheconst=auto` was specified). When using an optimization level of 2 or lower, the optimization cannot be enabled and use of the `-fcacheconst` option will be ignored.

As programs that use either a software stack (for any functions compiled to use a reentrant stack model) or a heap (used by dynamic memory allocation functions) typically require as much data memory as possible for these constructs, this optimization is disabled by default when these constructs are used, regardless of the optimization level. You can force the optimization to be applied (optimization level permitting) by using the `-fcacheconst` option in these cases. The optimization is disabled if local optimizations (`-flocal` option) have been enabled, but in this case, the optimization cannot be enabled using the `-fcacheconst` option.

Any object that is absolute (defined using `__at()`) or any object in a section (defined using `__section()`) or any object marked as `volatile` will never be cached; however, you should not make assumptions as to which other objects will be cached by this optimization, nor where cached objects will be located. As a program is developed, the objects cached and where they are placed can vary from build to build. Ensure that this optimization is disabled if you access C-defined program memory objects from assembly code.

As this optimization attempts to have programs utilize all of the device's data memory, information is provided in the map to indicate how much data memory is purely being used for the caching of `const`-qualified objects. It is important to realize that although it might appear that data memory is full or almost full when this optimization is enabled, you might still be able to define additional objects in your program. When you next build, the compiler will attempt to allocate these objects to data memory in the usual way, and if successful, there will be fewer `const`-qualified objects cached.

4.6.6.9 Local Option

The `-flocal` option limits the extent to which some optimizations are applied to the program.

This option will use omniscient code generation (OCG) optimizations with libraries or individual program modules but have the scope of those optimizations restricted to code within those libraries or modules. Normally optimizations in one module can be affected by code in other modules or libraries, and there are situations where you want to prevent this from occurring. The output of source compiled with this setting enabled with typically be larger but will change little from build to build, even if other project code that does not use this setting is modified. Such changes in the output might be undesirable if you have validated code that is to be distributed and used in many different applications.

All the source code specified with a build command that uses local optimizations constitutes one group and you can create as many groups as required by building source code with separate build commands. Any code built without local optimizations becomes part of the default (unrestricted) group. The standard libraries that are supplied with the compiler are built with local optimizations disabled and are always part of this default group.

Enabling local optimizations restricts the scope of many optimizations, but does not necessarily disable the optimizations themselves. Optimizations can still be performed within each group, but those optimizations will not be made if they depend on code that is contained in another group. For example, abstraction of common code sequences will not be made if the sequences are contained in different groups, but would be made if the sequences are from the same group. Since a group can be limited to just a few modules of source code (which you can build into a library in the usual way if you prefer), this still allows you to fully optimize the bulk of a project.

By default this option is disabled. It can be enabled when building for enhanced mid-range and PIC18 devices and an error message will be emitted if the optimization is selected with an incompatible device.

When code is built with local optimizations, all variables defined in that group are allocated to banked memory unless they are qualified with near. Bank selection instructions are often output when they might normally have been emitted. Page selection instructions before and after function calls are always output, constant propagation is disabled, floating-point type sizes are fixed at 32 bits for both `float` and `double` types (and this will be enforced for the entire program) and pointer sizes can be fixed based on their definition (see 5.3.6.2 Pointer-target Qualifiers). Some assembly optimizations are also restricted, such as procedural abstraction, routine inlining, psect merging, and peephole optimizations.

4.6.6.10 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.

4.6.7 Options for Controlling the Preprocessor

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

Table 4-25. Preprocessor Options

Option	Controls
`-D`*`macro`* `-D`*`macro`*`=`*`text`*	The definition of preprocessor macros
`-M`	Generation of dependencies
`-mcmacros`	Predefine macros relevant for C programs
`-MD`	Generation of dependencies
`-MF`	Where dependency information is written
`-MM`	Generation of dependencies
`-MMD`	Generation of dependencies
`-U`*`macro`*	The undefinition of preprocessor macros
`-Wp,`*`option`*	Options passed to the preprocessor
`-Xpreprocessor` *`option`*	Options passed to the preprocessor

4.6.7.1 D: Define a Macro

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

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

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

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

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

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

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

See 5.13.1.1 Preprocessor Arithmetic for clarification of how the replacement text might be used.

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

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

4.6.7.2 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 dependencies is printed to a file with a `.d` extension and compilation will terminate after preprocessing.

4.6.7.3 Cmacros Option

The `-mcmacros` option requests that when building a project, the command-line driver define preprocessor macros that are relevant for C programs. This is the default action taken by the driver.

The `-mno-cmacros` form of this option can be used if you prefer that C-related preprocessor macros are not defined. Those macros that relate to the device are still defined (e.g. `__16F1937` or `_ROMSIZE`), but any macros that relate to the C program are suppressed (e.g. `__XC8`, `__OPTIM_FLAGS`, or `__SIZEOF_CHAR__`). Use the `-mno-cmacros` option with caution. Ensure the operation of your project is not dependent on the presence of or value equated to these macros.

4.6.7.4 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.

4.6.7.5 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.

4.6.7.6 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.

4.6.7.7 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.

4.6.7.8 U: Undefine Macros

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

Any macro defined using `-D` will be undefined by this option. All `-U` options are evaluated after all `-D` options.

4.6.7.9 Wp: Pass Option To The Preprocessor Option

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

4.6.7.10 Xpreprocessor Option

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

4.6.8 Options for Parsing

The options shown in the table below control parser operations and are discussed in the sections that follow.

Table 4-26. Parser Options

Option	Controls
`-Xparser` *option*	Options to passed to the parser

4.6.8.1 Xparser Option

The `-Xparser` *option* option passes its option argument directly to the parser. For example, -`Xparser -v` runs the parser in verbose mode. The options `-Xp1` and `-Xclang` are alternate forms of this option.

4.6.9 Options for Assembling

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

Table 4-27. Assembly Options

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

4.6.9.1 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.

4.6.9.2 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.

4.6.10 Mapped Assembler Options

The option tabulated below is a commonly used assembler option.

Table 4-28. Mapped Assembler Options

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

4.6.11 Options for Linking

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

Table 4-29. Linking Options

Option	Controls
`-l`*library*	Which library files are scanned
`-mc90lib`	Whether C90-compliant libraries will be linked
`-m[no-]clink`	Generate and use linker options relevant for C programs
`-mserial=`*options*	The insertion of a serial number in the output
`-nodefaultlibs`	Whether library code is linked with the project
`-nostartfiles`	Whether the runtime startup module is linked in

...........continued	
Option	**Controls**
`-nostdlib`	Whether the library and startup code is linked with the project
`-Wl,option`	Options to passed to the linker
`-Xlinker option`	System-specific options to passed to the linker
`--fill=options`	Filling of unused memory

4.6.11.1 L: Specify Library File Option

The `-llibrary` option looks for the specified file (with a `.a` extension) and scans this library archive for unresolved symbols when linking.

The directories searched include several standard system directories, plus any that you specify with `-L`.

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

4.6.11.2 c90lib Option

The `-mc90lib` option links in a set libraries that are compatible with the C90 language standard. By default, a C99 compatible library set is linked.

This option does not change the language specification used for source code in your project; use the `-std` option for that purpose. (See 4.6.3.4 Std Option.)

4.6.11.3 Clink Option

The `-mclink` option requests that when building a project, the command-line driver generate and use linker options that are relevant for C programs. This is the default action taken by the driver.

The `-mno-clink` form of this option can be used if you prefer to manually specify the linker options for a C program. The linker class options will still be defined, but any options that place psects into classes (for example `-p` linker options) will not be present. Use the `-mno-clink` option with caution. Ensure suitable linker options are manually specified to ensure the correct operation of your project.

4.6.11.4 Serial Option

The `-mserial=options` option allows a hexadecimal code to be stored at a particular address in program memory. A typical task for this option might be to position a serial number in program memory.

The byte-width of data to store is determined by the byte-width of the hexcode parameter in the option. For example, to store a one-byte value, 0, at program memory address 1000h, use `-mserial=00@1000`. To store the same value as a four byte quantity use `-mserial=00000000@1000`.

This option is functionally identical to Hexmate's `-serial` option. For more detailed information and advanced controls that can be used with this option (refer to 8.2.2.21 Serial).

The driver will also define a label at the location where the value was stored and can be referenced from C code as `_serial0`. To enable access to this symbol, remember to declare it, for example:

```
extern const int _serial0;
```

4.6.11.5 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.

4.6.11.6 Nostartfiles Option

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

4.6.11.7 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.

4.6.11.8 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. This means that this option can not be used to pass in a linker option such as `-pcodeStart,codeEnd`, which uses the comma to separate psect names. In this case, use the `-Xlinker` option.

4.6.11.9 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.

For example `-Xlinker -presetVec=0h,intCode=04h`

4.6.11.10 Fill Option

The `--fill=`*`options`* option allows you to fill unused memory with specified values in a variety of ways.

This option is functionally identical to Hexmate's `-fill` option. For more detailed information and advanced controls that can be used with this option, refer to 8.2.2.12 Fill.

4.6.12 Mapped Linker Options

The options shown in the table below are commonly used linker options.

Table 4-30. Mapped Linker Options

Option	Controls
`-Wl,--[no-]data-init`	Clearing and initialization of C objects at runtime startup
`-Wl,-Map=`*`mapfile`*	The generation of a linker map file

4.6.13 Options for Directory Search

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

Table 4-31. Directory Search Options

Option	Controls
`-I`*`dir`*	The directories searched for preprocessor include files
`-L`*`dir`*	Directories searched for libraries
`-nostdinc`	Directories searched for headers

4.6.13.1 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.

4.6.13.2 L: Specify Library Search Path Option

The `-L`*`dir`* 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.

4.6.13.3 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.

4.6.14 Options for Code Generation Conventions

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

Table 4-32. Code Generation Convention Options

Option	Controls
`-f[no-]short-double`	The size of the `double` type
`-f[no-]short-float`	The size of the `float` type

4.6.14.1 Short Double Option

The `-fshort-double` option controls the size of the `double` type.

When building to the C99 standard, all floating-point types must be the IEEE754 32-bit format. If you are building for C90 standard, you may use this option to explicitly request the 24-bit form of this format for `double` objects. When using the `-fno-short-double` form of the option, the `double` type can be changed to the full 32-bit IEEE754 format. The selection of this option must be consistent across all modules of the program.

4.6.14.2 Short Float Option

The `-fshort-float` option controls the size of the `float` type.

When building to the C99 standard, all floating-point types must be the IEEE754 32-bit format. If you are building for C90 standard, you may use this option to explicitly request the 24-bit form of this format for `float` objects. When using the `-fno-short-float` form of the option, the `float` type can be changed to the full 32-bit IEEE754 format. The selection of this option must be consistent across all modules of the program.

4.7 MPLAB X IDE Integration

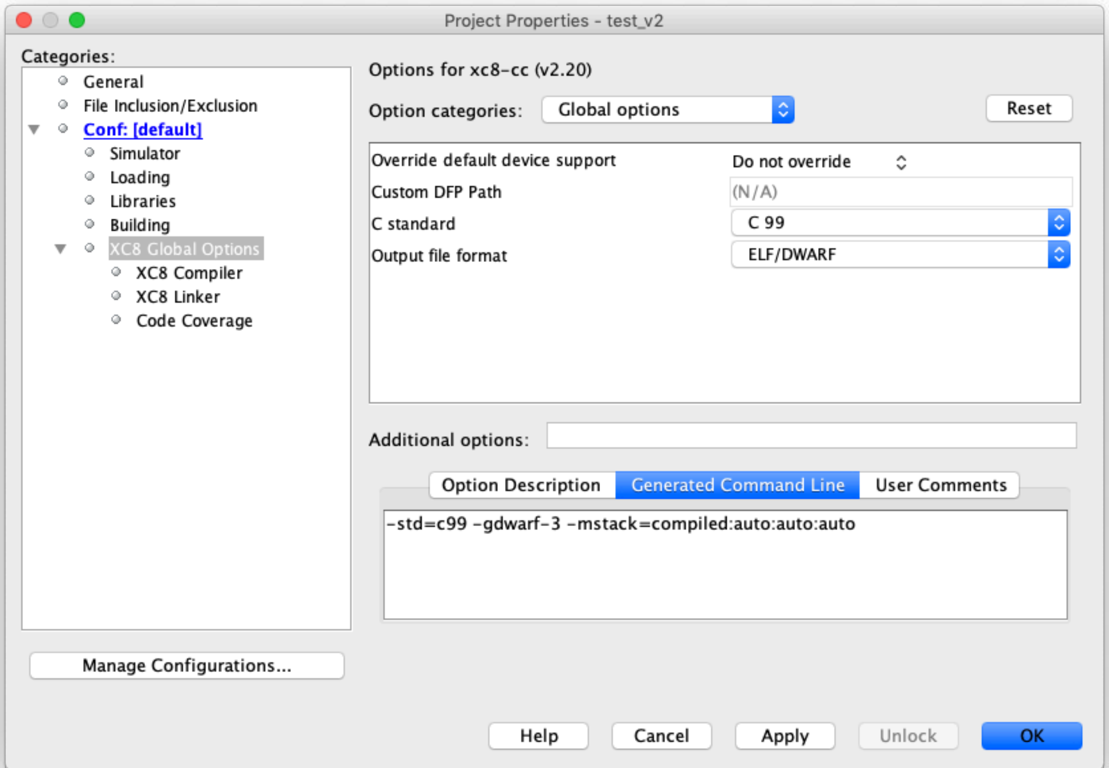
The 8-bit language tools may be integrated into and controlled from the MPLAB X IDE, to provide a GUI-based development of application code for the 8-bit PIC MCU families of devices.

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*.

4.7.1 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.

4.7.1.1 XC8 Global Options - Global options



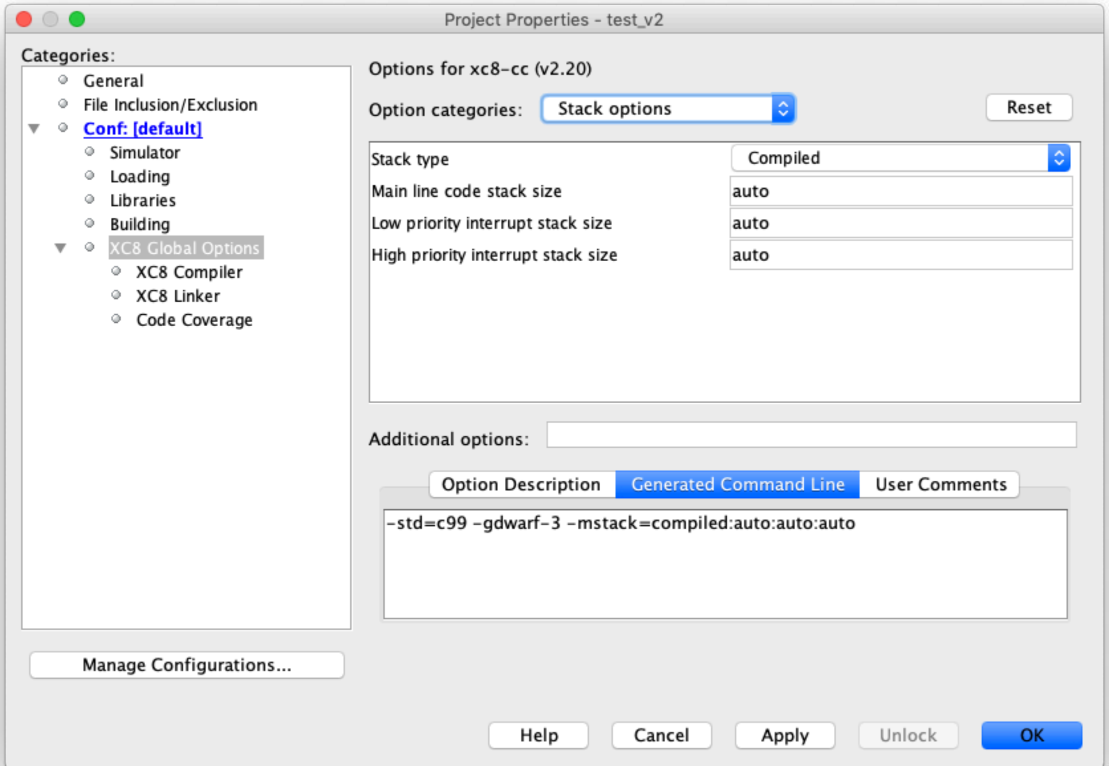
Override default device support This selector allows you to indicate how Device Family Pack (DFP) management should be performed. The **Do not override** selection will let the MPLAB X IDE provide a list of DFPs that can be selected. If you would like to use a DFP that you have manually downloaded, select **User specified location** and then enter the path to the DFP in the **Customer 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.

C standard This selector specifies the language standard used. See 4.6.3.4 Std Option.

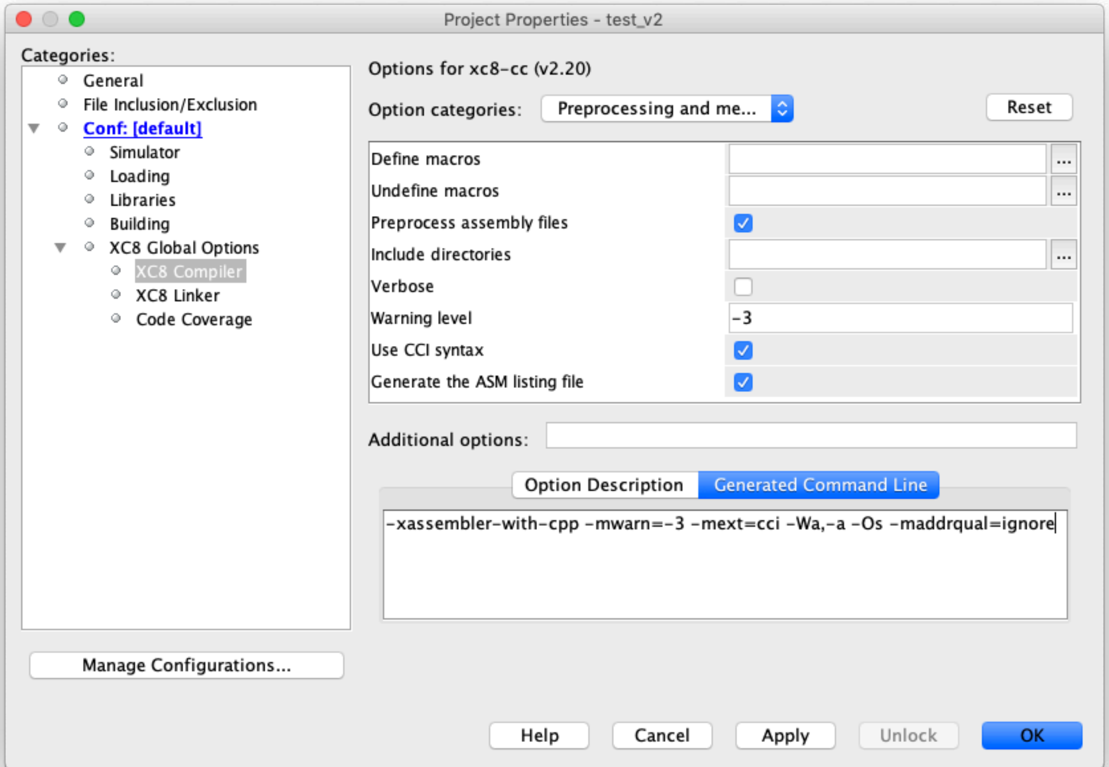
Output file format This selector specifies the output file type. See 4.6.5.2 G: Produce Debugging Information Option.

4.7.1.2 XC8 Global Options - Stack options



Stack options All the fields in this dialog correspond to which data stack is used and how that is configured. See 4.6.1.22 Stack Option.

4.7.1.3 XC8 Compiler - Preprocessing and messaging options



Define macros This field allows you to define preprocessor macros. See 4.6.7.1 D: Define a Macro.

Undefine macros This field allows you to undefine preprocessor macros. See 4.6.7.8 U: Undefine Macros.

Preprocess assembly files This checkbox is used to preprocess assembly source files. See the 4.6.9.2 Xassembler Option.

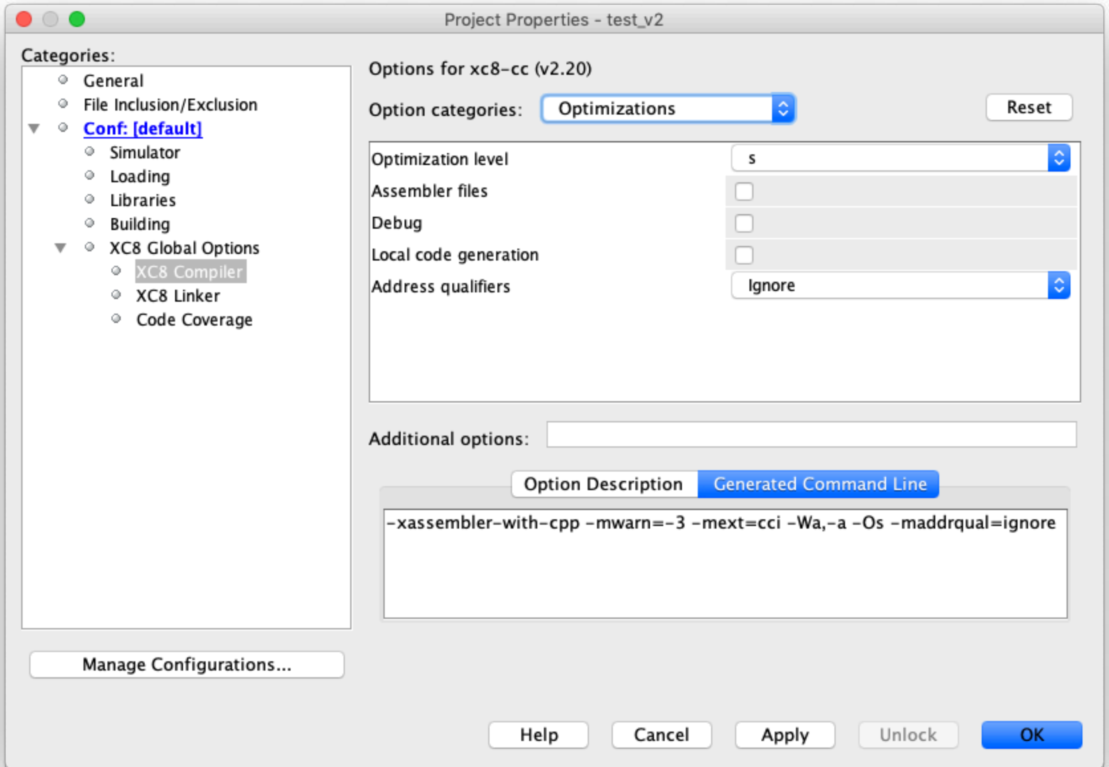
Include directories This field allows you to specify the directories searched for header files. See 4.6.13.1 I: Specify Include File Search Path Option.

Verbose This checkbox shows the build command lines. See 4.6.2.5 V: Verbose Compilation.

Use CCI syntax This checkbox requests that the CCI language extension be enforced. See 4.6.3.3 Ext Option.

Generate the ASM listing file This checkbox generates an assembly listing file. See 4.6.10 Mapped Assembler Options.

4.7.1.4 XC8 Compiler - Optimizations options



Optimization level	This selector controls the level to which programs are optimized. See 4.6.6 Options for Controlling Optimization.
Assembly files	This checkbox allows optimization of assembly source files. See 4.6.6.7 Asmfile Option.
Debug	This checkbox inhibits aggressive optimization that can impact on the debugability of code. See 4.6.6.5 Og: Better Debugging Option.
Local code generation	This checkbox limits the extent to which some optimizations are applied to the program. See 4.6.6.9 Local Option.
Address qualifiers	This selector controls the compiler’s response to non-standard memory qualifiers in C source code. See 4.6.1.1 Addrqual Option.

4.7.1.5 XC8 Linker - Runtime options

Figure 4-2. Linker runtime options displayed for PIC18 devices

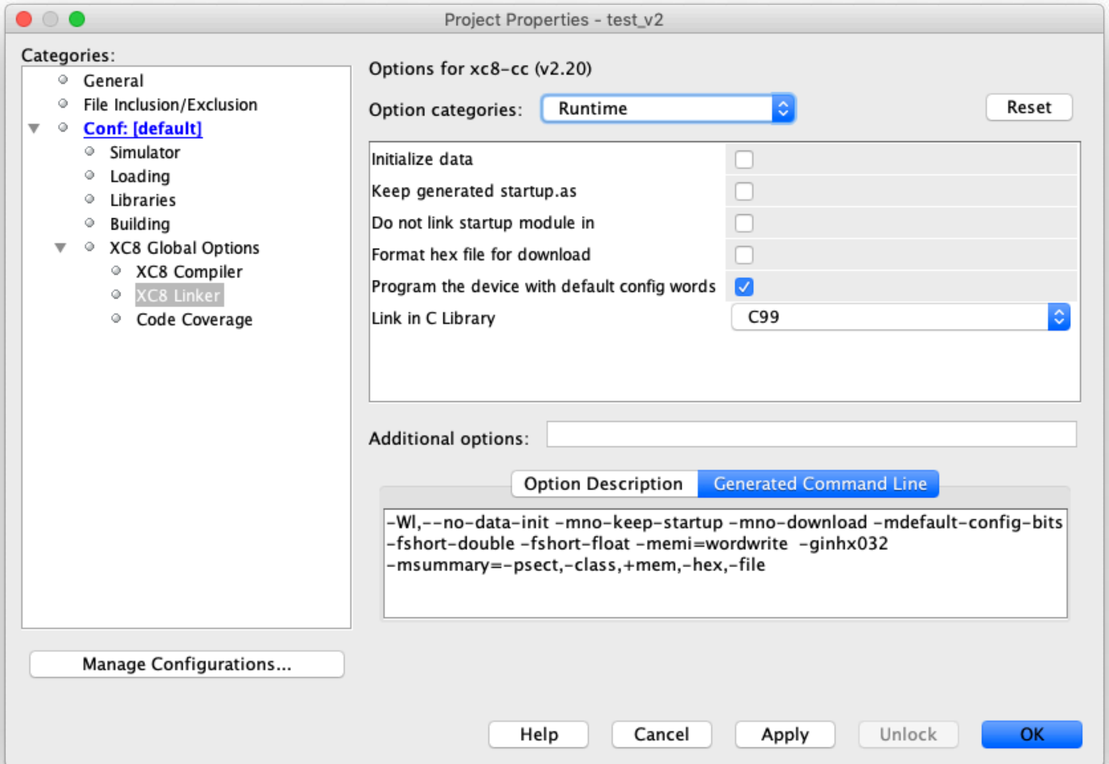
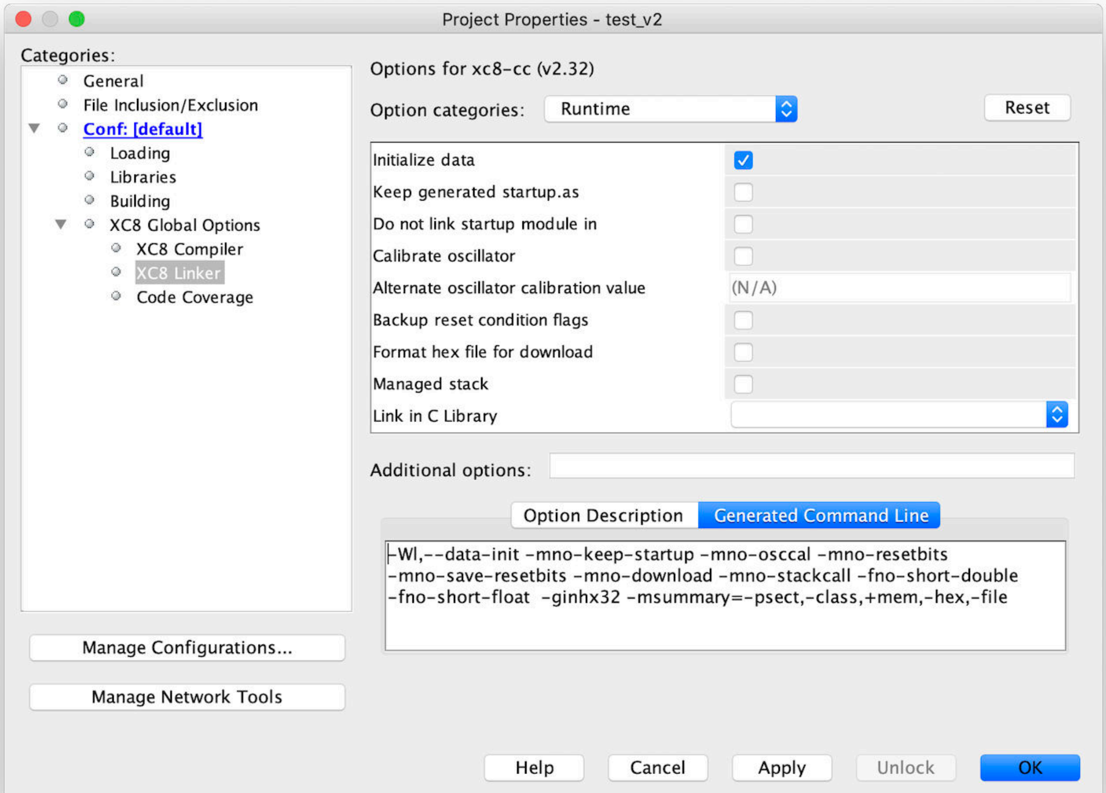


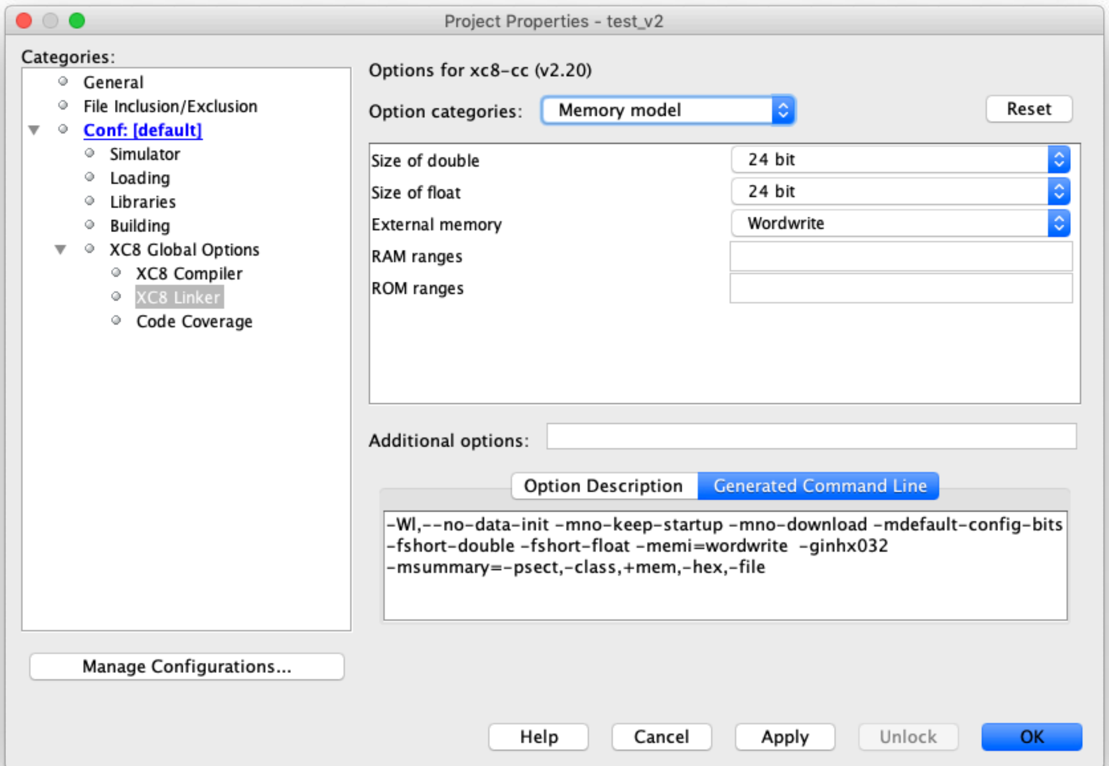
Figure 4-3. Linker runtime options displayed for Mid-range and Baseline devices



Initialize data	This checkbox controls whether initialized C objects are assigned their starting value. See 4.6.12 Mapped Linker Options.
Keep generated startup.as	This checkbox controls whether the runtime startup module is retained after compilation. See 4.6.1.13 Keep Startup Option.
Do not link startup module in	This checkbox controls whether the runtime startup module linked into the program. See 4.6.11.6 Nostartfiles Option.
Calibrate oscillator	This checkbox controls whether the oscillator calibration constant is applied by runtime startup module linked into the program. Not all devices uses this calibration method. See 4.6.1.15 Osccal Option.
Alternate oscillator calibration value	This field allows you to specify an alternate oscillator calibration constant to that preprogrammed into the device. Not all devices uses this calibration method. See 4.6.1.16 Oscval Option.
Backup reset condition flags	This checkbox controls whether the runtime startup code will preserve the state of the status register, so that it can later be examined to determine the cause of Reset. This option is not implemented for PIC18 device. See 4.6.1.19 Resetbits Option.
Format hex file for download	This checkbox controls the special formatting of the final HEX file. See 4.6.1.8 Download Option.
Managed stack	This checkbox controls whether a table look-up method of calling functions will be employed. See 4.6.1.23 Stackcall Option.

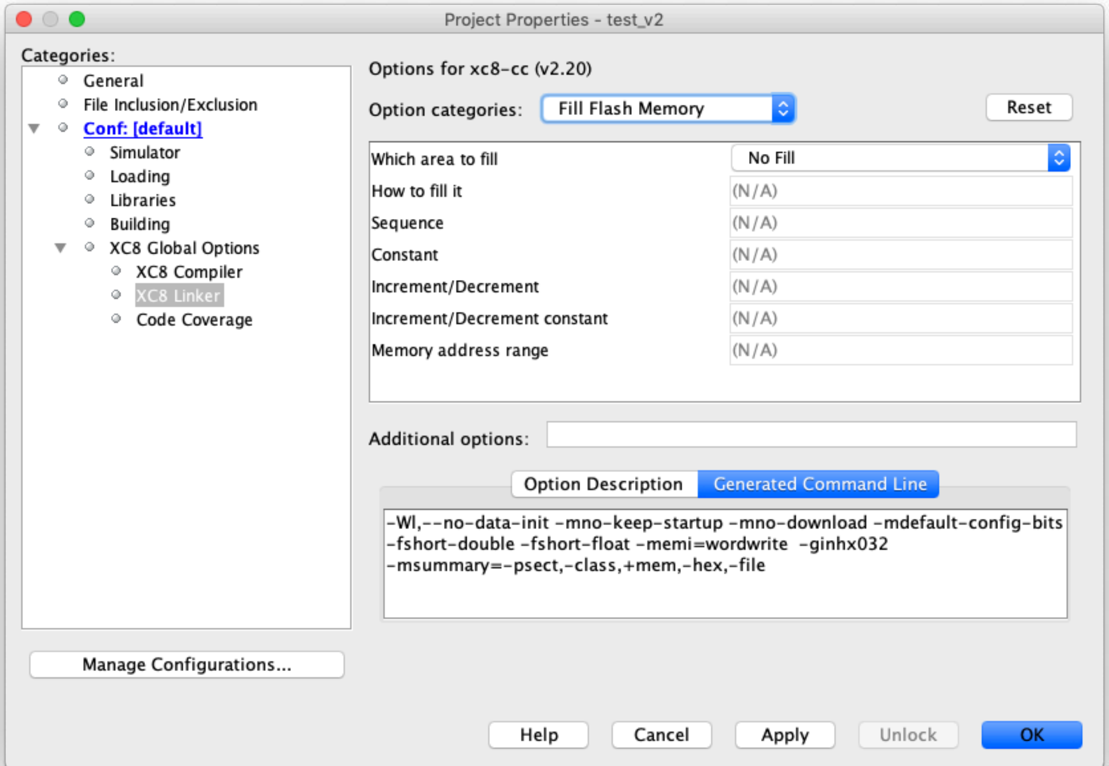
Program the device with default config words	This checkbox controls whether unspecified configuration bits will be programmed with default values. See 4.6.1.4 Config Option.
Link in C library	This selector controls the inclusion and language standard of libraries linked into the program. This utilizes the `-mc90lib` option (see 4.6.11.2 c90lib Option) and the `-nodefaultlibs` option, see 4.6.11.5 Nodefaultlibs Option.

4.7.1.6 XC8 Linker - Memory model options



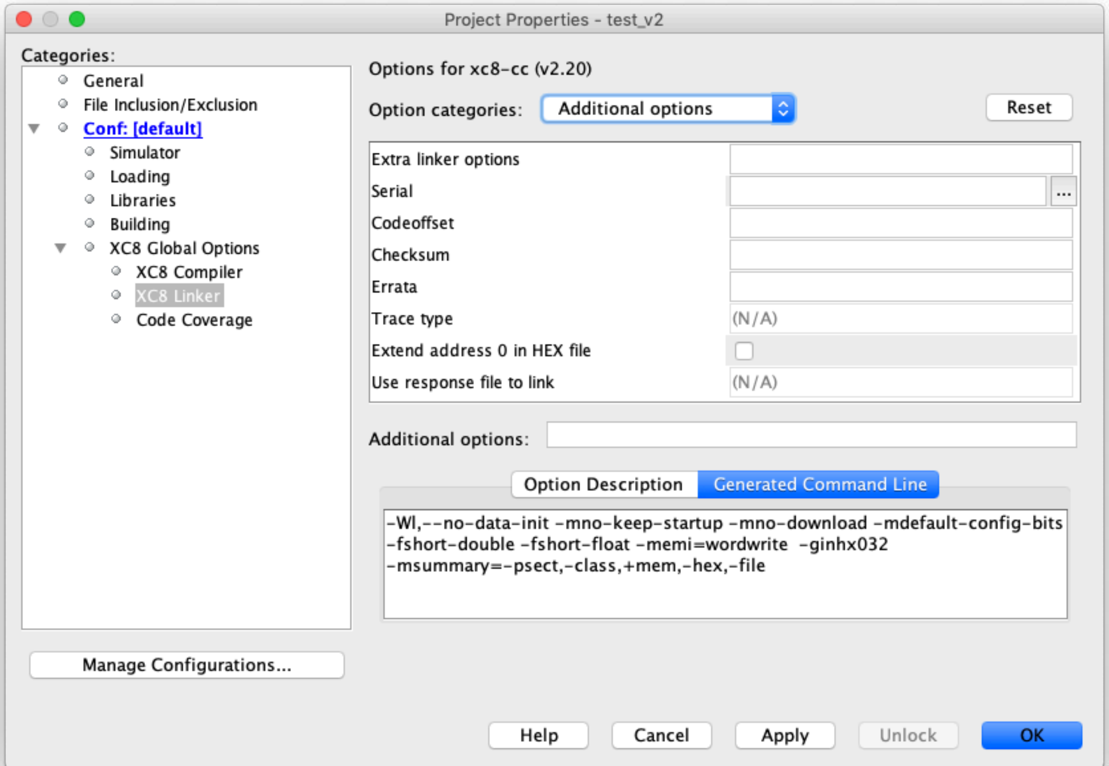
Size of double	This selector controls the size of the double type. See 4.6.14.1 Short Double Option.
Size of floats	This selector controls the size of the float type. See 4.6.14.2 Short Float Option.
External memory	This selector controls how external memory is accessed. See 4.6.1.9 Emi Option.
RAM ranges	This field adjusts the data memory used by the program. See 4.6.1.17 Ram Option.
ROM ranges	This field adjusts the program memory used by the program. See 4.6.1.20 Rom Option.

4.7.1.7 XC8 Linker - Fill flash memory options



Fill options All the fields in this dialog correspond to filling unused memory with values. See 4.6.11.10 Fill Option.

4.7.1.8 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 4.6.11.8 Wl: Pass Option To The Linker, Option.

Serial This field allows you to have codes placed into the HEX file. See 4.6.11.4 Serial Option.

Codeoffset This field allows you to move the entire program image up in memory. See 4.6.1.3 Codeoffset Option.

Checksum This field allows you to have a hash value calculated and inserted into the HEX file. See 4.6.1.2 Checksum Option.

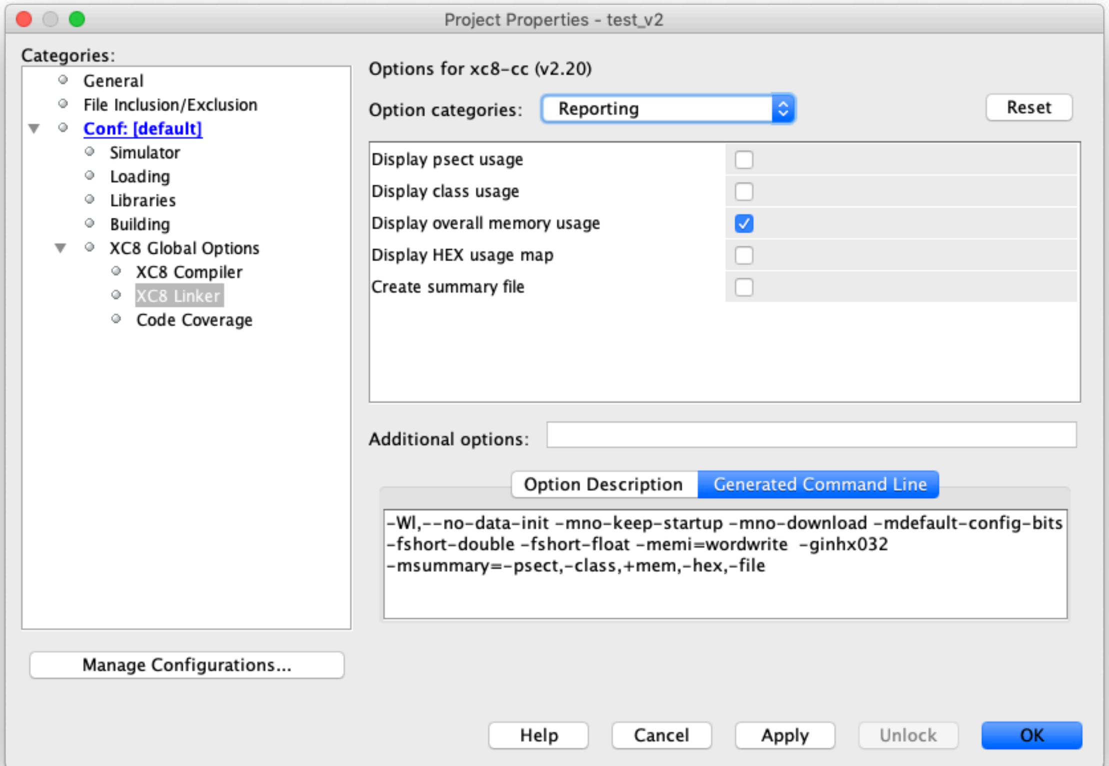
Errata This field allows you to adjust the errata workarounds employed by the compiler. See 4.6.1.10 Errata Option.

Trace type This selector is not yet implemented. Native trace is supported.

Extend address in HEX file This checkbox generates HEX files with extended liner addresses. See 4.6.5.2 G: Produce Debugging Information Option.

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.

4.7.1.9 XC8 Linker - Reporting options



Display psect usage This checkbox allows you to have psect (section) information included in the post-build summary. See 4.6.1.24 Summary Option.

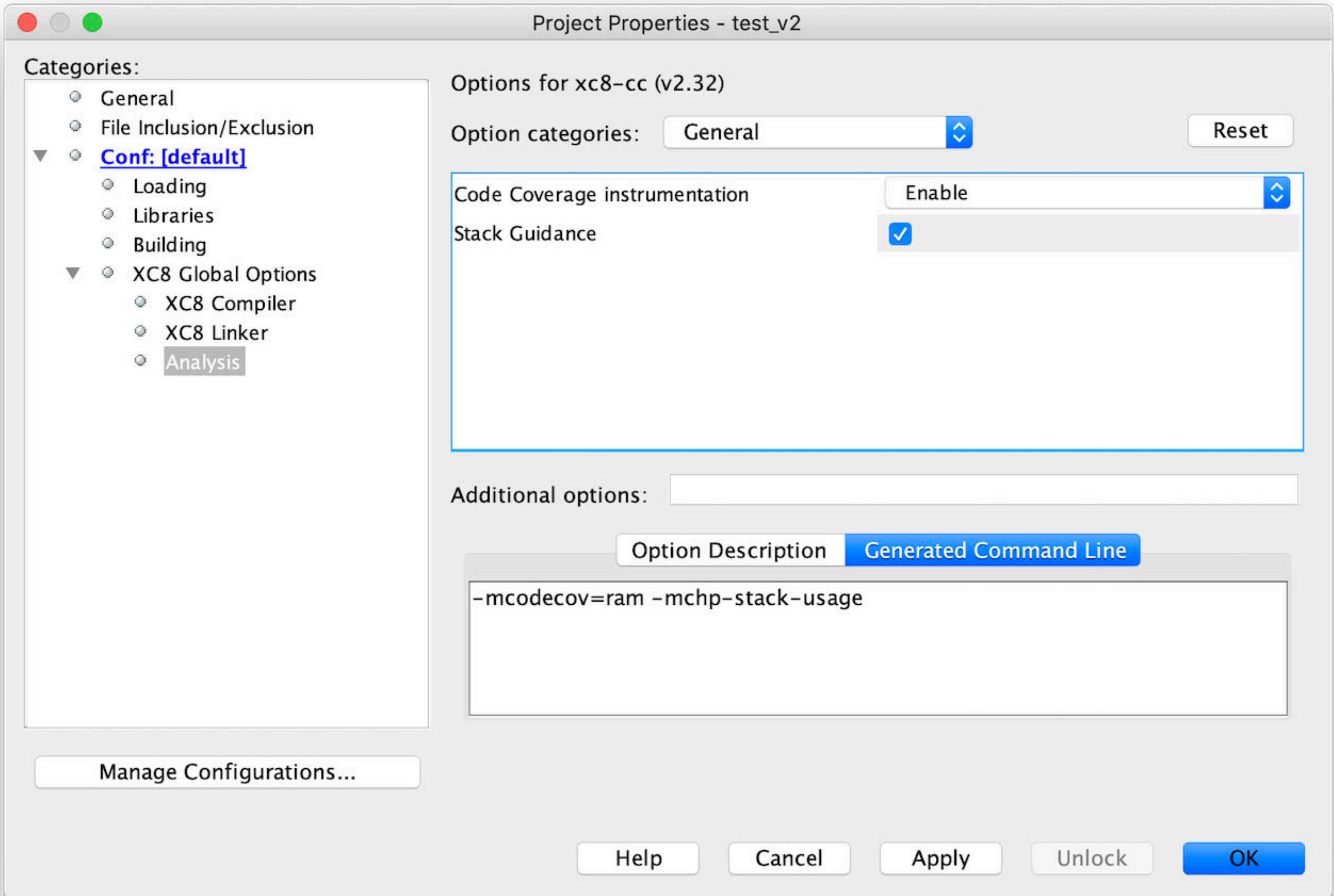
Display class usage This checkbox allows you to have linker class (memory area) information included in the post-build summary. See 4.6.1.24 Summary Option.

Display overall memory summary This checkbox allows you to have general memory usage included in the post-build summary. See 4.6.1.24 Summary Option.

Display HEX usage map This checkbox allows you to see a graphical representation of the usage of the device memory included in the post-build summary. See 4.6.1.24 Summary Option.

Create summary file This checkbox allows you to have the post-build summary redirect to a file. See 4.6.1.24 Summary Option.

4.7.1.10 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 4.6.5.4 Codecov Option and 5.2.14 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 4.6.5.3 Stack Guidance Option and 5.2.15 Stack Guidance.

5. 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 PIC devices. This chapter documents the special language features that are specific to these devices.

5.1 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 4.6.3.4 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.

5.1.1 Common C Interface Standard

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

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

CCI can be enforced by using the `-mext=cci` option (see 4.6.3.3 Ext Option).

5.1.2 Divergence from the C90 Standard

The C language implemented on MPLAB XC8 C Compiler can diverge from the C90 Standard in several areas detailed in the sections below.

5.1.2.1 Reentrancy

MPLAB XC8 C Compiler does not support function reentrancy for all devices.

This divergence from the C standard is due to there being no hardware implementation of a data stack on some devices. See 5.2.4 Stacks for more information on the stack models used by the compiler for each device family.

Functions can be encoded reentrantly for Enhanced Mid-range and PIC18 devices, provided that the software stack is used for data. Recursive functions are also possible with these devices. Baseline and other Mid-range devices do not support a software stack and functions cannot be made reentrant.

For those devices that do not support reentrancy or if you stipulate that functions must use the compiled stack, the compiler can make functions called from main-line and interrupt code appear to be reentrant via a duplication feature. See 5.8.7 Function Duplication for more about duplication.

5.1.2.2 Sizeof Operator With Pointer Types

The C `sizeof` operator when acting on pointer *types* or on structure or array *types* that contain a pointer may not work reliably. This operator, however, may be used with pointer, structure, or array *identifiers*. This is due to the dynamic nature of pointer size used by the compiler.

For the following code:

```
char * cp;
size_t size;
size = sizeof(char *);
size = sizeof(cp);
```

`size` in the first example will be assigned the maximum size a pointer can be for the particular target device you have chosen. In the second example, `size` will be assigned the actual size of the pointer variable, `cp`.

The `sizeof` operator using a pointer variable operand cannot be used as the number-of-elements expression in an array declaration. For example, the size of the following array is unpredictable:

```
unsigned buffer[sizeof(cp) * 10];
```

5.1.2.3 Empty Function Parameter List

When building projects that conform to the C90 standard, the compiler assumes that functions declared with an empty parameter list have no parameters.

According to the C standard, functions *defined* with an empty parameter list, as in the following example:

```
int range()
{ ... }
```

are assumed to take no arguments, as if `void` was specified in the brackets. A similar prototype in a *declaration*, as in the following example:

```
extern int range();
```

should specify that no function parameter information has been provided. In this instance, MPLAB XC8 C Compiler instead assumes that the function has no parameters.

This limitation is not present when you are compiling against the C99 standard, where such a declaration would be correctly interpreted to mean that no information is specified regarding the function’s parameters.

5.1.2.4 Const Auto Objects

If a Baseline or Mid-range device is selected, any `auto` objects that are specified as `const` are treated as if they were also specified `static`. This does not affect the scope of the object, but does imply that its value will be stored in program memory and any initializer assigned to the variable when it is defined must be a constant expression.

This limitation does not affect projects using PIC18 devices, since `const auto` objects with these devices are stored in data memory.

5.1.3 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.

5.1.3.1 Library Support

C99-compliant C libraries are shipped with this product; however, some functionality is not present. See the compiler's release notes for specific information on known issues.

5.1.3.2 Inlined Functions

There are several areas where the inlining of functions deviates from the C standard.

The standard states that an `inline` function should only be inlined in the translation unit in which it is defined. With MPLAB XC8, inlining can occur in any module.

The standard indicates that if no `extern` declaration is provided in the same translation unit as an `inline` function, then the definition is an *inlined definition*. This also means that it does not provide an external definition of that function and poses as an alternative to an external definition. With this implementation, an external definition is implied by an `inline` function.

The standard states that if an external definition is provided in addition to an `inline` definition, then it is unspecified which definition the compiler should use in the translation unit of the inlined definition. With MPLAB XC8, a function redefinition error will be emitted if both `inline` and external definitions are encountered.

5.1.3.3 Aliasing Using Effective Type

The compiler neither checks for aliased types, nor performs any optimizations that could fail as a result of aliased types.

5.1.3.4 Restrict Pointer-type Qualifier

The `restrict` pointer-type qualifier is allowed in programs, but will be ignored by the compiler.

5.1.3.5 Variable Length Arrays

The size of arrays must be known at compile time. Thus the dimensions of arrays must be constant expressions. Function prototypes cannot use the `[*]` syntax with an array to indicate a variable length array type.

5.1.3.6 Flexible Array Members

The compiler will not accept an incomplete type array as the last member in a structure. All array members of a structure must specify a number of elements.

5.1.3.7 Complex Number Support

Complex types are not supported and use of the `_Complex` and `_Imaginary` and related types will trigger a warning and will be ignored. The `<complex.h>` header is also not supported.

5.1.3.8 Extended Identifiers

C identifiers cannot currently use extended characters.

5.1.4 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 11. Implementation-Defined Behavior.

5.2 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.

5.2.1 Device Support

The MPLAB XC8 C Compiler aims to support all 8-bit PIC and AVR devices. This user's guide should be consulted when you are programming PIC devices; when programming AVR targets, see the MPLAB XC8 C Compiler User's Guide for AVR® MCU.

New PIC 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 4.6.2.8 Print-devices). A list of all devices will be printed.

You can also see the supported devices in your favorite web browser. Open the files `pic_chipinfo.html` for a list of all supported baseline or mid-range device, or `pic18_chipinfo.html` for all PIC18 devices. Both these files are located in the `docs` directory under your compiler's installation directory.

5.2.2 Instruction Set Support

The compiler supports all instruction sets for PIC10/12/16 devices as well as the standard (legacy) PIC18 instruction set. The extended instruction mode available on some PIC18 devices is not currently supported and setting the configuration bit (typically `XINST`) to enable this instruction set will trigger an error from the compiler.

5.2.3 Device Header Files

There is one header file that is typically included into each C source file you write. The `<xc.h>` file is a generic header file that will include other device- and architecture-specific header files when you build your project.

Inclusion of this file will allow access to SFRs via special variables, as well as macros which allow special memory access or inclusion of special instructions, like `CLRWDT()`.

Do not include chip-specific header files in your code, as this will reduce portability and these headers may not contain all the required definitions for the successful compilation of your code.

The header files shipped with the compiler are specific to that compiler version. Future compiler versions may ship with modified header files. Avoid including header files that have been copied into your project. Such projects might no longer be compatible with future versions of the compiler.

For information about assembly include files (`.inc`) (see 5.11.3.4 Accessing Registers From Assembly Code).

5.2.4 Stacks

Stacks are used for two different purposes by programs running on 8-bit PIC devices: one stack is for storing function return addresses and one or two other stacks are used for data allocation.

5.2.4.1 Function Return Address Stack

The 8-bit PIC devices use a hardware stack for function return addresses. This stack cannot be manipulated directly and has a limited depth, which is indicated in your device data sheet and the relevant device chipinfo file.

Nesting functions too deeply can exceed the maximum hardware stack depth and lead to program failure. Remember that interrupts and implicitly called library functions also use this stack.

The compiler can be made to manage stack usage for some devices using the `-mstackcall` option (see 4.6.1.23 Stackcall Option). This enables an alternate means of calling functions to allow function nesting deeper than the stack alone would otherwise allow.

A call graph is provided by the code generator in the assembler list file (see 6.3.6 Call Graph). This will indicate the stack levels at each function call and can be used as a guide to stack depth. The code generator can also produce warnings if the maximum stack depth is exceeded.

The warnings and call graphs are guides to stack usage. Optimizations and the use of interrupts can decrease or increase the program's stack depth over that determined by the compiler.

5.2.4.2 Data Stacks

The compiler can implement two types of data stack: a compiled stack and a software stack. Both these stacks are for storing stack-based variables which have automatic storage duration, such as auto, parameter, and temporary variables.

Either one or both of these types of stacks may be used by a program. Compiler options, specifiers and how the functions are called will dictate which stacks are used. See 5.4.2.2 Automatic Storage Duration Objects for more information on how the compiler allocates a function's stack-based objects.

A section, called `stack`, reserves the memory used by the stack.

5.2.4.2.1 Compiled Stack Operation

A compiled stack is one or more areas of memory that are designated for automatic storage duration objects. Objects allocated space in the compiled stack are assigned a static address which can be accessed via a compiler-allocated symbol. This is the most efficient way of accessing stack-based objects, since it does not use a stack pointer.

Functions which allocate their stack-based objects in the compiled stack will not be reentrant, since each instance of the functions' objects will be accessed via the same symbols (see 5.11.3 Interaction between Assembly and C Code). Memory in a compiled stack can be reused, just like that in a conventional software stack. If two functions are never active at the same time, then their stack-based objects can overlap in memory with no corruption of data. The compiler can determine which functions could be active at the same time and will automatically reuse memory when possible. The compiler takes into account that interrupt functions, and functions they call, need their own dedicated memory (see also 5.8.7 Function Duplication).

The size of the compiled stack can be determined at compile time, so available space can be confirmed by the compiler.

5.2.4.2.2 Software Stack Operation

A software stack is a dynamic allocation of memory that is used for automatic storage duration objects and which is indirectly accessed via a stack pointer. Although access of objects on a software stack can be slower, functions which use a software stack are reentrant. This form of stack is available only for Enhanced Mid-range and PIC18 devices.

As functions are called, they allocate a chunk of memory for their stack-based objects and the stack grows in memory. When the function exits, the memory it claimed is released and made available to other functions. Thus, a software stack has a size that is dynamic and varies as the program is executed. The stack grows up in memory, toward larger addresses, when objects are allocated to the stack; it decreases in size when a function returns and its stack-based objects are no longer required.

A register, known as the stack pointer, is permanently assigned to hold the address of the "top" of the stack. MPLAB XC8 uses the FSR1 register as the stack pointer, and it holds the address of the next free location in the software stack. The register contents are increased when variables are allocated (pushed) to the stack and decreased when a function returns and variables are removed (popped) from the stack. There is no register assigned to hold a frame pointer. All access to the stack must use an offset to the stack pointer.

Note that if there are any functions in the program that use the software stack, the FSR1 register is reserved as the stack pointer for the duration of the entire program, even when executing functions that do not use the software stack. With this register unavailable for general use, the code generated may be less efficient or “Can’t generate code” errors may result.

The maximum size of the stack is not exactly known at compile time and the compiler typically reserves as much space as possible for the stack to grow during program execution. The stack is always allocated a single memory range, which may cross bank boundaries, but within this range it may be segregated into one area for main-line code and an area for each interrupt routine, if required. The maximum size of each area can be specified using the `-mstack` option (see 4.6.1.22 Stack Option). The stack pointer is reloaded when an interrupt occurs so it will access the separate stack area used by interrupt code. It is restored by the interrupt context switch code when the interrupt routine is complete.

The compiler cannot detect for overflow of the memory reserved for the stack as a whole, nor are any runtime checks made for stack overflow. If the software stack overflows, data corruption and code failure can result.

5.2.5 Configuration Bit Access

The `config` pragma allows the configuration bits (or fuses) and id-location registers to be specified in the assembly source file. Configuration bits, or fuses, are used to set up fundamental device operation, such as the oscillator mode, watchdog timer, programming mode and code protection. These bits must be correctly set to ensure your program executes correctly.

The pragma has the following forms:

```
#pragma config setting = value
#pragma config register = literal_value
```

Here, *`setting`* is a configuration setting descriptor, e.g., `WDT`, and *`value`* can be either a textual description of the desired state, e.g., `OFF`, or a numerical value. Either the *`setting`* or *`value`* tokens or the *`setting = value`* expression can be surrounded by either double or single quotes to protect them from any macro substitution performed by the preprocessor, for example:

```
#pragma config "WDT = ON"      // turn on watchdog timer
#pragma config WDTPS = 0x1A    // specify the timer postscale value
```

Some `value` tokens that appear to be purely numerical are in fact a textual description of the value, for example in the following:

```
config WDTPS = 32
```

the `32` token is a textual description for some devices that indicates a watchdog timer post-scale of 1:32, not the number 32. In such a case, it might not be the number 32 that is programmed into the relevant bits of the configuration register. The assembler will first check if *`value`* represents a predefined string, and only if that is not the case, assume it represents a numerical constant, which will then be subject to the same constraints as other numerical constant operands.

You should never assume that the `OFF` and `ON` tokens used in configuration macros equate to 0 and 1, respectively, as that is often not the case.

The *`register`* field is the name of a configuration or id-location register, and this must always be used with a *`value`* that is a numerical constant, for example:

```
#pragma config CONFIG1L = 0x8F
```

The available *`setting`*, *`register`* and *`value`* fields are documented in the chipinfo file relevant to your device (i.e. `pic_chipinfo.html` and `pic18_chipinfo.html`) and that are 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 `config` pragma can be used to set each configuration setting; alternatively, several comma-separated configuration settings can be specified by the same pragma. The pragma can be used as many times as required to fully configure the device.

The following example shows a configuration register being programmed as a whole and programmed using the individual settings contained within that register.

```
// PIC18F67K22
// VREG Sleep Enable bit : Enabled
// LF-INTOSC Low-power Enable bit : LF-INTOSC in High-power mode during Sleep
// SOSC Power Selection and mode Configuration bits : High Power SOSC circuit selected
// Extended Instruction Set : Enabled
#pragma config RETEN = ON, INTOSCSEL = HIGH, SOSCSEL = HIGH, XINST = ON

// Alternatively
#pragma config CONFIG1L = 0x5D

// IDLOC @ 0x200000
#pragma config IDLOC0 = 0x15
```

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

All the bits in the Configuration Words should be programmed to prevent erratic program behavior. Do not leave them in their default/unprogrammed state. Not all Configuration bits have a default state of logic high; some have a logic low default state. Consult your device data sheet for more information.

If you are using MPLAB X IDE, take advantage of its built-in tools to generate the required pragmas, so that you can copy and paste them into your source code. See the *MPLAB® X IDE User's Guide* for a description and use of the Configuration Bits window.

5.2.6 ID Locations

The 8-bit PIC devices have locations outside the addressable memory area that can be used for storing program information, such as an ID number. The `config` pragma is also used to place data into these locations by using a special register name. The pragma is used as follows:

```
#pragma config IDLOCx = value
```

where *`x`* is the number (position) of the ID location, and *`value`* is the nibble or byte that is to be positioned into that ID location. The value can only be specified in decimal or in hexadecimal, the latter radix indicated by the usual `0x` prefix. Values must never be specified in binary (i.e., using the `0b` prefix). If *`value`* is larger than the maximum value allowable for each location on the target device, the value will be truncated and a warning message is issued. The size of each ID location varies from device to device. See your device data sheet for more information. For example:

```
#pragma config IDLOC0 = 1
#pragma config IDLOC1 = 4
```

will attempt fill the first two ID locations with 1 and 4. One pragma can be used to program several locations by separating each register-value pair with a comma. For example, the above could also be specified as shown below.

```
#pragma config IDLOC0 = 1, IDLOC1 = 4
```

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

5.2.7 Using SFRs From C Code

The Special Function Registers (SFRs) are typically memory mapped registers and are accessed by absolute C structure variables that are placed at the register's address. These structures can be accessed in the usual way so that no special syntax is required to access SFRs.

The SFRs control aspects of the MCU and peripheral module operation. Some registers are read-only; some are write-only. Always check your device data sheet for complete information regarding the registers.

The SFR structures are predefined in header files and are accessible once you have included `<xc.h>` (see 5.2.3 Device Header Files) into your source files. Structures are mapped over the entire register and bit-fields within those structures allow access to specific SFR bits. The names of the structures will typically be the same as the corresponding register, as specified in the device data sheet, followed by `bits` (see 3.3.2.5 How Do I Find

The Names Used To Represent SFRs And Bits?). For example, the following shows code that includes the generic header file, clears `PORTA` as a whole and sets bit 2 of `PORTA` using the bit-field definition.

```
#include <xc.h>
int main(void)
{
    PORTA = 0x00;
    PORTAbits.RA2 = 1;
}
```

Care should be taken when accessing some SFRs from C code or from in-line assembly. Some registers are used by the compiler to hold intermediate values of calculations and writing to these registers directly can result in code failure. A list of registers used by the compiler and can be found in 5.6 Register Usage.

5.2.7.1 Special PIC18 Register Issues

Some of the SFRs used by PIC18 devices can be grouped to form multi-byte values, e.g., the TMRxH and TMRxL register combine to form a 16-bit timer count value. Depending on the device and mode of operation, there can be hardware requirements to read these registers in certain ways, e.g., often the TMRxL register must be read before trying to read the TMRxH register to obtain a valid 16-bit result.

It is not recommended that you read a multi-byte variable mapped over these registers as there is no guarantee of the order in which the bytes will be read. It is recommended that each byte of the SFR should be accessed directly, and in the required order, as dictated by the device data sheet. This results in a much higher degree of portability.

The following code copies the two timer registers into a C `unsigned` variable `count` for subsequent use.

```
count = TMR0L;
count += TMR0H << 8;
```

Macros are also provided to perform reading and writing of the more common timer registers. See the macros `READTIMERx` and `WRITETIMERx` in 9.1 Library Example Code. These guarantee the correct byte order is used.

5.2.8 Bit Instructions

Wherever possible, the MPLAB XC8 C Compiler will attempt to use bit instructions, even on non-bit integer values. For example, when using a bitwise operator and a mask to alter a bit within an integral type, the compiler will check the mask value to determine if a bit instruction can achieve the same functionality.

```
unsigned int foo;
foo |= 0x40;
```

will produce the instruction:

```
bsf _foo,6
```

To set or clear individual bits within integral type, the following macros could be used:

```
#define bitset(var, bitno)    ((var) |= 1UL << (bitno))
#define bitclr(var, bitno)    ((var) &= ~(1UL << (bitno)))
```

To perform the same operation on `foo` as above, the `bitset()` macro could be employed as follows:

```
bitset(foo, 6);
```

5.2.9 Multiplication

The PIC18 instruction set includes several 8-bit by 8-bit hardware multiple instructions, with these being used by the compiler in many situations. Non-PIC18 targets always use a library routine for multiplication operations.

There are three ways that 8x8-bit integer multiplication can be implemented by the compiler:

Hardware Multiply Instructions (HMI) These assembly instructions are the most efficient method of multiplication, but they are only available on PIC18 devices.

A bitwise iteration (8loop)	Where dedicated multiplication instructions are not available, this implementation produces the smallest amount of code – a loop cycles through the bit pattern in the operands and constructs the result bit-by-bit. The speed of this implementation varies and is dependent on the operand values; however, this is typically the slowest method of performing multiplication.
An unrolled bitwise sequence (8seq)	This implementation performs a sequence of instructions that is identical to the bitwise iteration (above), but the loop is unrolled. The generated code is larger, but execution is faster than the loop version.

Multiplication of operands larger than 8 bits can be performed one of the following two ways:

A bitwise iteration (xloop)	This is the same algorithm used by 8-bit multiplication (above) but the loop runs over all (x) bits of the operands. Like its 8-bit counterpart, this implementation produces the smallest amount of code but is typically the slowest method of performing multiplication.
A bytewise decomposition (bytdec)	This is a decomposition of the multiplication into a summation of many 8-bit multiplications. The 8-bit multiplications can then be performed using any of the methods described above. This decomposition is advantageous for PIC18 devices, which can then use hardware multiply instructions. For other devices, this method is still fast, but the code size can become impractical.

Multiplication of floating-point operands operates in a similar way – the integer mantissas can be multiplied using either a bitwise loop (xfploop) or by a bytewise decomposition.

The following tables indicate which of the multiplication methods are chosen by the compiler when performing multiplication of both integer and floating point operands. The method is dependent on the size of the operands, the type of optimizations enabled and the target device.

The table below shows the methods chosen when speed optimizations are enabled (see 4.6.6 Options for Controlling Optimization).

Table 5-1. Multiplication With Speed Optimizations

Device	8-bit	16-bit	24-bit	32-bit	24-bit FP	32-bit FP
PIC18	HMI	bytdec+HMI	bytdec+HMI	bytdec+HMI	bytdec+HMI	bytdec+HMI
Enhanced Mid-range	8seq	bytdec+8seq	bytdec+8seq	bytdec+8seq	bytdec+8seq	bytdec+8seq
Mid-range/ Baseline	8seq	16loop	24loop	32loop	24fploop	32fploop

The table below shows the method chosen when space optimizations are enabled or when no C-level optimizations are enabled.

Table 5-2. Multiplication With No Or Space Optimizations

Device	8-bit	16-bit	24-bit	32-bit	24-bit FP	32-bit FP
PIC18	HMI	bytdec+HMI	24loop	32loop	24fploop	32fploop
Enhanced Mid-range	8loop	bytdec+8loop	24loop	32loop	24fploop	32fploop
Mid-range/Baseline	8loop	16loop	24loop	32loop	24fploop	32fploop

The source code for the multiplication routines (documented with the algorithms employed) is available in the `pic/c99/sources` directory, located in the compiler's installation directory. Look for files whose name has the form `Umulx.c`. where `x` is the size of the operation in bits.

If your device and optimization settings dictate the use of a bitwise multiplication loop you can sometimes arrange the multiplication operands in your C code to improve the operation's speed. Where possible, ensure that the left operand to the multiplication is the smallest of the operands.

For example, in the code:

```
x = 10;
y = 200;
result = x * y;  // first multiply
result = y * x;  // second multiply
```

the variable result will be assigned the same value in both statements, but the first multiplication expression will be performed faster than the second.

5.2.10 Baseline PIC MCU Special Instructions

Baseline devices can use the OPTION and TRIS SFRs, which are not memory mapped.

The definition of these registers use a special qualifier, `__control`, to indicate that the registers are write-only, outside the normal address space and must be accessed using special instructions.

When these SFRs are written in C code, the compiler will use the appropriate instruction to store the value. For example, to set the TRIS register, the following code:

```
TRIS = 0xFF;
```

would be encoded by the compiler as:

```
movlw 0ffh
TRIS
```

Those Baseline PIC devices which have more than one output port can have `__control` definitions for objects: `TRISA`, `TRISB` and `TRISC`, which are used in the same manner as described above.

Any register that uses the `__control` qualifier must be accessed as a full byte. If you need to access bits within the register, copy the register to a temporary variable first, then access that temporary variable as required.

5.2.11 Oscillator Calibration Constants

Some Baseline and Mid-range devices have an oscillator calibration constant pre-programmed into their program memory. This constant can be read and written to the OSCCAL register to calibrate the internal RC oscillator.

On some Baseline PIC devices, the calibration constant is stored as a `movlw` instruction at the top of program memory, e.g., the PIC10F509 device. On Reset, the program counter is made to point to this instruction and it is executed first before the program counter wraps around to 0x0000, which is the effective Reset vector for the device. The default runtime startup routine (see 5.9.2 Runtime Startup Code) will automatically include code to load the OSCCAL register with the value contained in the W register after Reset on such devices. No other code is required.

For other chips, such as PIC12F629 device, the oscillator constant is also stored at the top of program memory, but as a `retlw` instruction. The compiler's startup code will automatically generate code to retrieve this value and perform the configuration. This value can also be read at runtime by calling `__osccal_val()`, whose prototype is provided in `<xc.h>`. For example:

```
unsigned char calVal;
calVal = __osccal_val();
```

Loading of the calibration value at startup can be turned off via the `-mno-osccal` option (see 4.6.1.15 Osccal Option).

Note: The location that stores the calibration constant is never code protected and will be lost if you reprogram the device. Thus, the calibration constant must be saved before the device is erased. The constant must then be reprogrammed at the same location along with the new program and data.
If you are using an in-circuit emulator (ICE), the location used by the calibration `retlw` instruction cannot be programmed and subsequent calls to `__osccal_val()` will not work. If you wish to test code that calls this function on an ICE, you must program a `retlw` instruction at the appropriate location. Remember to remove this instruction when programming the actual part so you do not destroy the calibration value.

Legacy projects can use the macro `_READ_OSCCAL_DATA()`, which maps to the `__osccal_val()` function.

5.2.12 MPLAB REAL ICE In-Circuit Emulator Support

The compiler supports log and trace functions (instrumented trace) when using a Microchip MPLAB REAL ICE In-Circuit Emulator. See the emulator's documentation for more information on the instrumented trace features.

Not all devices support instrumented trace and only native trace is currently supported by the compiler.

The log and trace macro calls need to be either added by hand to your source code or inserted by right-clicking on the appropriate location in MPLAB X IDE editor, as described by the emulator documentation. These macros should not be used in assembly code. The `<xc.h>` header must be included in any modules that use these macros.

The macros have the following form.

```
__TRACE(id);
__LOG(id, expression);
```

MPLAB X IDE will automatically substitute an appropriate value for *`id`* when you compile; however, you can specify these by hand if required. The trace *`id`* should be a constant in the range of 0x40 to 0x7F and the log *`id`* is a constant in the range of 0x0 to 0x7F. Each macro should be given a unique number so that it can be properly identified. The same number can be used for both trace and log macros.

Trace macros should be inserted in the C source code at the locations you wish to track. They will trigger information to be sent to the debugger and IDE when they are executed, recording that execution reached that location in the program.

The log expression can be any integer or 32-bit floating-point expression whose value will be recorded along with the program location. Typically, this expression is simply a variable name so the variable's contents are logged.

Adding trace and log macros will increase the size of your code as they contribute to the program image that is downloaded to the device.

Here is an example of these macros that you might add.

```
#include <xc.h>
inpStatus = readUser();
if(inpStatus == 0) {
    __TRACE(id);
    recovery();
}
__LOG(id, inpStatus);
```

5.2.13 Function profiling

The compiler can generate function registration code for the MPLAB REAL ICE In-Circuit Emulator to provide function profiling. To obtain profiling results, you must also use a Power Monitor Board and MPLAB X IDE and power monitor plugin that support code profiling for the MPLAB XC8 C Compiler.

The `-finstrument-functions` option (see 4.6.5.1 Instrument Functions Option) enables this feature and inserts assembly code into the prologue and epilogue of each function. This code communicates runtime information to the debugger to signal when a function is being entered and when it exits. This information, along with further measurements made by a Microchip Power Monitor Board, can determine how much energy each function is using. This feature is transparent, but note the following points when profiling is enabled:

- The program will increase in size and run slower due to the profiling code
- One extra level of hardware stack is used

- Some additional RAM memory is consumed
- Inlining of functions will not take place for any profiled function

If a function cannot be profiled (due to hardware stack constraints) but is qualified inline, the compiler might inline the function. See 5.7.1.2 Inline Specifier for more information on inlining functions.

5.2.14 Code Coverage

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

This feature is available in the compiler for all enhanced Mid-range and PIC18 devices.

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.

This feature differs from function profiling (Function Profiling) in that code coverage indicates program execution of smaller blocks of code, as opposed to execution of a function, and is primarily used as part of a testing regime to ensure that all parts of a program have been executed and hence tested. It does not verify that code executed correctly nor provide any indication of code execution times, which can be determined using the function profiling feature.

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

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

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

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

There is one assembly instruction inserted into each instrumented basic block. Other instructions might be required to perform bank selection, depending on where the code coverage bits are located and which target device is being used. Note that some optimizations are disabled when enabling the code coverage feature, and this will increase the size of the built project.

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 www.microchip.com/Developmenttools/ProductDetails/SW006026-COV for information on how to use the new code coverage features).

5.2.15 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.

5.2.15.1 Stack Guidance Information

The stack guidance features estimates the stack usage of several stacks which are used by programs compiled with the MPLAB XC8 C Compiler, those being:

- The hardware stack, used for function return addresses
- The software data stack, used by any function compiled using a reentrant model

The hardware stack is described in 5.2.4.1 Function Return Address Stack. The size of this stack is fixed by the device. Nesting function calls too deeply will overflow the stack and ultimately cause program failure. Interrupts also use this stack. With some devices, the `-mstackcall` option can be used to employ a different function call method that reduces stack usage.

Functions are encoded by the compiler to use either a software or compiled stack for data objects they define, as described in 5.2.4.2 Data Stacks. A function will only ever use one stack type, but a program can be comprised of functions that use a compiled stack and functions that use the software stack. Typically, a program will follow a stack model (`-mstack` option) and have all functions compiled to use the same stacks.

A compiled stack is a static allocation of data memory for stack-based objects. As the size of this stack can be accurately determined at compile time and cannot change at runtime, the memory errors issued by the compiler should this stack become too large will alert you to this situation. The stack guidance feature will not issue any additional information for the compiled stack used by a program, other than to indicate its use.

By comparison, a software stack is a dynamic allocation of memory that occurs at runtime and its size is more difficult to predict. This is where the information produced by the stack guidance feature will be useful. The stack guidance feature will alert you to potential runtime overflow of this stack.

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

```
=============================== STACK USAGE GUIDANCE ================================

The program uses a compiled stack model.

The call-graph beginning at main(),
   uses an estimated 17 of 31 hardware stack locations and a compiled stack

However, the following cautions exist:

1) There is 1 "high-priority" interrupt service routine detected:
   isr() call-graph uses an estimated 1 hardware stack location and a compiled stack
You must add stack allowances for those functions.

2) Current optimization level might use a level of hardware stack.
You must add a hardware stack allowance for this feature.

=====================================================================================
```

The information will show the following.

- The stack model used by the program
- The hardware stack usage estimate
- The software stack usage estimate

In addition, a number of cautionary messages might be displayed. These clarify potential stack overflows reported by the above estimates and indicate additional stack usage that must be considered to determine the total 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 program. As the number of iterations of a recursive call cannot be predicted, the number of hardware stack levels and 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.
External functions	Calls to external functions were detected in the program. External functions are those not defined in the C program. Assembly routines, for example, are considered as external functions. External functions cannot be scanned for the number of hardware stack levels they consume, nor for any data they place on a software stack. If external functions are called, no guidance is possible. The name of any external functions called are listed in this caution.
Deviations from the current stack model	Although the compiler will build programs using a stack model (specified by the `-mstack` option), which dictates the stack type used by all functions, there are instances where a function might use a different stack type, e.g., if that function used a stack specifier. This caution lists those functions using a different stack type to that specified by the model.
Main-line code exceeds hardware stack size	This caution indicates that the function call depth of the main-line code call-graph has exceeded the number of hardware stack levels available on the target device. If the option is supported by your target device, you may use `-mstackcall` (see 4.6.1.23 Stackcall Option) to have the compiler use an alternate method of calling functions that does not need as many stack levels. The reduced stack usage of this method will then be reflected in the stack guidance report printed by the compiler.
Interrupts might exceed hardware stack size	Even though the depth of the call graphs associated with main-line and interrupt code might be known, interrupts can trigger at any time, and in such a situation, the compiler cannot reliably determine the program's total function call depth. 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 hardware stack usage of your program.
Call graph exceeds software stack size	The stack guidance feature has estimated that the indicated call graph (main-line or interrupt) has or might have used more software data stack than it was allocated. The `-mstack` option can be used to change the size allocated to each call graph, see 4.6.1.22 Stack Option.
Interrupts might exceed software stack size	Even though the size of the software 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.
Optimization level might use hardware stack	Some optimizations might result in additional calls being made that will increase the usage of the hardware stack over what has been reported. This caution indicates that the current optimization settings could lead to increased hardware stack usage and must be taken into account when considering the hardware stack usage of your program.
Debugger might use hardware stack	The debug executives used by some debuggers consume levels of the device's hardware stack. In this caution, the indicated number of levels required by the debugger must be taken into account when considering the hardware stack usage of your program.

5.3 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.

5.3.1 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`.

Up to 255 characters are significant in an identifier. If two identifiers differ only after the maximum number of significant characters, then the compiler will consider them to be the same symbol.

5.3.2 Integer Data Types

The MPLAB XC8 compiler supports integer data types with 1, 2, 3, 4 and 8 byte widths as well as a single bit type. The table below shows the data types and their corresponding size and arithmetic type. The default type for each type group is bolded.

Table 5-3. Integer Data Types

Type	Size (bits)	Arithmetic Type
`__bit`	1	Unsigned integer
`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
`__int24`	24	Signed integer
`__uint24`	24	Unsigned integer
`signed long`	32	Signed integer
`unsigned long`	32	Unsigned integer
`signed long long`	32/64	Signed integer
`unsigned long long`	32/64	Unsigned integer

The `__bit` and `__int24` types are non-standard types available in this implementation. The `long long` types are 64-bit C99 standard types when building for PIC18 or Enhanced Mid-range devices, but this implementation limits their size to only 32 bits for projects conforming to the C90 Standard or any project targeting any other device.

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

If no signedness is specified in the type, then the type will be `signed` except for the `char` and `__bit` types which are always `unsigned`. The concept of a signed bit is meaningless.

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. The macros associated with the `__int24` type are non-standard macros available in this implementation. The values associated with the `long long` macros are dependent on the C standard being used.

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

5.3.2.1 Bit Data Types And Variables

The MPLAB XC8 C Compiler supports a single-bit integer type via the `__bit` type specifier.

Bit variables behave in most respects like normal `unsigned char` variables, but they can only contain the values 0 and 1. They provide a convenient and efficient method of storing flags, since eight bit objects are packed into each byte of memory storage. Operations on `bit` variables are performed using the single bit instructions (`bsf` and `bcf`) wherever possible.

```
__bit init_flag;
```

These variables cannot be auto or parameters to a function, but can be qualified `static`, allowing them to be defined locally within a function. For example:

```
int func(void) {
    static __bit flame_on;
    // ...
}
```

A function can return a `bit` by using the `__bit` keyword in the function's prototype in the usual way. The returned value will be stored in the STATUS register carry flag.

It is not possible to declare a pointer to `bit` types or assign the address of a `bit` object to any pointer. Nor is it possible to statically initialize `bit` variables so they must be assigned any non-zero starting value (i.e., 1) in the code itself. Objects qualified `bit` will be cleared on startup, unless the object is qualified `__persistent`.

When assigning a larger integral type to a `bit` variable, only the least significant bit is used. For example, in the following code:

```
int data = 0x54;
__bit bitvar;
bitvar = data;
```

`bitvar` will be cleared by the assignment since the LSb of data is zero. This sets the `__bit` type apart from `_Bool`, which is a boolean type (See 5.3.3 Boolean Types).

All addresses assigned to `bit` objects and the sections that hold them will be bit addresses. For absolute `bit` variables (see 5.4.4 Absolute Variables), the address specified in code must be a bit address. Take care when comparing these addresses to byte addresses used by all other variables.

5.3.3 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 `__bit` integer types. Values converted to a `_Bool` type result in 0 (false) if the value is 0; otherwise, they result in 1 (true). Values converted to an integer `__bit` type are truncated to the least significant bit.

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;
```

If you are compiling with the C90 standard, `_Bool` is not available, but there is a `bool` type available if you include `<stdbool.h>`, but which is merely a `typedef` for `unsigned char`.

5.3.4 Floating-Point Data Types

The MPLAB XC8 compiler supports 32- and 24-bit floating-point types, being an IEEE 754 32-bit format, or a truncated, 24-bit form of this, respectively. Floating-point sizes of 32-bits will be automatically set when you select C99 compliance. If 24-bit floating-point types are explicitly selected, the compiler will use the C90 libraries. The table below shows the data types and their corresponding size and arithmetic type.

Table 5-4. Floating-point Data Types

Type	Size (bits)	Arithmetic Type
`float`	24 / 32	Real

...........continued		
Type	**Size (bits)**	**Arithmetic Type**
`double`	24 / 32	Real
`long double`	same size as `double`	Real

For both `float` and `double` values, the 24-bit format is the default. The options `-fshort-float` and `-fshort-double` can also be used to specify this explicitly. The 32-bit format is used for `double` values if `-fno-short-double` option is used and for `float` values if `-fno-short-float` is used.

Variables can be declared using the `float` and `double` keywords, respectively, to hold values of these types. Floating-point types are always signed and the `unsigned` keyword is illegal when specifying a floating-point type. Types declared as `long double` will use the same format as types declared as `double`. All floating-point values are represented in little-endian format with the LSB at the lower address.

The 32-bit floating-point type supports “relaxed” semantics when compared to the full IEEE implementation, which means the following rules are observed.

Tiny (subnormal) arguments to floating-point routines are interpreted as zeros. There are no representable floating-point values possible between -1.17549435E-38 and 1.17549435E-38, except for 0.0. This range is called the denormal range. Subnormal results of routines are flushed to zero. There are no negative 0 results produced.

Not-a-number (NaN) arguments to routines are interpreted as infinities. NaN results are never created in addition, subtraction, multiplication, or division routines where a NaN would be normally expected—an infinity of the proper sign is created instead. The square root of a negative number will return the “distinguished” NaN (default NaN used for error return).

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

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

- *sign* is the sign bit, which indicates if 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 the mantissa, which is 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 5-5. Floating-point Formats

Format	Sign	Biased Exponent	Mantissa
IEEE 754 32-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
modified IEEE 754 24-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx

Here are some examples of the IEEE 754 32-bit formats 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 5-6. 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)	—
24-bit	42123Ah	10000100b	1.001001000111010b	36.557
		(132)	(1.142395019531)	—

Use the following process to manually calculate the 32-bit example in the above table.

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

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

which is approximately equal to:

2.77000e+37

Binary floating-point values are sometimes misunderstood. It is important to remember that not every floating-point value can be represented by a finite sized floating-point number. 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. Thus the 24-bit format allows for values with approximately the same range of values representable by the 32-bit format, but the values that can be exactly represented by this format are more widely spaced.

For example, if you are using a 24-bit wide floating-point type, it can exactly store the value 95000.0. However, the next highest number it can represent is 95002.0 and it is impossible to represent any value in between these two in such a type as it will be rounded. This implies that C code which compares floating-point values might not behave as expected.

For example:

```
volatile float myFloat;
myFloat = 95002.0;
if(myFloat == 95001.0)       // value will be rounded
    PORTA++;                 // this line will be executed!
```

in which the result of the `if` expression will be true, even though it appears the two values being compared are different.

Compare this to a 32-bit floating-point type, which has a higher precision. It also can exactly store 95000.0 as a value. 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.

5.3.5 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.

The members of structures and unions cannot be objects of type `__bit`.

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

5.3.5.1 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 };  // record place in program memory
```

In this case, the entire structure will be placed into the program space and each member will be read-only. Remember that all members are usually initialized if a structure is `const` as they cannot be initialized at runtime.

If the members of the structure were individually qualified `const`, but the structure was not, then the structure would be positioned into RAM, but each member would be read-only. Compare the following structure with the above.

```
struct {
    const int number;
    int * const ptr;
} record = { 0x55, &i };  // record placed in data memory
```

5.3.5.2 Bit-fields In Structures

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

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

Bit-fields can never cross the boundary between 8-bit allocation units. Bit-fields of type `_Bool` are also supported; however, they can only be one bit in size.

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.

5.3.5.3 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.

Objects defined with anonymous structures or unions can only be initialized if you are using the C99 Standard.

5.3.6 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.

The MPLAB XC8 compiler records all assignments of addresses to each pointer the program contains, and as a result, non-standard qualifiers are not required when defining pointer variables. The standard qualifiers `const` and `volatile` can still be used and have their usual meaning.

The size and format of the address held by each pointer is based on the set of all possible targets the pointer can address. This information is specific to each pointer defined in the program, thus two pointers with the same C type can hold addresses of different sizes and formats due to the way the pointers were used in the program.

The compiler tracks the memory location of all targets, as well as the size of all targets to determine the size and scope of a pointer. The size of a target is important as well, particularly with arrays or structures. It must be possible to increment a pointer so it can access all the elements of an array, for example.

5.3.6.1 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.

5.3.6.2 Pointer-target Qualifiers

The `__rom` and `__ram` pointer-target qualifiers can be used if you would like the compiler to confirm that targets assigned to a pointer are in a particular memory space.

These qualifiers can be used only when declaring or defining pointers. They cannot be used with ordinary variables and they have no effect on the placement of the pointers themselves. These qualifiers are always enforced by the compiler and they are not affected by the `-maddrqual` option (see 4.6.1.1 Addrqual Option) or `#pragma addrqual`.

The assignment of an incompatible target to a pointer that uses one of these qualifiers will trigger an error, so in the following example:

```
const int __rom * in_ptr;
```

an error would be generated if your program assigned to `in_ptr` the address of an object that was in data memory. Use of `__rom` implies the `const` qualifier, but it is recommended that `const` is explicitly used to ensure the meaning of your code is clear.

The use of these qualifiers must be consistent across all declarations of a pointer and it is illegal to use both qualifiers with the same pointer variable.

5.3.6.3 Data Pointers

There are several pointer classifications used with the MPLAB XC8 C Compiler, such as those indicated below. Those classification marked with (local) are the only classifications considered when local OCG optimizations have been selected (see 5.3.6.3.2 Pointer Classifications with Local Optimization).

For Baseline and Mid-range devices:

- 8-bit pointer capable of accessing common memory and two consecutive (even-odd) banks, e.g., banks 0 and 1, or banks 6 and 7, etc.
- 16-bit pointer capable of accessing the entire data memory space (local)
- 8-bit pointer capable of accessing up to 256 bytes of program space data
- 16-bit pointer capable of accessing up to 64 Kbytes of program space data (local)
- 16-bit mixed target space pointer capable of accessing the entire data space memory and up to 64 Kbytes of program space data (local)

For PIC18 devices:

- 8-bit pointer capable of accessing the access bank
- 16-bit pointer capable of accessing the entire data memory space (local)
- 8-bit pointer capable of accessing up to 256 bytes of program space data
- 16-bit pointer capable of accessing up to 64 Kbytes of program space data (local)
- 24-bit pointer capable of accessing the entire program space (local)
- 16-bit mixed target space pointer capable of accessing the entire data space memory and up to 64 Kbytes of program space data
- 24-bit mixed target space pointer capable of accessing the entire data space memory and the entire program space (local)

Each data pointer will be allocated one of the available classifications after preliminary scans of the source code. There is no mechanism by which the programmer can specify the style of pointer required (other than by the assignments to the pointer). The C code must convey the required information to the compiler.

Information about the pointers and their targets are shown in the pointer reference graph (described in 6.3.5 Pointer Reference Graph). This graph is printed in the assembly list file.

5.3.6.3.1 Pointers to Both Memory Spaces

When a data pointer is assigned the address of one or more objects that have been allocated memory in the data space and also assigned the address of one or more objects that have been allocated memory in the program memory space, the pointer is said to have targets with mixed memory spaces. Such pointers fall into one of the mixed target space pointer classifications (listed in 5.3.6.3 Data Pointers) and the address will be encoded so that the target memory space can be determined at runtime. The encoding of these pointer types are as follows.

For the Baseline/Mid-range 16-bit mixed target space pointer, the MSb of the address (i.e., bit number 15) indicates the memory space that the address references. If this bit is set, it indicates that the address is of something in program memory; clear indicates an object in the data memory. The remainder of this address represents the full address in the indicated memory space.

For the PIC18 16-bit mixed target space pointer, any address above the highest data space address is that of an object in the program space memory; otherwise, the address is of a data space memory object.

For the PIC18 24-bit mixed target space pointer, bit number 21 indicates the memory space that the address references. If this bit is set, it indicates that the address is of an object residing in data memory; if it is clear, it indicates an object in the program memory. The remainder of this address represents the full address in the indicated memory space. Note that for efficiency reasons, the meaning of the memory space bit is the opposite to that for Baseline and Mid-range devices.

If assembly code references a C pointer, the compiler will force that pointer to become a 16-bit mixed target space pointer, in the case of Baseline or Mid-range programs, or a 24-bit mixed target space pointer, for PIC18 programs. These pointer types have unrestricted access to all memory areas and will operate correctly, even if assignments (of a correctly formatted address) are made to the pointer in the assembly code.

5.3.6.3.2 Pointer Classifications with Local Optimization

Where local optimizations have been enabled, pointers can have a size and classification based purely on their definition, not on the targets assigned to them by the program.

The pointer-target specifiers, `__ram` and `__rom`, (see 5.3.6.2 Pointer-target Qualifiers) can be used to ensure that addresses assigned to a pointer during the program's execution are within an intended memory space. Together with a restricted range of pointer classifications, this ensures that pointers will have a size that is predictable.

Pointers defined in code built with local optimizations and which have the indicated targets will have the following sizes, where type is a valid, unqualified C type.

`const` ***`type *`*** can be assigned the address of any object in data or program memory. These pointers will be 3 bytes wide if the program is being built for a PIC18 device that has more than 64kB of program memory; they will be 2 bytes wide, otherwise.

`__rom` ***`type *`*** can be assigned the address of any object in program memory, and attempts to assign the address of a data memory object will result in an error. These pointers will be 3 bytes wide if the program is being built for a PIC18 device that has more than 64kB of program memory; they will be 2 bytes wide, otherwise. The `const` specifier can be optionally used with `__rom` without changing the pointer size.

`type *` can be assigned the address of any object in data memory. As per normal operation, the compiler will issue a warning if the address of a `const` object is assigned to such pointers. These pointers are always 2 bytes in size.

`__ram` ***`type *`*** can be assigned the address of any object in data memory and attempts to assign the address of a program memory object will result in an error. These pointers will always be 2 bytes in size. The presence of the `const` specifier indicates only that the target objects must be treated as read-only. The `const` specifier can be optionally used with `__ram` without changing the pointer size.

The size and operation of pointers to `__far`, pointers to `__eeprom`, and function pointers are not affected by the local optimization setting.

5.3.6.4 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.

For Baseline and Mid-range devices, function pointers are always one byte in size and hold an offset into a jump table that is output by the compiler. This jump table contains jumps to the destination functions.

For Enhanced Mid-range devices, function pointers are always 16-bits wide and can hold the full address of any function.

For PIC18 devices, function pointers are either 16 or 24 bits wide. The pointer size is purely based on the amount of program memory available on the target device.

As with data pointers, the target assigned to function pointers is tracked. This is an easier process to undertake compared to that associated with data pointers as all function instructions must reside in program memory. The pointer reference graph (described in 6.3.5 Pointer Reference Graph) will show function pointers, in addition to data pointers, as well as all their targets. The targets will be names of functions that could possibly be called via the pointer.

One notable runtime feature for Baseline and Mid-range devices is that a function pointer which contains null (the value 0) and is used to call a function indirectly will cause the code to become stuck in a loop which branches to itself. This endless loop can be used to detect this erroneous situation. Typically calling a function via a null function would result in the code crashing or some other unexpected behavior. The label to which the endless loop will jump is called `fpbase`.

5.3.6.5 Special Pointer Targets

Pointers and integers are not interchangeable. Assigning an integer to a pointer will generate a warning to this effect. For example:

```
const char * cp = 0x123;  // the compiler will flag this as bad code
```

There is no information in the integer constant, 0x123, relating to the type, size or memory location of the destination. There is a very good chance of code failure when dereferencing pointers that have been assigned integer addresses, particularly for PIC devices that have more than one memory space.

Always take the address of a C object when assigning an address to a pointer. If there is no C object defined at the destination address, then define or declare an object at this address which can be used for this purpose. Make sure the size of the object matches the range of the memory locations that are to be accessed by the pointer.

For example, a checksum for 1000 memory locations starting at address 0x900 in program memory is to be generated. A pointer is used to read this data. You can be tempted to write code such as:

```
const char * cp;
cp = 0x900;  // what resides at 0x900???
```

and increment the pointer over the data.

However, a much better solution is this:

```
const char * cp;
extern const char inputData[1000] __at(0x900);
cp = inputData;
// cp is incremented over inputData and used to read values there
```

In this case, the compiler can determine the size of the target and the memory space. The array size and type indicates the size of the pointer target, the `const` qualifier on the object (not the pointer) indicates the target is located in program memory space. Note that the `const` array does not need initial values to be specified in this instance, see 5.3.8.1 Const Type Qualifier and can reside over the top of other objects at these addresses.

If the pointer has to access objects in data memory, you need to define a different object to act as a dummy target. For example, if the checksum was to be calculated over 10 bytes starting at address 0x90 in data memory, the following code could be used.

```
const char * cp;
extern char inputData[10] __at(0x90);
cp = inputData;
// cp is incremented over inputData and used to read values there
```

No memory is consumed by the `extern` declaration and this can be mapped over the top of existing objects.

User-defined absolute objects will not be cleared by the runtime startup code and can be placed over the top of other absolute variables.

Take care when comparing (subtracting) pointers. For example:

```
if(cp1 == cp2)
    ; // take appropriate action
```

The C standard only allows pointer comparisons when the two pointers' addresses are of the same object. One exception is that the address can extend to one element past the end of an array.

Never compare pointers with integer constants as that is even more risky, for example:

```
if(cp1 == 0x246)
    ; // take appropriate action
```

A null pointer is the one instance where a constant value can be assigned to a pointer and this is handled correctly by the compiler. A null pointer is numerically equal to 0 (zero), but this is a special case imposed by the C standard. Comparisons with the macro `NULL` are also allowed.

5.3.7 Constant Types and Formats

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

5.3.7.1 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 5-7. 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 5-8. 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;
```

5.3.7.2 Floating-point Constant

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

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

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

5.3.7.3 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 5.3.6.1 Combining Type Qualifiers And Pointers and qualify the pointer as follows.

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

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

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

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

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

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

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

5.3.8 Standard Type Qualifiers

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

5.3.8.1 Const Type Qualifier

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

Auto and parameter objects qualified const are treated differently, but all other objects declared using const are placed in a special psect that is linked into the program memory space. These objects can also be made absolute. The __at(*address*) construct is used to place the object at the specified address in program memory, as in the following example which places the object tableDef at address 0x100.

```
const int tableDef[] __at(0x100) = { 0, 1, 2, 3, 4};
```

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. However, uninitialized absolute extern const objects can be defined and are useful if you need to place an object in program memory over the top of other objects at a particular location, as in the following example.

```
extern const char checksumRange[0x100] __at(0x800);
```

will define `checksumRange` as an array of 0x100 characters located at address 0x800 in program memory. This definition will not place any data in the HEX file.

5.3.8.2 Volatile Type Qualifier

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

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

```
#include <xc.h>
volatile static unsigned int  TACTL __at(0x160);
```

The `volatile` qualifier does not guarantee that the object will be accessed atomically. Because the 8-bit PIC MCU architecture can only access a maximum of 1 byte of data per instruction, reading and writing most objects requires more than one instruction to complete.

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

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

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

Some variables are treated as being `volatile` even though they are not qualified. See 5.11.3.6 Undefined Symbols if you have assembly code in your project.

5.3.9 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.

5.3.9.1 Bank Type Qualifier

The `__bank(n)` type qualifier and the `-maddrqual` compiler option are used to place objects in a particular memory bank.

The data memory on PIC devices is arranged into memory banks. The compiler automatically allocates objects to one of the available banks, but there are times when you might require the object to be located in a particular bank, as might be the case if assembly code selects that bank prior accessing the object. They can be used to place objects in banks 0 through 3 (higher bank selection is not currently available).

This qualifier can be used with any variable with static storage duration, for example to following places `playMode` into bank 1:

```
__bank(1) unsigned char playMode;
```

These qualifiers are controlled by the compiler option `-maddrqual`, which determines their effect (see 4.6.1.1 Addrqual Option). Based on this option's settings, these qualifiers can be binding or ignored (which is the default operation). Qualifiers which are ignored will not produce an error or warning, but will have no effect.

5.3.9.2 EEPROM Type Qualifier

The `__eeprom` type qualifier is used to place objects in EEPROM memory for Baseline and Mid-range devices that implement this memory. A warning is produced if the qualifier is not supported for the selected device. Check your device data sheet to see the memory available.

This qualifier can be used with any object with static storage duration, for example to place the `inputData` array into EEPROM, use:

```
__eeprom unsigned char inputData[3];
```

See 5.4.5 Variables in EEPROM for other ways of accessing the EEPROM.

5.3.9.3 Far Type Qualifier

The `__far` type qualifier and the `-maddrqual` compiler option are used to place variables into external memory.

Some PIC18 devices can support external memory. If your hardware supports this, you must first specify this memory with the `-mram` option (see 4.6.1.17 Ram Option). For example, to map additional data memory from 20000h to 2FFFFh use

```
-mram=default,+20000-2FFFF.
```

Memory added to the RAM ranges is exclusively used by variables that are qualified `__far`. Access of external memory is less efficient than that of ordinary data memory and will be slower to execute and use more code. Here is an example of an `unsigned int` object placed into the device's external program memory space:

```
__far unsigned int farvar;
```

This qualifier is controlled by the compiler option `-maddrqual`, which determines its effect on PIC18 devices (see 4.6.1.1 Addrqual Option). Based on this option's settings, this qualifier can be binding or ignored (which is the default operation). Qualifiers which are ignored will not produce an error or warning, but will have no effect.

Note that this qualifier will be ignored when compiling for PIC10/12/16 targets and that not all PIC18 devices support external memory.

5.3.9.4 Near Type Qualifier

The `__near` type qualifier and the `-maddrqual` compiler option are used to place variables in common memory.

Some of the 8-bit PIC architectures implement data memory which can be always accessed regardless of the currently selected bank. This common memory can be used to reduce code size and execution times as the bank selection instructions that are normally required to access data in banked memory are not required when accessing the common memory. PIC18 devices refer to this memory as the access bank memory. Mid-range and Baseline devices have very small amounts of this memory, if it is present at all. PIC18 devices have substantially more common memory, but the amount differs between devices. See your device data sheet for more information.

This qualifier can be used with any variable with static storage duration, for example:

```
__near unsigned char fred;
```

This qualifier is controlled by the compiler option `-maddrqual`, which determines its effect (see 4.6.1.1 Addrqual Option). Based on this option's settings, this qualifier can be binding or ignored (which is the default operation). Qualifiers which are ignored will not produce an error or warning, but will have no effect.

The compiler must use common memory for some temporary objects. Any remaining space is available for general use. The compiler automatically places frequently accessed user-defined objects in common memory, so this qualifier is only needed for special memory placement of objects.

5.3.9.5 Persistent Type Qualifier

The `__persistent` type qualifier is used to indicate that variables should not be cleared by the runtime startup code by having them stored in a different area of memory to other variables.

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

For example, the following ensures that the static local object, `intvar`, is not cleared at startup:

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

```
// ...
}
```

5.3.9.6 Ram And Rom Pointer-target Qualifiers

The `__ram` and `__rom` qualifiers ensure that pointers access targets only in a desired memory space. They do not affect the placement of pointers with which they are used. See 5.3.6.3 Data Pointers for more information.

5.3.9.7 Section Qualifier

The `__section()` qualifier allocates the object to a user-nominated section rather than allowing the compiler to place it in a default section. See 5.14.3 Changing and Linking the Allocated Section for full information on the use of this qualifier.

5.4 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.

5.4.1 Address Spaces

All 8-bit PIC devices have a Harvard architecture, which has a separate data memory (RAM) and program memory space. Some devices also implement EEPROM.

The data memory (referred to in the data sheets as the general purpose register file) is banked to reduce the assembly instruction width. A bank is “selected” by one or more instructions that sets one or more bits in an SFR. Consult your device data sheet for the exact operation of the device you are using.

Both the general purpose RAM and SFRs both share the same data space and can appear in all available memory banks. PIC18 devices have all SFRs in the one data bank, but Mid-range and Baseline devices have SFRs at the lower addresses of each bank. Due to the location of SFRs in these devices, the general purpose memory becomes fragmented and this limits the size of most C objects.

The Enhanced Mid-range devices overcome this fragmentation by allowing a linear addressing mode, which allows the general purpose memory to be accessed as one contiguous chunk. Thus, when compiling for these devices, the maximum allowable size of objects typically increases. Objects defined when using PIC18 devices can also typically use the entire data memory.

Many devices have several bytes which can be accessed regardless of which bank is currently selected. This memory is called common memory. The PIC18 data sheets refer to the bank in which this memory is stored as the access bank, and hence it is often referred to as the access bank memory. Since no code is required to select a bank before accessing these locations, access to objects in this memory is typically faster and produces smaller code. The compiler always tries to use this memory if possible.

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.

5.4.2 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.

Note that in addition to the situations described in this section, some optimizations (see 4.6.6.8 Cacheconst Option) attempt to cache `const`-qualified objects that otherwise would be allocated to program memory into data memory, where they are accessed more efficiently.

5.4.2.1 Static Storage Duration Objects

Objects which are allocated a memory location that remains fixed for the duration of the program (i.e. not allocated space on a stack) are said to have static storage duration and are located by the compiler into any of the available data banks.

Allocation is performed in two steps. The code generator places each object into a specific section, then the linker places these sections into their predetermined memory areas. See 5.14.1 Compiler-Generated Psects for an introductory guide to sections. Thus, during compilation, the code generator can determine the bank in which each object will reside, so that it can efficiently handle bank selection, but it will not know the object's exact address.

The compiler considers three broad categories of object, which relate to the value the object should contain at the time the program begins. Each object category has a corresponding family of sections (see 5.14.1 Compiler-Generated Psects), which are tabulated below.

`nv` These sections are used to store objects qualified `__persistent`, whose values are not cleared by the runtime startup code.

`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.

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

5.4.2.1.1 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.

5.4.2.1.2 Object Size Limits

An object with static storage duration cannot be made larger than the available device memory size, but there can be other restrictions as to how large each object can be.

When compiling for Enhanced Mid-range PIC devices, the size of an object is typically limited only by the total available data memory. Objects that will not fit into any one of the available data banks will be allocated across several banks and accessed using the device's linear data memory feature. Linear memory access is typically slower than accessing the object directly.

When compiling for PIC18 devices, the size of an object is also typically limited only by the data memory available. The instruction set allows any object to span several data banks; however, the code to access such objects will typically be larger and slower.

On Baseline and other Mid-range devices, the object must entirely fit in one bank and so objects are limited in size to the largest of the available spaces in the data banks.

5.4.2.1.3 Changing the Default Allocation

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

- Reserving memory locations
- Using specifiers
- Making the objects absolute; or
- Placing objects in their own section and linking that section

If you wish to prevent objects from using one or more data memory locations so that these locations can be used for some other purpose, you are best reserving the memory using the memory adjust options. See 4.6.1.18 Reserve Option for information on how to do this.

Objects can be placed in specific memory banks using the `__bank()` specifier (see 5.3.9.1 Bank Type 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 5.4.4 Absolute Variables). Since absolute objects have a known address, they do not follow the normal memory allocation procedure.

Objects can also be placed in their own section by using the `__section()` specifier, allowing this section to be linked at the required location (see 5.14.3 Changing and Linking the Allocated Section).

5.4.2.2 Automatic Storage Duration Objects

Objects with automatic storage duration, such as auto, parameter objects, and temporary variables, are typically allocated space on a stack implemented by the compiler.

MPLAB XC8 has two stack implementations: a compiled stack and a software stack (described in 5.2.4.2 Data Stacks). Each C function is compiled to use exactly one of these stacks for its automatic storage duration objects and the table below summarizes how the choice of stack affects a function's reentrancy.

Table 5-9. Function Models Implementation

Data Stack Used	Function Model	Supported Device Families
Compiled stack	Non-reentrant	All devices
Software stack	Reentrant	Enhanced Mid-range and PIC18 devices

When compiling for those devices that do not support the reentrant function model, all functions are encoded to use the compiled stack, which are non-reentrant functions.

For the Enhanced Mid-range and PIC18 devices, by default the compiler will use the non-reentrant model for all functions. The `-mstack` option (see 4.6.1.22 Stack Option) can be used to change the compiler's default behavior when assigning function models. Select the `software` argument with this option so that the compiler will always choose the reentrant model (software stack) for each function. Set this option to `hybrid` to allow the compiler to decide how each function should be encoded. If the function is not reentrantly called, then it will be encoded to use the non-reentrant model and the compiled stack. If the function appears in more than one call graph (i.e., it is called from main-line and interrupt code), or it appears in a loop in a call graph (i.e., it is called recursively), then the compiler will use the reentrant model. The hybrid mode allows the program to use recursion but still take advantage of the more efficient compiled stack.

Alternatively you can change the function model for individual functions by using function specifiers when you define the function. Use either the `__compiled` or `__nonreentrant` specifier (identical meanings) to indicate that the specified function must use the compiled stack, without affecting any other function. Alternatively, use either the `__software` or `__reentrant` specifier to indicate a function must be encoded to use the software stack.

The function specifiers have precedence over the `-mstack` option setting. If, for example, the option `-mstack=compiled` has been used, but one function uses the `__software` (or `__reentrant`) specifier, then the specified function will use the software stack and all the remaining functions will use the compiled stack. These functions specifiers also override any choice made by the compiler in hybrid mode.

If the `-mstack=compiled` option has been issued or a function has been specified as `__compiled` (or `__nonreentrant`) and that function appears in more than one call graph in the program, then a function duplication feature automatically comes into effect (see 5.8.7 Function Duplication). Duplicating a non-reentrant function allows it to be called from multiple call graphs, but cannot allow the function to be called recursively.

The auto objects defined in a function will not necessarily be allocated memory in the order declared, in contrast to parameters which are always allocated memory based on their lexical order. In fact, auto objects for one function can be allocated in many RAM banks.

The standard `const` qualifier can be used with auto objects and forces them to be read-only. These objects, however, might not be stored on the stack. See 5.1.2.4 Const Auto Objects for how the compiler allocates such objects.

5.4.2.2.1 Object Size Limits

The compiled stack can be built up in more than one block, each located in a different data bank, thus the total size of the stack is roughly limited only by the available memory on the device; however, individual objects in the stack are limited in size to the largest of the available spaces in the data banks.

The software stack is always allocated one block of memory; however, this memory may cross bank boundaries. The maximum size of the software stack is typically limited by the amount of free data space remaining. There are no compile-time or runtime checks made for stack overflow, but compiler errors are produced if a function defines too many stack-based objects.

There are instruction-set-imposed limitations on the amount of stack data that each function can define and the compiler can detect if these data allocations will be exceeded. The limits are 127 bytes for PIC18 devices and typically 31 bytes for Enhanced Mid-range devices. This latter limit can be exceeded, but for each additional 31 bytes of offset required to access an object, there will be several additional instructions output.

When reentrant functions call other reentrant functions, the stack pointer is incremented as any parameters to the called function are loaded. This increases the offset from the new top-of-stack position to the stack-based objects defined by the calling function. If this offset becomes too large, a warning (1488) or error might result when trying to access stack-based objects in the calling function. A similar situation exists if the called reentrant function returns a value, as this might also be located on the stack. For these reasons, the entire stack depth might not be usable for every function.

5.4.2.2.2 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.

5.4.3 Objects in Program Space

Variables with static storage duration and that are qualified `const` are placed into program memory. Some auto objects might also be positioned there, as discussed in 5.1.2.4 Const Auto Objects. Note that some optimizations (see 4.6.6.8 Cacheconst Option) attempt to cache `const`-qualified objects into data memory, where they are accessed more efficiently.

Accessing data located in program memory is much slower than accessing objects in the data memory. The code associated with the access is also larger.

Enhanced Mid-range devices can directly read their program memory, although the compiler will still usually store data as `retlw` instructions. This way the compiler can either produce code that can call these instructions to obtain the program memory data as with the ordinary Mid-range devices, or directly read the operand to the instruction (the LSB of the `retlw` instruction). The most efficient access method can be selected by the compiler when the data needs to be read.

Data can be stored as individual bytes in the program memory of PIC18 devices. This can be read using table read instructions.

For other 8-bit PIC devices, the program space is not directly readable by the device. For these devices, the compiler stores data in the program memory by means of `retlw` instructions which can be called and will return a byte of data in the W register. The compiler will generate the code necessary to make it appear that program memory is being read directly.

5.4.3.1 Object Size Limitations

A `const`-qualified object cannot be made larger than the available device memory size, but there can be other restrictions as to how large each object can be.

For Baseline PIC devices, the maximum size of a single `const` object is 255 bytes. However, you can define as many `const` objects as required provided the total size does not exceed the available program memory size of the device.

For Mid-range devices, the maximum size of a `const`-qualified object is limited by the available program memory.

For PIC18 devices, the maximum size of a `const`-qualified object is limited by the smaller of the size of the available program memory or 0xFFFF, that is, objects must be less than 64kB in size, even if the device implements more than this amount of program memory. Note, however, that program memory from address 0 up to an address equal to the highest data memory address is typically not used to hold this data.

In addition to the `const` data itself, the compiler might need to output short routines to access the data in program memory. This additional code is included only once, regardless of the amount or number of `const`-qualified objects but might further limit the maximum size of a `const` object.

5.4.3.2 Changing The Default Allocation

You can change the default memory allocation of `const`-specified objects by either:

- Reserving memory locations
- Making the objects absolute
- Placing objects in their own section and linking that section

- Using the caching optimization

If you wish to prevent variables from using one or more program memory locations so the locations can be used for some other purpose, it is recommended to reserve the memory using the memory adjust options. See 4.6.1.18 Reserve Option for information on how to do this.

If only a few `const` objects are to be located at specific addresses in program space memory, then the objects can be made absolute. Absolute variables are described in 5.4.4 Absolute Variables.

Objects in program memory can also be placed in their own section by using the `__section()` specifier, allowing this section to be linked at the required location (see 5.14.3 Changing and Linking the Allocated Section).

Some objects in program memory can be cached in data memory by enabling the caching optimization (see 4.6.6.8 Cacheconst Option). It is not possible to cache individual objects, and which objects are cached by this optimization can vary from build to build as the program is changed.

5.4.4 Absolute Variables

Objects can be located at a specific address by following their declaration with the construct `__at(`*`address`*`)`, where *`address`* is an integer constant that represents the location in memory where the variable is to be positioned. Such a variable is known as an absolute variable.

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 for each object to be relocated.

5.4.4.1 Absolute Objects In Data Memory

Any object which has static storage duration and which has file scope can be placed at an absolute address in data memory, thus all but static objects defined inside a function and stack-based objects can be made absolute.

For example:

```
volatile unsigned char Portvar __at(0x06);
```

will declare a variable called `Portvar` located at 06h in the data memory. Note that the `__at()` construct can be placed before or after the variable identifier in the definition, but to be compatible with the C90 standard, it should be place after the identifier

Note: Defining absolute objects can fragment memory and can make it impossible for the linker to position other objects. If absolute objects must be defined, try to place them at either end of a memory bank so that the remaining free memory is not fragmented into smaller chunks.

The compiler will mark storage for absolute objects as being used if the address is within general-purpose RAM (GPR). Note that address 0 (zero) is not considered to be part of the GPR for PIC18 devices. Taking the address of an object at this address would yield a `NULL` pointer, and the compiler often uses this location for internally-defined objects. You should not place absolute objects at address 0.

No checks are made for the overlap of absolute variables with other absolute variables. There is no harm in defining more than one absolute variable to live at the same address if this is what you require. No warning will be issued if the address of an absolute object lies outside the memory of the device or outside the GPR defined by the linker classes.

Absolute variables in RAM cannot be initialized when they are defined and not cleared by the runtime startup code. After defining absolute variables, assign them an initial value at a suitable point in your main-line code, if required.

Objects should not be made absolute to force them into common (unbanked) memory. Always use the `__near` qualifier for this purpose (see 5.3.9.4 Near Type Qualifier).

When defining absolute `bit` variables (see 5.3.2.1 Bit Data Types And Variables), the address specified must be a bit address. A bit address is obtained by multiplying the byte address by 8, then adding the bit offset within that bit. For example, to place a `bit` variable called mode at bit position #2 at address 0x50, use the following:

```
bit mode __at(0x282);
```

When compiling for an Enhanced Mid-range PIC device, the address specified for absolute objects can be either a conventional banked memory address or a linear address. As the linear addresses start above the largest banked address, it is clear which type of address has been specified. In the following example:

```
int input[100] __at(0x2000);
```

it is clear that input should placed at address 0x2000 in the linear address space, which is address 0x20 in bank 0 RAM in the conventional banked address space.

5.4.4.2 Absolute Objects In Program Memory

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

For example:

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

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

Note that the `__at()` construct can be placed before or after the variable identifier in the definition, but to be compatible with the C90 standard, it should be place after the identifier.

An uninitialized `extern const` object can be made absolute and is useful when you want to define a placeholder object that does not make a contribution to the output file.

5.4.5 Variables in EEPROM

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

5.4.5.1 EEPROM Variables

When compiling for Baseline and Mid-range parts, the `__eeprom` qualifier allows you to create named C objects that reside in the EEPROM space (see 5.3.9.2 EEPROM Type Qualifier).

Objects qualified `__eeprom` 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 the HEX file and are burnt into the EEPROM when you program the device. Thus, 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.

The following example defines two arrays in EEPROM.

```
__eeprom char regNumber[10] = "A93213";
__eeprom int lastValues[3];
```

For both these objects, their initial values will appear in the HEX file. Zeros will be used as the initial values for `lastValues`.

The generated code to access `__eeprom`-qualified objects will be much longer and slower than code to access RAM-based objects. Consider copying values from EEPROM to regular RAM-based objects and using these in complicated expressions to avoid can't generator code error messages.

5.4.5.2 EEPROM Initialization

For those devices that support external programming of their EEPROM data area, the `__EEPROM_DATA()` macro can be used to place values into the HEX file ready for programming into the EEPROM. This macro cannot used to write to EEPROM locations during runtime.

The macro is used as follows.

```
#include <xc.h>
__EEPROM_DATA(0, 1, 2, 3, 4, 5, 6, 7);
```

The macro has eight parameters, representing eight data values. Each value should be a byte in size. Unused values should be specified with zero.

The `__EEPROM_DATA()` macro arguments expand into assembly code. Ensure that any operators or tokens in argument expressions are written in assembly code (see 6.1 MPLAB XC8 Assembly Language).

The macro can be called multiple times to define the required amount of EEPROM data. It is recommended that the macro be placed outside any function definition.

The values defined by this macro share the EEPROM space with `__eeprom`-qualified objects, but cannot be used to initialize such objects. The section used by this macro to hold the data values is different to those used by `__eeprom`-qualified objects. The link order of these sections can be adjusted, if required (see 4.6.12 Mapped Linker Options).

For convenience, the macro `_EEPROMSIZE` represents the number of bytes of EEPROM available on the target device.

5.4.5.3 EEPROM Access Functions

The library functions `eeprom_read()` and `eeprom_write()`, can be called to read from, and write to, the EEPROM during program execution.

These functions are available for all Mid-range devices that implement EEPROM (described in 9.2.4 eeprom_read Function and 9.2.5 eeprom_write Function).

For convenience, the macro `_EEPROMSIZE` represents the number of bytes of EEPROM available on the target device.

5.4.5.4 EEPROM Access Macros

Macro versions of the EEPROM access functions are also provided (described in 9.2.6 EEPROM_READ Macro and 9.2.7 EEPROM_WRITE Macro).

5.4.6 Variables in Registers

With MPLAB XC8, there is no direct control of placement of variables in registers. The `register` keyword (which can only be used with auto variables) is silently ignored and has no effect on the allocation of variables.

Some arguments are passed to functions in the W register rather than in a memory location; however, these values will typically be stored back to memory by code inside the function so that W can be used by code associated with that function. See 5.7.5 Function Parameters for more information as to which parameter variables can use registers.

5.4.7 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. 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*.

5.4.7.1 Dynamic Memory Allocation for PIC Devices

The MPLAB XC8 C Compiler implements a simplified dynamic memory allocation scheme for projects targeting PIC18 or Enhanced Mid-range devices and when the C99 language standard has been selected. Dynamic memory allocation is not implemented for other devices, due to their restrictive memory arrangement. Dynamic allocation is not available for any device when the C90 language standard has been selected.

The simplified dynamic memory allocation scheme permits the use of all the standard memory allocation library functions, but the algorithms employed are cut down to reduce the resources used. This is advantageous for devices with small amounts of data and program memory, but requires that this form of allocation be used carefully to avoid memory wastage.

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 maintained in a linked list and will be considered for re-allocation whenever additional memory is subsequently requested. The management of these freed blocks, in particular how they might be merged, is rudimentary.

How your program is structured can affect how efficiently dynamic memory allocation operates. If a block of allocated memory can be freed before allocating additional blocks, this will aid in the reuse of the freed memory blocks. Allocating and freeing memory blocks in any order will lead to fragmentation of the unused memory in the heap, which might render it unavailable for future allocation. Prudent program designers will avoid using dynamic memory allocation, if at all possible.

Attempting to allocate 0 bytes of memory will result in 1 byte being requested, with a pointer to that memory, when available, being returned in the usual way.

Reallocation of memory using `realloc()` will always allocate a new block with the requested size, where possible, and then free the old block after the existing data has been copied to the new block.

As the heap can grow during program execution, there is the potential for it to overwrite other memory areas or exceed the memory of the device. To prevent these situations, the heap can be limited to a maximum size, which is specified using the `-mheap` option (see 4.6.1.11 Heap Option). If an attempt at runtime to allocate memory would result in the heap exceeding the maximum size, the allocation will not take place and the allocation function will return a null pointer to indicate this situation.

Programs that use a heap might also use software stacks for memory allocation of stack-based objects. (One software stack is used by each call graph in a program, e.g. main-line code and interrupt functions.) As both the heap and stacks grow and shrink during program execution, device memory not consumed by statically allocated objects must be shared between them. The maximum size of the software stacks can also be set. That is performed using the `-mstack` option (see 4.6.1.22 Stack Option). An argument of `auto` can be specified to both the `-mstack` and `-mheap` options, in which case the compiler will evenly distribution free memory between the stack and heap.

The memory allocated to the heap is held by a psect called `heap`.

The maximum amount of heap memory *used* by a program is not known at compile time, but the maximum amount of memory *reserved* for the heap is shown in the memory summaries and reports produced by the compiler after compilation.

5.4.8 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.

5.5 Operators and Statements

The MPLAB XC8 C Compiler supports all the ANSI operators, some of which behave in an implementation defined way, see 11. 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.

5.5.1 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.

5.5.2 Rotation

Rotate operations can be performed in your code, despite the C language not including a rotate operator.

The compiler will detect expressions that implement rotate operations using shift and logical operators and compile them efficiently.

For the following code:

```
c = (c << 1) | (c >> 7);
```

if `c` is unsigned and non-volatile, the compiler will detect that the intended operation is a rotate left of 1 bit and will encode the output using the PIC MCU rotate instructions. A rotate left of 2 bits would be implemented with code like:

```
c = (c << 2) | (c >> 6);
```

This code optimization will also work for integral types larger than a `char`. If the optimization cannot be applied, or this code is ported to another compiler, the rotate will be implemented, but typically with shifts and a bitwise OR operation.

5.5.3 Switch Statements

The compiler can encode `switch` statements using one of several strategies. By default, the compiler chooses a strategy based on the `case` values that are used inside the `switch` statement. Each `switch` statement is assigned its strategy independently.

The type of strategy can be indicated by using the `#pragma switch` directive (see 5.13.3.10 The #pragma Switch Directive), which also lists the available strategy types. There can be more than one strategy associated with each type.

There is information printed in the assembly list file for each `switch` statement detailing the value being switched and the `case` values listed (see 6.3.4 Switch Statement Information).

Case ranges (`lo_value … hi_value:`) are not a feature of the C90 or C99 standard, and it is recommended that they are not used in programs. The compiler will correctly compile `switch` statements using case ranges if the P1 front end is being used and the CCI is not enabled. The P1 front end is used when building for the C90 standard or building for any device other than PIC18 or Enhanced Mid-range. At other times, case ranges are not honored.

5.6 Register Usage

The assembly generated from C source code by the compiler will use certain registers in the PIC MCU register set. Most importantly, the compiler assumes that nothing other than code it generates can alter the contents of these registers.

The registers that are special and which are used by the compiler are listed in the following table.

Table 5-10. Registers Used By The Compiler

Register name	Applicable devices
`WREG`	All 8-bit devices
`STATUS`	All 8-bit devices
`PCLATH`	All Mid-range devices
`PCL`, `PCLATH`, `PCLATU`	All PIC18 devices
`BSR`	Enhanced Mid-range and PIC18 devices
`FSR`	Non-Enhanced Mid-range devices
`FSR0L`, `FSR0H`, `FSR1L`, `FSR1H`	Enhanced Mid-range and PIC18 devices
`FSR2L`, `FSR2H`	All PIC18 devices
`TBLPTRL`, `TBLPTRH`, `TBLPTRU`, `TABLAT`	All PIC18 devices
`PRODL`, `PRODH`	All PIC18 devices
`btemp`, `wtemp`, `ttemp`, `ltemp`, `lltemp`	Enhanced Mid-range and PIC18 devices

The `xtemp` registers are variables that the compiler treats as registers. These are saved like any other register if they are used in interrupt code. The `lltemp` register is only available when using Enhanced Mid-range or PIC18 devices.

The compiler will not be aware of changes to a register's value when the register itself is a C lvalue (assignment destination). For example, if the statement `WREG = 0;` was encoded using the `clrf` instruction, the compiler would not consider this as modifying the W register. Nor should these registers be changed directly by in-line assembly code, as shown in the following example which modifies the ZERO bit in the STATUS register.

```
#include <xc.h>
void getInput(void)
```

```
{
    asm("bcf ZERO"); //do not write using inline assembly code
    process(c);
}
```

If any of the applicable registers listed in the table are used by interrupt code, they will be saved and restored when an interrupt occurs, either in hardware or software (see 5.8.4 Context Switching).

5.7 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.

5.7.1 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.

5.7.1.1 Interrupt Specifier

The `__interrupt()` specifier indicates that the function is an interrupt service routine and that it is to be encoded specially to suit this task. Interrupt functions are described in detail in 5.8.1 Writing an Interrupt Service Routine.

5.7.1.2 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.

The rules for defining inline functions differ slightly to the C99 language standard in that a file scope declaration for a function using the `inline` function specifier (but not `extern`) is treated as an external definition of that function. This means that you do not need to provide an additional external definition of the function to build the code.

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

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

The compiler can either inline the function or call the function definition, at its discretion. Your code should not make any assumption about whether in-lining took place. A function containing in-line assembly will not be in-lined. Some generated assembly code sequences will also prevent in-lining of a function. A warning will be generated if the inline function references `static` objects (to comply with the C Standard) or is not in-lined successfully.

All function calls to a function that was in-lined by the compiler will be encoded as if the call was replaced with the body of the called function. This is performed at the assembly code level. in-lining will only take place if the assembly optimizers are enabled, which occurs at the higher optimization levels and which precludes this action if you are running an unlicensed compiler. The function itself might still be encoded normally by the compiler even if it is in-lined.

If in-lining takes place, this will increase the program's execution speed, since the call/return sequences associated with the call will be eliminated. It will also reduce the hardware stack usage as no call instruction is executed; however, this stack-size reduction is not reflected in the call graphs, as these graphs are generated before in-lining takes place.

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 (see 3.5.13 How Can I Tell How Big a Function Is?).

This specifier performs the same task as the `#pragma inline` directive (see). Note that the optimizers can also implicitly inline small called-only-once routines (see 5.13.3.4 The #pragma Intrinsic Directive6.2 Assembly-Level Optimizations).

If the `xc8-cc` flag `-Wpedantic` is used, the `inline` keyword becomes unavailable, but you can use the `__inline` keyword.

5.7.1.3 Reentrant And Nonreentrant Specifiers

The `__reentrant` and `__nonreentrant` function specifiers indicate the function model (stack) that should be used for that function's stack-based variables (auto and parameters), as shown in the Table 5-11 table. The aliases `__software` and `__compiled`, respectively, can be used if you prefer. You would only use these specifiers if the default allocation is unacceptable.

Table 5-11. Stack Related Function Specifiers

Specifier	Allocation for Stack-based variables
`__compiled, __nonreentrant`	Always use the compiled stack; functions are nonreentrant
`__software, __reentrant`	Use the software stack, if available; functions are reentrant

The following shows an example of a function that will always be encoded as reentrant.

```
__reentrant int setWriteMode(int mode)
{
    accessMode = (mode!=0);
    reutrn mode;
}
```

These specifiers override any setting indicated using the `-mstack` option (see 4.6.1.22 Stack Option). If neither function specifier is used with a function and the `-mstack` option is not specified (or specified as hybrid), then the compiler will choose the stack to be used by that function for its stack-based variables.

The `__reentrant` specifier only has an effect if the target device supports a software stack. In addition, not all functions allow reentrancy. Interrupt functions must always use the compiled stack, but functions they call may use the software stack. Functions encoded for Baseline and Mid-range devices always use the nonreentrant model and the compiled stack.

Repeated use of the `__software` (`__reentrant`) specifier will increase substantially the size of the software stack leading to possible overflow. The size of the software stack is not accurately known at compile time, so the compiler cannot issue a warning if it is likely to overwrite memory used for some other purpose.

See 5.2.4.2 Data Stacks for device specific information relating to the data stacks available on each device.

5.7.1.4 External Functions

Functions that are defined outside the projects C source files (e.g., a function defined in a separate bootloader project or in an assembly module) will require declarations so that the compiler knows how to encode calls to those functions.

A function declaration will look similar to the following example. Note that the `extern` specifier is optional, but make it clear this is a declaration.

```
extern int clockMode(int);
```

5.7.2 Allocation of Executable Code

Code associated with functions is always placed in program memory.

On Baseline and Mid-range devices, the program memory is paged. Program memory addresses are still sequential across a page boundary, but the paging means that calls or jumps from code in one page to a label in another must use a longer sequence of instructions. Your device data sheet has more information on the program memory and instruction set for your device.

PIC18 devices do not implement any program memory paging. The `call` and `goto` instruction are two-word instructions and their destinations are not limited.

The generated code associated with each function is initially placed in sections by the compiler (see 5.14.1 Compiler-Generated Psects). When the program memory is paged, the optimizer tries to allocate several functions to the same section so they can use the shorter form of `call` and `jump`. These sections are linked anywhere in the program memory (see 5.14.2 Default Linker Classes), although Baseline devices restrict the entry point of functions to within the first 256 location in each page.

The base name of each section is tabulated below. See 5.14.1.1 Program Space Psects for a full list of all program-memory section names.

maintext The generated code associated with the special function, `main()`, is placed in this section. Some optimizations and features are not applied to this psect.

text*n* These sections (where *n* is a decimal number) contain all other executable code that does not require a special link location.

5.7.3 Changing the Default Function Allocation

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

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

If you wish to prevent functions from using one or more program memory locations so that these locations can be used for some other purpose, it is recommended to reserve the memory using the memory adjust options (see 4.6.1.18 Reserve Option).

The easiest method to explicitly place individual functions at a known address is to make them absolute by using the `__at(`*`address`*`)` construct in a similar fashion to that used with absolute variables.

The compiler will issue a warning if code associated with an absolute function overlaps with code from other absolute functions. No warning will be issued if the address of an absolute object lies outside the memory of the device or outside the memory defined by the linker classes. 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 */
}
```

Note that the `__at()` construct should be placed before the function identifier in the definition (but in this position it is not legal when building code for the C90 language standard). You can, however, place the `__at()` after the identifier if it is used in a declaration. The `__at()` construct can then be removed from the definition altogether, as in the following example.

```
int mach_status(int mode) __at(0x400); // declaration indicates address

int mach_status(int mode) { ... } // defintition can omit the __at()
```

This construct cannot be used with interrupt functions. See 4.6.1.3 Codeoffset Option for information on how to move Reset and interrupt vector locations (which can be useful for designing applications such as bootloaders).

The code generated for absolute functions is placed in a section dedicated only to that function. The section name has the form shown below (see 5.14.1 Compiler-Generated Psects for a full list of all section names).

***xxx*_text** Defines the section for a function that has been made absolute. *`xxx`* will be the assembly symbol associated with the function, e.g., the absolute function `rv()` would appear in the psect called `_rv_text`.

Functions can be allocated to a user-defined psect using the `__section()` specifier (see 5.14.3 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 another section, or anywhere in an address range. When used with interrupt functions, this specifier will only affect the position of the interrupt function body. Never place functions into a section that is also used to hold non-executable objects, such as `const` objects, as this might cause runtime failures or affect the ability to debug the functions.

Regardless of how a function is located, take care choosing its address. If possible, place functions at either end of a program memory page (if relevant) to avoid fragmenting memory and increasing the possibility of linker errors. Place functions in the first page, which contains the reset and interrupt code, rather than in pages higher in memory, as this will assist the optimizations that merge psects.

5.7.4 Function Size Limits

For all devices, the code generated for a regular function is limited only by the available program memory; however the longer jump sequences within a function if it is located across more than one page will decrease efficiency. See 5.7.2 Allocation of Executable Code for more details.

Interrupt functions (see 5.8.1 Writing an Interrupt Service Routine) however, are limited to one page in size and cannot be split over multiple pages.

5.7.5 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, and on which stack model is used with the function.

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.

5.7.5.1 Compiled Stack Parameters

For non-reentrant functions using the compiled stack, the compiler will pass arguments in the W register, or in the called function's parameter memory.

If the first parameter is one byte in size, it is passed in the W register. All other parameters are passed in the parameter memory. The parameter memory will be located in the compiled stack (see 5.2.4.2.1 Compiled Stack Operation).

Parameters are referenced as an offset from the symbol `?_function`, where *`function`* is the name of the function in which the parameter is defined (i.e., the function that is to be called).

Unlike auto variables, parameter variables are allocated memory strictly in the order in which they appear in the function's prototype. This means that a function's parameters will always be placed in the same memory bank; whereas auto variables for a function can be allocated across multiple banks and in any order.

The arguments for unnamed parameters in functions that take a variable argument list (defined using an ellipsis in the prototype), are placed in the parameter memory, along with named parameters.

Take, for example, the following prototyped function.

```
void test(char a, int b);
```

The function `test()` will receive the parameter `b` in parameter memory (using the two bytes `?_test` and `?_test+1`) and a in the W register.

The compiler needs to take special action if more than one function using the compiled stack can be indirectly called via the same function pointer. Such would be the case in the following example, where any of `sp_ad`, `sp_sub` or `sp_null` could be called via the pointer, `fp`.

```
int (*funcs[])(int, int) = {sp_add, sp_sub, sp_null};
int (*fp)(int, int);
fp = funcs[getOperation()];
result = fp(37, input);
```

In such a case, the compiler treats all three functions in the array as being "buddies."

The parameter(s) to all buddy functions will be aligned in memory, i.e., they will all reside at the same address(es). This way the compiler does not need to know exactly which function is being called. The implication of this is that a function cannot call (either directly or indirectly) any of its buddies. To do so would corrupt the caller function's parameters. An error will be issued if such a call is attempted.

5.7.5.2 Software Stack Parameters

When a function uses the software stack, most arguments to that function will be passed on the stack (see 5.2.4.2.2 Software Stack Operation).

Arguments are pushed onto the stack by the calling function, in a reverse order to that in which the corresponding parameters appear in the function's prototype. Subsequently, and if required, the called function will increase the value stored in the stack pointer to allocate storage for any auto or temporary variables it needs to allocate.

The W register is sometimes used for the first function argument if it is byte-sized. This will only take place for Enhanced Mid-range devices and provided the function does not take a variable number of arguments. If a reentrant function is external (see 5.7.2 Allocation of Executable Code), the W register will never be used to hold any function arguments. The W register is never used by reentrant function arguments when compiling for PIC18 devices.

The arguments for unnamed parameters in functions that take a variable argument list (defined using an ellipsis in the prototype), are placed on the software stack, before those for the named parameters. After all the function's arguments have been pushed, the total size of the unnamed parameters is pushed on to the stack. A maximum of 256 bytes of non-prototyped parameters are permitted per function.

As there is no frame pointer, accessing function parameters (or other stack-based objects) is not recommended in hand-written assembly code.

5.7.6 Function Return Values

Values returned from functions are loaded into a register or placed on the stack used by that function. The mechanism will depend on the function model used by the function.

5.7.6.1 Compiled Stack Return Values

For functions that use the compiled stack, return values are passed to the calling function using the W register, or the function's parameter memory. The memory assigned to the function's parameters (which is no longer needed when the function is ready to return) is reused to reduce the function's code and data requirements.

Single-byte values are returned from a function in the W register. Values larger than a byte are returned in the function's parameter memory area.

For example, the function:

```
int returnWord(void)
{
    return 0x1234;
}
```

will return with the value 0x34 in `?_returnWord`, and 0x12 in `?_returnWord+1`.

For PIC18 targets returning values greater than 4 bytes but less than 8 bytes in size, the address of the parameter area is also placed in the FSR0 register.

Functions that return a value of type `__bit` do so using the carry bit in the STATUS register.

5.7.6.2 Software Stack Return Values

Functions that use the software stack will pass values back to the calling function via `btemp` variables, provided the value is 4 bytes or less in size. The W register will be used to return byte-sized values for Enhanced Mid-range device functions that do not have a variable number of arguments. For objects larger than 4 bytes in size, they are returned on the stack. Reentrant PIC18 functions that return a value of type `__bit` do so using bit #0 in `btemp0`; other devices use the carry bit in the STATUS register.

As there is no frame pointer, accessing the return value location, or other stack-based objects, is not recommended in hand-written assembly code.

5.7.7 Calling Functions

All 8-bit devices use a hardware stack for function return addresses. The maximum depth of this stack varies from device to device.

Typically, call assembly instructions are used to transfer control to a C function when it is called. A call uses one level of hardware stack that is freed after the called routine executes a `return` or `retlw` instruction. Nested function calls will increase the stack usage of a program. If the hardware stack overflows, function return addresses will be overwritten and the code will eventually fail.

For PIC18 devices, a call instruction is the only way in which function calls are made, but for other 8-bit devices, the `-mstackcall` option, (see 4.6.1.23 Stackcall Option), can controls how the compiler behaves when the compiler detects that the hardware stack might overflow due to too many nested calls. When this option is enabled, the compiler will, instead of issuing a warning, automatically swap to using a managed stack that involves the use of a lookup table and which does not require use of the hardware stack.

When the lookup method is being employed, a function is reached by a jump (not a call) directly to its address. Before this is done the address of a special “return” instruction (implemented as a jump instruction) is stored in a temporary location inside the called function. This return instruction will be able to return control back to the calling function.

This means of calling functions allows functions to be nested deeply without overflowing the limited stack available on Baseline and Mid-range devices; however, it does come at the expense of memory and program speed.

5.7.7.1 Indirect Calls

When functions are called indirectly using a pointer, the compiler employs a variety of techniques to call the intended function.

The PIC18 and Enhanced Mid-range devices all use the value in the function pointer to load the program counter with the appropriate address. For PIC18 devices, the code loads the TOS registers and executes a return to perform the call. For Enhanced Mid-range devices, the `callw` instruction is used. The number of functions that can be called indirectly is limited only by the available memory of the device.

The Baseline and Mid-range devices all use a lookup table which is loaded with jump instructions. The lookup table code is called and an offset is used to execute the appropriate jump in the table. The table increases in size as more functions are called indirectly, but cannot grow beyond 0xFF bytes in size. This places a limit on the number of functions that can be called indirectly, and typically this limit is approximately 120 functions. Note that this limit does not affect the number of function pointers a program can define, which are subject to the normal limitations of available memory on the device.

Indirect calls are not affected by the `-mstackcall` option and the depth of indirect calls on Baseline and Mid-range devices are limited by the hardware stack depth.

5.8 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 operation of interrupts is handled differently by the different device families. Most Baseline devices do not implement interrupts at all; Mid-range devices have one vector location which is linked to all interrupt sources; some PIC18 devices have two independent interrupt vectors, one assigned to low-priority interrupt sources, the other to high-priority sources; and some PIC18 devices implement a vectored interrupt controller (VIC) module with support for one or more interrupt vector tables (IVTs), which can be populated with the addresses of high- or low-priority interrupt functions.

The operation of the IVT on devices with a VIC module can be disabled by clearing the `MVECEN` configuration bit. The device is then said to be operating in legacy mode, operating with dual priorities and dual vector locations. This bit is also used by the compiler to determine how interrupt functions should be programmed, so ensure it is set to `OFF` for legacy operation. Although the vector table is disabled in this mode, the vector locations are still relocatable. By default the vector location will be 0x8 and 0x18, the same for regular PIC18 devices without the VIC module.

The priority scheme implemented by PIC18 devices can also be disabled by clearing the `IPEN` SFR bit. Such devices are then said to be operating in Mid-range compatibility mode and utilize only one interrupt vector, located at address 0x8.

The following are the general steps you need to follow to use interrupts. More detail about these steps is provided in the sections that follow.

For Enhanced Baseline devices with interrupts, Mid-range devices, or PIC18 devices operating in Mid-range compatibility mode:

- Write one interrupt function to process all interrupt sources.
- At the appropriate point in your main-line code, unmask the interrupt sources required by setting their interrupt enable bit in the corresponding SFR.
- At the appropriate point in your code, enable the global interrupt enable to allow interrupts to be generated.

For PIC18 devices without the VIC module, or PIC18 devices operating in legacy mode:

- Plan the priorities to be assigned to each interrupt source. If the device is operating in legacy mode, determine the number of sets of dual interrupt vectors you require.
- Program the `MVECEN` configuration bit if appropriate. A vector table will be encoded by the compiler if this bit is not set to `OFF` for devices with the VIC module.
- Write one interrupt function to process each priority being used. You can define at most two interrupt functions, or two interrupt functions per vector set for devices operating in legacy mode. Consider implementing both interrupt functions to handle accidental triggering of unused interrupts, or use the `-mundefints` option to provide a default action (see 4.6.1.25 Undefints Option).
- Write code to assign the required priority to each interrupt source by writing the appropriate bits in the SFRs.
- If the device is operating in legacy mode and if required, at the appropriate points in your code, select the required set of dual vectors by writing to the IVTBASE registers. Never write the IVTBASE registers if interrupts are enabled. The initial vectors can also be selected by using the `-mivt` option (see 4.6.1.12 Ivt Option).
- At the appropriate point in your code, enable the interrupt sources required.
- At the appropriate point in your code, enable the global interrupt enable.

For devices using the VIC module:

- Plan the priorities associated with each interrupt source and determine the number of interrupt vector tables you require.
- Write as many interrupt functions as required. For fast interrupt response times, write a dedicated function for each interrupt source, although multiple sources can be processed by one function, if desired. Consider one or more additional functions to handle accidental triggering of unused interrupt sources, or use the `-mundefints` option to provide a default action (see 4.6.1.25 Undefints Option).
- Write code to assign the required priority to each interrupt source by writing the appropriate bits in the SFRs.
- If you are using more than one interrupt vector table, at the appropriate points in your code, select the required IVT by writing to the IVTBASE registers. Never write the IVTBASE registers if interrupts are enabled. The initial IVT can also be selected by using the `-mivt` option (see 4.6.1.12 Ivt Option).
- At the appropriate point in your code, enable the interrupt sources required.
- At the appropriate point in your code, enable the global interrupt enable.

Interrupt functions must not be called directly from C code (due to the different return instruction that is used), but interrupt functions can call other functions, both 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.

5.8.1 Writing an Interrupt Service Routine

The prototype and content of an ISR will vary based on the target device and the project being compiled. Observe the following guidelines when writing an ISR.

For devices that do not have the VIC module:

- Write each ISR prototype using the `__interrupt()` specifier.
- Use `void` as the return type and for the parameter specification.
- If your device supports interrupt priorities, with each function use the `low_priority` (or `__low_priority`) or `high_priority` (or `__high_priority`) arguments to `__interrupt()`.
- Inside the ISR body, determine the source of the interrupt by checking the interrupt flag and the interrupt enable for each source that is to be processed and make the relevant interrupt code conditional on those being set.

For devices operating in legacy mode:

- Write each ISR prototype using the `__interrupt()` specifier.
- Use `void` as the return type and specify a parameter list of either `void` or one `char` argument if you need to identify the interrupt source. It is recommended that the parameter list be set to void if you want to ensure portability with devices that do not have the VIC module.

- As arguments to the `__interrupt()` specifier in the ISR prototype, specify the interrupt priority assigned to the function's source, using `low_priority` (or `__low_priority`) or `high_priority` (or `__high_priority`); and optionally, specify the base address of the IVT in which to place the function's address, using `base()` (or `__base()`). It is recommended that the base address be left as the default if you want to ensure portability with devices that do not have the VIC module.
- If the ISR processes more than one source, determine the source of the interrupt from the function's parameter, if specified, or by checking the interrupt flag and the interrupt enable for each source that is to be processed.

For devices which are using the VIC module:

- Write each ISR prototype using only the `__interrupt()` specifier.
- Use `void` as the return type and specify a parameter list of either `void` or one `char` argument if you need to identify the interrupt source.
- As arguments to the `__interrupt()` specifier in the ISR prototype, specify which sources each interrupt function should handle, using either `irq()` or `__irq()`; specify the interrupt priority assigned to the function's source, using either `low_priority` (or `__low_priority`) or `high_priority` (or `__high_priority`); and optionally, specify the base address of the IVT in which to place the function's address, using either `base()` (or `__base()`).
- If the ISR processes more than one source, determine the source of the interrupt from the function's parameter, if specified, or by checking the interrupt flag and the interrupt enable for each source that is to be processed.

For all devices:

- Inside the ISR body, clear the relevant interrupt flag once the source has been processed.
- Do not re-enable interrupts inside the ISR body. This is performed automatically when the ISR returns.
- Keep the ISR as short and as simple as possible. Complex code will typically use more registers that will increase the size of the context switch code.

If interrupt priorities are being used but an ISR does not specify a priority, it will default to being high priority. It is recommended that you always specify the ISR priority to ensure your code is readable.

If you supply an `irq()` or `base()` argument to the `__interrupt()` specifier with a device that does not have the VIC module, an error will be issued by the compiler. If you use this specifier with a device that is configured for legacy mode, supplying an `irq()` argument will result in an error from the compiler; however, you may continue to use the `base()` argument if required.

Devices that have the VIC module identify each interrupt with a number. This number can be specified with the `irq()` argument to `__interrupt()` if the vector table is enabled, or you can use a compiler-defined symbol that equates to that number. You can see a list of all interrupt numbers, symbols and descriptions by opening the files `pic_chipinfo.html` or `pic18_chipinfo.html` in your favorite web browser, and selecting your target device. Both these files are located in the `docs` directory under your compiler's installation directory.

Interrupt functions always use the non-reentrant function model. These functions ignore any option or function specifier that might otherwise specify reentrancy.

The compiler processes interrupt functions differently to other functions, generating code to save and restore any registers used by the function and a special return instruction.

An example of an interrupt function written for code not using the IVT is shown below. Notice that the interrupt function checks for the source of the interrupt by looking at the interrupt enable bit (e.g., `TMR0IE`) and the interrupt flag bit (e.g., `TMR0IF`). Checking the interrupt enable flag is required since interrupt flags associated with a peripheral can be asserted even if the peripheral is not configured to generate an interrupt.

```
int tick_count;
void __interrupt(high_priority) tcInt(void)
{
    if (TMR0IE && TMR0IF) {  // any timer 0 interrupts?
        TMR0IF=0;
        ++tick_count;
    }
    if (TMR1IE && TMR1IF) {  // any timer 1 interrupts?
        TMR1IF=0;
        tick_count += 100;
    }
```

```
    // process other interrupt sources here, if required
    return;
}
```

Here is the same function code, split and modified for a device using vector tables. Since only one interrupt source is associated with each ISR, the interrupt code does not need to determine the source and is therefore faster.

```
void __interrupt(irq(TMR0),high_priority) tc0Int(void)
{
    TMR0IF=0;
    ++tick_count;
    return;
}
void __interrupt(irq(TMR1),high_priority) tc1Int(void)
{
    TMR1IF=0;
    tick_count += 100;
    return;
}
```

If you prefer to process multiple interrupt sources in one function, that can be done by specifying more than one interrupt source in the `irq()` argument and using a function parameter to hold the source number, such as in the following example.

```
void __interrupt(irq(TMR0,TMR1),high_priority) tInt(unsigned char src)
{
    switch(src) {
    case IRQ_TMR0:
        TMR0IF=0;
        ++tick_count;
        break;
    case IRQ_TMR1:
        TMR1IF=0;
        tick_count += 100;
        break;
    }
    return;
}
```

The VIC module will load the parameter, in this example, `src`, with the interrupt source number when the interrupt occurs.

The special interrupt source symbol, `default`, can be used to indicate that the ISR will be linked to any interrupt vector not already explicitly specified using `irq()`. You can also populate unused vector locations by using the `-mundefints` option (see 4.6.1.25 Undefints Option).

By default, the interrupt vector table will be located at an address equal to the reset value of the IVTBASE register, which is the legacy address of 0x8. The `base()` argument to `__interrupt()` can be used to specify a different table base address for that function. This argument can take one or more comma-separated addresses. The base address cannot be set to an address lower than the reset value of the IVTBASE register.

By default and if required, the compiler will initialize the IVTBASE register in the runtime startup code. You can disable this functionality by turning off the `-mivt` option (see 4.6.1.12 Ivt Option). This option also allows you to specify an initial address for this register, for the initial vector table that will be used. If vectored interrupts are enabled but you do not specify an address using this option, the vector table location will be set to the lowest table address used in the program, as specified by the `base()` arguments to `__interrupt()`.

If you use the `base()` argument to implement more than one table of interrupt vectors, you must ensure that you allocate sufficient memory for each table. The compiler will emit an error message if it detects an overlap of any interrupt vectors.

The following examples show the interrupt function prototypes for two ISRs which handle the timer 0 and 1 interrupt sources. These are configured to reside in independent vector tables whose base addresses are 0x100 and 0x200. All other interrupt sources are handled by a low-priority ISR, `defIsr()`, which appears in both vector tables. For these ISRs to become active, the IVTBASE register must first be loaded either 0x100 or 0x200. Changing the address in this register allows you to select which vector table is active.

```
void __interrupt(irq(TMR0,TMR1),base(0x100)) timerIsr(void)
{...}
void __interrupt(irq(TMR0,TMR1),base(0x200)) altTimerIsr(void)
{...}
void __interrupt(irq(default),base(0x100,0x200),low_priority) defIsr(void)
{...}
```

The vector table for devices operating in legacy mode will only have two vectors, those being the high and low priority interrupt vectors. The offsets of these vectors from the base of the table are 0x8 and 0x18, respectively. As it does in other instances, the `base()` argument, if it is required, specifies the base address of the vector table, not the address of a vector within that table. Thus, the following code defines both priority ISRs for a device in legacy mode and where the table base address has been moved to address 0x2000. These ISRs will appear at addresses 0x2008 and 0x2018.

```
void __interrupt(base(0x2000), high_priority) highIsr(void)
{...}

void __interrupt(base(0x2000), low_priority) lowIsr(void)
{...}
```

Devices operating in legacy mode still allow you to define more than one interrupt vector table. For each table, use a different base address and define at most two interrupt functions for the high and low priority interrupts.

5.8.2 Changing the Default Interrupt Function Allocation

Moving the code associated with interrupt functions is more difficult than that for ordinary functions, as interrupt routines have entry points strictly defined by the device.

You can use the `__section()` specifier (see 5.14.3 Changing and Linking the Allocated Section) if you want to move the interrupt function, but leave the interrupt entry point at the default vector location.

To move the vector location, see 5.8.3 Specifying the Interrupt Vector.

5.8.3 Specifying the Interrupt Vector

For devices that do not have the VIC module, the process of populating the interrupt vector locations is fully automatic. The compiler links the interrupt code entry point to the fixed vector locations. Typically the entry point code will be all or part of the code that performs the interrupt context switch and the body of the interrupt function will be located elsewhere.

The location of these 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 and select which will be active during program execution. An error will result if there are more interrupt functions than interrupt vectors in a program.

For devices that have the VIC module, you have more freedom in how interrupt functions can be executed at runtime. When the IVT is enabled, these devices employ a table of interrupt vectors. Each table entry can hold an address, which is read when the corresponding interrupt is triggered, and the device will jump to that address. The vector table entry corresponding to an interrupt function is automatically completed by the compiler, based on the information in the `irq()` (or `__irq()`) and `base()` (or `__base()`) arguments to `__interrupt()`, see 5.8.1 Writing an Interrupt Service Routine.

Although the addresses in the vector table cannot be changed at runtime, it is possible to construct more than one table and have the device swap from one table to another. Changing the active vector table is performed by changing the vector table base address, which is stored in the IVTBASE registers. Since these registers cannot be modified atomically, you must disable all interrupts before changing their content. The following example shows how this might be performed in C code.

```
di();    // disable all interrupts
IVTBASEU = 0x0;
IVTBASEH = 0x2;
IVTBASEL = 0x0;
ei();    // re-enable interrupts
```

You can also preload this register by using the `-mivt` option (see 4.6.1.12 Ivt Option).

For devices with the VIC module operating in legacy mode, the vector table is disabled and the dual-priority vectors employed by regular PIC18 devices are used. These vector locations will then hold an instruction, not an address, but unlike regular PIC18 devices, the program can use the IVTBASE register to map the vector locations to any address and you can define two interrupt functions for each base address.

Do not confuse the function of the IVTBASE register with the `-mcodeoffset` option (see 4.6.1.3 Codeoffset Option). The option moves the code associated with reset but does not change the address to which the device will vector on reset. For devices without the VIC, the option also moves the code associated with the interrupts, but again does not change the interrupt vector addresses. For devices with the VIC, even if they are running in legacy mode, the code associated with interrupts is not affected by this option. Instead, for those devices, the IVTBASE registers controls the address at which the vector table is assumed to occupy (or the address of the interrupt vectors when devices are running in legacy mode).

If you are writing an application that is loaded by a bootloader for devices with the VIC, you will typically need to use both the `-mcodeoffset` option and the IVTBASE registers to ensure that the reset and interrupt entry points are shifted. The option shifts the reset location and, provided the interrupt functions have used the appropriate `base()` address argument, changing the register will ensure that the shifted vector table is correctly accessed. This requirement is relevant for devices operating in legacy mode.

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 default as its `irq()` interrupt source, or assigned a default instruction by using the `-mundefints` option (see 4.6.1.25 Undefints Option).

5.8.4 Context Switching

The compiler will automatically link code into your project which saves the current status when an interrupt occurs and then restores this status when the interrupt returns.

5.8.4.1 Context Saving On Interrupts

Some registers are automatically saved by the hardware when an interrupt occurs. Any registers or compiler temporary objects used by the interrupt function, other than those saved by the hardware, will be saved in code generated by the compiler. This is the context save or context switch code.

See 5.6 Register Usage for the registers that must be saved and restored either by hardware or software when an interrupt occurs.

Enhanced Mid-range PIC devices save the WREG, STATUS, BSR and FSRx registers in hardware (using special shadow registers) and hence these registers do not need to be saved by software. The registers that might need to be saved by software are the BTEMP registers(3), compiler temporary locations that act like registers.

Other Mid-range PIC processors only save the entire PC (excluding the PCLATH register) when an interrupt occurs. The WREG, STATUS, FSR and PCLATH registers and any BTEMP registers must be saved by code produced by the compiler, if required.

By default, the PIC18 high-priority interrupt function will utilize its internal shadow register to save the W, STATUS and BSR registers. For the low priority PIC18 interrupts, or when the shadow registers cannot be used, all registers that have been used by the interrupt code will be saved by software.

If the PIC18 device has the Vectored Interrupt Controller module, it additionally saves the FSRx, PCLATHx and PRODx registers to shadow registers. All other used registers are saved by software. Separate shadow registers are available for low- and high-priority interrupts.

Note that for some older devices, the compiler will not use the shadow registers if compiling for the MPLAB ICD debugger, as the debugger itself utilizes these shadow registers. Some errata workarounds also prevent the use of the shadow registers (see 4.6.1.10 Errata Option).

The compiler determines exactly which registers and objects are used by an interrupt function, or any of the functions that it calls and saves these appropriately.

Assembly code placed in-line within the interrupt function is not scanned for register usage. Thus, if you include in-line assembly code into an interrupt function (or functions called by the interrupt function), you may have to add extra assembly code to save and restore any registers used.

[2] These registers are memory locations allocated by the compiler, but are treated like registers for code generation purposes. They are typically used when generating reentrant code.

If the software stack is in use, the context switch code will also initialize the stack pointer register so it is accessing the area of the stack reserved for the interrupt. See 5.4.2.2.1 Object Size Limits for more information on the software stack.

5.8.4.2 Context Restoration

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.

If the software stack is in use, the context restoration code will also restore the stack pointer register so that it is accessing the area of the stack used before the interrupt occurred. See 5.4.2.2.1 Object Size Limits for more information on the software stack.

5.8.5 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 GIE bit in the INTCON or INTCON0 register. 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:

```
ADIE = 1;  // A/D interrupts will be used
PEIE = 1;  // all peripheral interrupts are enabled
ei();      // enable all interrupts
// ...
di();      // disable all interrupts
```

Note: Never re-enable interrupts inside the interrupt function itself. Interrupts are automatically re-enabled by hardware on execution of the `retfie` instruction. Re-enabling interrupts inside an interrupt function can result in code failure.

5.8.6 Accessing Objects From Interrupt Routines

Reading or writing objects from interrupt routines can be unsafe if other functions access these same objects.

The compiler will automatically treat as `volatile` any variables that are referenced in an interrupt routine; however, it is recommended that you explicitly mark these variables using the `volatile` specifier to ensure your code is portable (see 5.3.8.2 Volatile Type Qualifier). The compiler will restrict the optimizations performed on `volatile` objects (see 5.12 Optimizations).

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, but many operations on single-byte or bit objects cannot be performed in one instruction.

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. Check the assembler list file to see the code generated for a statement, but remember that the instructions can change as the program is developed, particularly if the optimizers are enabled.

5.8.7 Function Duplication

MPLAB XC8 compiler employs a feature that duplicates the generated code associated with any function that uses a non-reentrant model and that is called from more than one call graph.

There is one call graph associated with main-line code and one for each interrupt function, if defined. The compiler assumes that interrupts can occur at any time. Functions encoded to use the compiled stack are not reentrant, thus such functions called from main-line code and from interrupt code could result in code failure. The compiler will duplicate the output for any non-reentrant function called from more than one call graph. This makes the function appear to be reentrant; however, recursion is still not possible.

Although the compiler could alternatively compile functions using a reentrant model, this feature is not available with all devices and the code might be less efficient. In addition, the `-mstack` option or the `__nonreentrant` specifier can be used to prevent the compiler from choosing this model. See 6.3.3 Function Information to determine which function model was used for each function.

If a function is duplicated, main-line code will call the code generated for the original function and code in each interrupt call graph will call a unique duplicated function. The duplication takes place only in the generated code output; there is no duplication of the C source code itself.

The duplicated code and objects defined by the function use unique identifiers. A duplicate identifier is identical to that used by the original code, but is prefixed with `i1`. Duplicated PIC18 functions use the prefixes `i1` and `i2` for the low- and high-priority interrupts, respectively.

To illustrate, in a program both the function `main()` and a function in interrupt code call a function called `input()`. The generated assembly code for the C function `input()` will use the assembly label `_input`. The corresponding label used by the duplicated function output will be `i1_input`. If `input()` makes reference to a temporary variable, the generated code will use the symbol `??_input` and the duplicate will use `??i1_input`. Even local labels within the function's generated code will be duplicated in the same way. The call graph in the assembly list file, will show the calls made to both of these functions as if they were independently written. These symbols will also be seen in the map file symbol table.

Code associated with library functions are duplicated in the same way. This also applies to implicitly-called library routines, such as those that perform division or floating-point operations associated with C operators.

5.8.7.1 Disabling Duplication

The automatic duplication of non-reentrant functions called from more than one call graph can be inhibited by the use of a special pragma.

Duplication should only be disabled if the source code guarantees that an interrupt cannot occur while the function is being called from any main-line code. Typically this would be achieved by disabling interrupts before calling the function. It is not sufficient to disable the interrupts inside the function after it has been called; if an interrupt occurs when executing the function, the code can fail. See 5.8.5 Enabling Interrupts for more information on how interrupts can be disabled.

The pragma is:

```
#pragma interrupt_level 1
```

The pragma should be placed before the definition of the function that is not to be duplicated. The pragma will only affect the first function whose definition follows.

For example, if the function `read()` is only ever called from main-line code when the interrupts are disabled, then duplication of the function can be prevented if it is also called from an interrupt function as follows.

```
#pragma interrupt_level 1
int read(char device)
{
    // ...
}
```

In main-line code, this function would typically be called as follows:

```
di();  // turn off interrupts
read(IN_CH1);
ei();  // re-enable interrupts
```

The value specified with the pragma indicates for which interrupt the function will not be duplicated. For Mid-range devices, the level should always be `1`; for PIC18 devices it can be `1` or `2` for the low- or high-priority interrupt functions, respectively. To disable duplication for both interrupt priorities, use the pragma twice to specify both levels `1` and `2`. The following function will not be duplicated if it is also called from the low- and high-priority interrupt functions.

```
#pragma interrupt_level 1
#pragma interrupt_level 2
int timestwo(int a) {
    return a * 2;
}
```

5.9 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.

5.9.1 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 PIC devices. This essentially performs a software Reset. Note that the state of registers after a software Reset can be different to that after a hardware Reset.

It is recommended that the `main()` function does not end. Add a loop construct (such as a `while(1)`) that will never terminate either around your code in `main()` or at the end of your code, so that execution of the function will never terminate. For example,

```
int main(void)
{
    // your code goes here
    // finished that, now just wait for interrupts
    while(1)
        continue;
}
```

5.9.2 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 global variables assigned a value when defined
- Clearing of non-initialized global variables
- General set up of registers or device state

Rather than the traditional method of linking in a generic, precompiled routine, MPLAB XC8 determines what runtime startup code is required from the user's program. Details of the files used and how the process can be controlled are described in 4.3.2 Startup and Initialization. The following sections detail the tasks performed by the runtime startup code.

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. If your program is an application invoked by a bootloader that will have already executed, you might need to ensure that data bank 0 is selected so that the runtime startup code executes correctly. This can be achieved by placing the appropriate code sequence towards the end of the bootloader as in-line assembly.

The following table lists the significant assembly labels used by the startup and powerup code.

Table 5-12. SIGNIFICANT ASSEMBLY LABELS

Label	Location
`reset_vec`	at the Reset vector location (0x0)
`powerup`	the beginning of the powerup routine, if used
`start`	the beginning of the runtime startup code, in startup.s
`start_initialization`	the beginning of the C initialization startup code, in the C output code

5.9.2.1 Initialization Of Objects

One task of the runtime startup code is to ensure that any static storage duration objects contain their initial value before the program begins execution. A case in point would be input in the following example.

```
int input = 88;
```

In the code above, the initial value (`0x88`) will be stored as data in program memory and will be copied to the memory reserved for `input` by the runtime startup code. For efficiency, initial values are stored as blocks of data and copied by loops.

Absolute variables are never initialized and must be explicitly assigned a value if that is required for correct program execution.

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

The runtime startup code that initializes objects will clobber the content of the STATUS register. If you need to determine the cause of reset from this register, the register content can be preserved.

Related Links

5.9.2.4 Status Register Preservation

5.9.2.2 Clearing Objects

Those objects with static storage duration which are not assigned a value must be cleared before the `main()` function begins by the runtime startup code, for example.

```
int output;
```

The runtime startup code will clear all the memory locations occupied by uninitialized objects so they will contain zero before `main()` is executed.

Absolute variables are never cleared and must be explicitly assigned a value of zero if that is required for correct program execution.

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.

The runtime startup code that clears objects will clobber the content of the STATUS register. If you need to use this register to determine the cause of reset, the register's content can be preserved.

Related Links

5.3.9.5 Persistent Type Qualifier

5.9.2.4 Status Register Preservation

5.9.2.3 Setup Of Device State

Some PIC devices come with an oscillator calibration constant which is pre-programmed into the device's program memory. Code is automatically placed in the runtime startup code to load this calibration value (see 5.2.11 Oscillator Calibration Constants).

If the software stack is being used by the program, the stack pointer (FSR1) is also initialized by the runtime startup code (see 5.4.2.2.1 Object Size Limits).

5.9.2.4 Status Register Preservation

The `-mresetbits` option (see 4.6.1.19 Resetbits Option) preserves some of the bits in the STATUS register before they are clobbered by the remainder of the program. The state of the bits within this saved register can then be examined at any subsequent time to determine the cause of Reset. This option is not available when compiling for PIC18 devices.

The STATUS register is saved to an 8-bit wide assembly variable `___resetbits`, although any bank selection bits in the register (if present for the target device) might not be accurately preserved. The compiler also defines the assembly symbols `___powerdown` and `___timeout` to represent the Power-down (PD) and Time-out (TO) bits within the STATUS register and which can be used in assembly code, if required. Check the runtime startup file `startup.s` to see these definitions and the compiler-generated code which stores the register.

The above symbols can also be accessed from C code once `<xc.h>` has been included. Note that the equivalent C identifiers will use just two leading underscore characters, e.g. `__resetbits`. See 5.11.3.3 Equivalent Assembly Symbols for more details of symbol mapping. The following Mid-range code example checks the state of the saved bits at the beginning of `main()` before proceeding. To determine the exact cause of Reset, you might also need to check the state of bits in other registers (such as PCON or PCON0). Check your device data sheet for all the recorded causes of Reset.

```
#include <xc.h>

int
main(void)
{
  // how did we get here?
  if(__timeout == 0 && __powerdown == 0)
    handleWDT_timeout();    // WDT wake-up from sleep
  // proceed with remaining code
}
```

The compiler will detect the usage of the above symbols in your code and automatically enable the `-mresetbits` option, if they are present. You may choose to enable this feature manually, if desired.

5.9.3 The Powerup Routine

Some hardware configurations require special initialization, often within the first few instruction cycles after Reset. To achieve this, there is a hook to the Reset vector provided via the powerup routine.

This routine can be supplied in a user-defined assembler module. A template powerup routine is provided in the file `powerup.S` which is located in the `pic/sources` directory of your compiler distribution. Refer to comments in this file for more details.

The file should be copied to your working directory, modified, and included into your project as a source file. No special linker options or other code is required. The compiler will detect if you have defined a powerup routine and will automatically execute this code after reset, provided the code is contained in a psect (section) called `powerup`.

For correct operation, the code must end with a `goto` instruction that jumps to the label called `start`. As with all user-defined assembly code, any code inside this file must take into consideration program memory paging and/or data memory banking, as well as any applicable errata issues for the device you are using.

5.10 Libraries

The MPLAB XC8 C Compiler provides libraries of functions, macros, types, and objects that can assist with your code development.

The compiler provides C language Standard libraries as well as some device-specific library resources. The libraries also contain C routines that are implicitly called by programs to perform tasks such as floating-point operations and integer division. These routines do not directly correspond to a function call in the source code. In addition, there are several library functions, such as functions relating to function-level profiling, mid-range EEPROM access, REALICE trace and log etc., that are built with the program as required. These will not be found in any library file.

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

5.10.1 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.

5.10.2 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 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`	`snprintf`	`sprintf`
`vprintf`	`vsnprintf`	`vsprintf`

The smart input (scan family) functions are:

`scanf`	`sscanf`	`vscanf`	`vsscanf`

Each time you build your project, the compiler analyzes your project's C source code, 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*.

5.10.2.1 Smart IO For PIC Devices

When using MPLAB XC8 for PIC MCUs, the code that processes the smart IO function format strings is defined in two source files (`doprnt.c` for input; `doscan.c` for output) and implements all the conversion specifiers, length modifiers, precision specifiers, field widths, and flags that can appear in these format strings. These features are implemented by individual sections of code that are conditionally included into the program image based on how you use the IO functions in your project. Tracking of the features used by smart output functions is conducted independently to that for smart input functions, but there is no other differentiation between functions, as the formatting code is shared between functions. That is, the code that processes the format strings used by `printf` functions for example is the same as that used by `vsnprintf`; only the code that writes to the destination is different.

If the format string in a call to an IO function is not a string literal, the compiler will not be able to detect which features of the IO function have been used. A more complete version of the formatting routine will be included in the program image; however, the compiler can still make certain assumptions. In particular, the number and type of the arguments following the format string can be used to determine which placeholders could be valid. This enables the size and complexity of the generated formatting routine to be kept to a minimum, even in these cases.

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

```
printf(myFormatString, 4, 6);
```

the compiler could at least determine from the type of the arguments that no floating-point placeholders are required for output functions, and so exclude those from the program image.

The analysis and customization of the smart IO library code is automatic. There are no options to control this operation. Those IO features linked into the program image can only be controlled by adjusting the format strings used by the smart IO functions in your program. Eliminating the use of floating-point specifiers and modifiers in calls to smart IO function in your program will make the largest reduction to code size.

Although the smart IO functions are regenerated from the original source code with each build, they are first packaged into a library file before being passed to the linker. This allows the functions to be replaced in the usual way by user-defined functions with the same name, if for example you wanted to define your own (non-smart) `printf()` function.

5.10.3 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, `xc8-ar`, (see 8.1 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.

5.10.4 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 9.1 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.

5.11 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.

5.11.1 Integrating Assembly Language Modules

Routines can be entirely encoded in assembly code. Ideally, the code for these should be placed in assembly source files. These can be included into your project, built alongside C source files, and combined into the program image by the linker.

Use a `.s` extension for source files containing plain assembly code, or `.S` for files that contain preprocessor directives that must be first processed by the preprocessor. Alternatively, you can use the `-xassembler-with-cpp` option (see 4.6.2.6 X: Specify Source Language Option) to force all assembly source files regardless of extension to be preprocessed.

By default, assembly modules are not optimized by the assembler; however, optimization can be enabled by using the `-fasmfile` option (see 4.6.6.7 Asmfile Option).

5.11.2 Inline Assembly

Assembly instructions can be directly embedded in-line into C code using the statement `asm();`.

The instructions are placed in a string inside what look like function call brackets, although no actual call takes place. Typically one instruction is placed in the string, but you can specify more than one assembly instruction by separating the instructions with a `\n` character, e.g., `asm("movlw 55\nmovwf _x");`, code will be more readable if you place one instruction in each statement and use multiple statements.

You can use the `asm()` form of in-line assembly at any point in the C source code as it will correctly interact with all C flow-of-control structures, as shown below.

```
unsigned int var;
int main(void)
{
    var = 1;
    asm("bcf 0,3");
    asm("BANKSEL _var");
    asm("rlf (_var)&07fh");
    asm("rlf (_var+1)&07fh");
}
```

In-line assembly code is never optimized by the assembler optimizer.

When using in-line assembler code, it is extremely important that you do not interact with compiler-generated code. The code generator cannot scan the assembler code for register usage; so it remains unaware if registers are clobbered or used by the assembly code. However, the compiler will reset all bank tracking once it encounters in-line assembly, so any SFRs or bits within SFRs that specify the current bank do not need to be preserved by in-line assembly.

The registers used by the compiler are explained in 5.6 Register Usage. If you are in doubt as to which registers are being used in surrounding code, compile your program with the `-Wa,-a` option and examine the assembler code generated by the compiler. Remember that as the rest of the program changes, the registers and code strategy used by the compiler will change as well.

If a C function is called from main-line and interrupt code, it can be duplicated (see 5.8.7 Function Duplication). Although a special prefix is used to ensure that labels generated by the compiler are not duplicated, this does not apply to labels defined in hand-written, in-line assembly code in C functions. Thus, you should not define assembly labels for in-lined assembly if the containing function might be duplicated.

5.11.3 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.

5.11.3.1 Writing Stand-alone Assembly Routines

It is possible to write assembly routines that can be called from C code. The following guidelines must be adhered to when writing a C-callable assembly routine.

- Include the `<xc.inc>` assembly header file if you need to use SFRs in your code. If this is included using `#include`, ensure the source file are preprocessed.
- Select, or define, a suitable psect for the executable assembly code (see 5.14.1 Compiler-Generated Psects for an introductory guide).
- Select a name (label) for the routine using a leading underscore character.
- Ensure that the routine's label is globally accessible from other modules.

- 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, write a reentrant routine if possible (see 5.11.3.2 Writing Reentrant Assembly Routines With Parameters); otherwise use ordinary variables for value passing.
- Optionally, use a signature value to enable type checking when the function is called.
- Use bank selection instructions and mask addresses of any variable symbols.

The following example shows a Mid-range device assembly routine that can add an 8-bit argument with the contents of PORTB and return this as an 8-bit quantity. The code is similar for other devices.

```
#include <xc.inc>
GLOBAL _add             ; make _add globally accessible
SIGNAT _add,4217        ; tell the linker how it should be called
; everything following will be placed into the code psect
PSECT code
; our routine to add to ints and return the result
_add:
    ; W is loaded by the calling function;
    BANKSEL (PORTB)               ; select the bank of this object
    addwf BANKMASK(PORTB),w    ; add parameter to port

    ; the result is already in the required location (W) so we can
    ; just return immediately
    return
```

The code has been placed in a predefined psect, `code`, that is available after including `<xc.inc>`. This section is part of the `CODE` linker class, so it will be automatically placed in the area of memory set aside for code without you having to adjust the default linker options. This section can be used by any device to hold executable code.

If you prefer to create your own psect, you could use the following for a Mid-range device:

```
PSECT mytext,local,class=CODE,delta=2
```

The `delta` flag used with this section indicates that the memory space in which the psect will be placed is word addressable (value of 2), which is true for PIC10/12/16 devices. For PIC18 devices, program memory is byte addressable, but instructions must be word-aligned, so you would instead use a section definition such as the following, which uses a `delta` value of 1 (which is the default setting), but the `reloc` (alignment) flag is set to 2, to ensure that the section starts on a word-aligned address.

```
PSECT text0,class=CODE,reloc=2
```

See 6.1.9.36 Psect Directive for detailed information on the flags used with the `PSECT` assembler directive.

The mapping between C identifiers and those used by assembly are described in 5.11.3 Interaction between Assembly and C Code. In assembly domain we must choose the routine name `_add` as this then maps to the C identifier add. Since this routine will be called from other modules, the label is made globally accessible, by using the `GLOBAL` assembler directive (6.1.9.21 Global Directive).

A `SIGNAT` directive (6.1.9.40 Signat Directive) was used so that the linker can check that the routine is correctly called.

The W register will be used for passing in the argument. See 5.7.5 Function Parameters for the convention used to pass parameters.

The `BANKSEL` directive and `BANKMASK` macro have been used to ensure that the correct bank was selected and that all addresses are masked to the appropriate size.

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 that the call from C code is correct
extern unsigned char add(unsigned char a);

int main(void) {
    volatile unsigned char result;
```

```
    result = add(5);  // call the assembly routine
}
```

5.11.3.2 Writing Reentrant Assembly Routines With Parameters

Hand-written assembly routines for Enhanced Mid-range and PIC18 devices can be written to use the software stack and be reentrantly called from C code. Such routines can take parameters, return values, and define their own local objects, if required.

The following are the steps that need to be followed to create such routines.

1. Declare the C prototype for the routine in C source code, choosing appropriate parameter and return value types.
2. Include the `<xc.inc>` assembly header file. If this is included using `#include`, ensure the source file uses `.S` as its extension.
3. If required, define each auto-like variable using the `stack_auto` *`name`*`,`*`size`* macro, where *`name`* can be any valid assembler identifier and *`size`* is the variable's size in bytes.
4. If required, define each parameter using the macro `stack_param` *`name`*`,`*`size`*, where *`name`* can be any valid assembly identifier and *`size`* is the variable's size in bytes. Parameters must be defined after autos and their order must match the order in which they appear in the C prototype.
5. Initialize the stack once using the macro `alloc_stack` before any instructions in the routine.
6. Immediately before each return instruction, restore the stack using the macro `restore_stack`

Write the routine in assembly in the usual way, taking note of the points in 5.11.1 Integrating Assembly Language Modules.

Each auto and parameter variable will be located at a unique offset to the stack pointer (FSR1). If you follow the above guidelines, you can use the symbol `name_offset`, which will be assigned the stack-pointer offset for the variable with name. These macros will exist for both auto and parameter variables.

If the routine returns a value, this must be placed into the location expected by the code that calls the routine (for full details of C-callable routines, see 5.7.6.2 Software Stack Return Values). To summarize, for objects 1 to 4 bytes in size, these must be loaded to temporary variables referenced as `btemp`, plus an offset. This symbol is automatically linked into your routine if you use the macros described above.

It is recommended that you do not arbitrarily adjust the stack pointer during the routine. The symbols that define the offset for each auto and parameter variable assume that the stack pointer has not been modified. However, if your assembly routine calls other reentrant routines (regardless of whether they are defined in C or assembly code), you must write the assembly code that pushes the arguments onto the stack, calls the function and then removes any return value from the stack.

The following is an example of a reentrant assembly routine, `_inc`, written for a PIC16F1xxx device. Its arguments and return value are described by the C prototype:

```
extern reentrant int inc(int foo);
```

This routine returns an `int` value that is one higher than the `int` argument that is passed to it. It uses an auto variable, `x`, strictly for illustrative purposes.

```
#include <xc.inc>
PSECT text2,local,class=CODE,delta=2
GLOBAL _inc
_inc:
    stack_auto x,2      ;an auto called 'x'; 2 bytes wide
    stack_param foo,2   ;a parameter called 'foo'; 2 bytes wide
    alloc_stack
    ;x = foo + 1;
    moviw   [foo_offset+0]FSR1
    addlw   low(01h)
    movwf   btemp+0
    moviw   [foo_offset+1]FSR1
    movwf   btemp+1
    movlw   high(01h)
    addwfc  btemp+1,f
    movf    btemp+0,w
    movwi   [x_offset+0]FSR1
```

```
    movf    btemp+1,w
    movwi   [x_offset+1]FSR1
    ;return x;
    moviw   [x_offset+0]FSR1
    movwf   btemp+0
    moviw   [x_offset+1]FSR1
    movwf   btemp+1
    restore_stack
    return
```

The following is an example of a reentrant assembly routine, `_add`, written for a PIC18 device. Its arguments and return value are described by the C prototype:

```
extern reentrant int add(int base, int index);
```

This routine returns an `int` value that is one higher than the `int` sum of the `base` and `index` arguments that are passed to it. It uses the auto variables, `tmp` and `result`, strictly for illustrative purposes.

```
#include <xc.inc>
psect        text1,class=CODE,space=0,reloc=2
GLOBAL _add
_add:
    stack_auto tmp,2      ;an auto called 'tmp'; 2 bytes wide
    stack_auto result,2   ;an auto called 'result'; 2 bytes wide
    stack_param base,2    ;a parameter called 'base'; 2 bytes wide
    stack_param index,2   ;a parameter called 'index'; 2 bytes wide
    alloc_stack
    ;tmp = base + index;
    movlw   base_offset
    movff   PLUSW1,btemp+0
    movlw   base_offset+1
    movff   PLUSW1,btemp+1
    movlw   index_offset
    movf    PLUSW1,w,c
    addwf   btemp+0,f,c
    movlw   index_offset+1
    movf    PLUSW1,w,c
    addwfc  btemp+1,f,c
    movlw   tmp_offset
    movff   btemp+0,PLUSW1
    movlw   tmp_offset+1
    movff   btemp+1,PLUSW1
    ;result = tmp + 1;
    movlw   tmp_offset
    movf    PLUSW1,w,c
    addlw   1
    movwf   btemp+0,c
    movlw   tmp_offset+1
    movff   PLUSW1,btemp+1
    movlw   0
    addwfc  btemp+1,f,c
    movlw   result_offset
    movff   btemp+0,PLUSW1
    movlw   result_offset+1
    movff   btemp+1,PLUSW1
    ;return result;
    movlw   result_offset
    movff   PLUSW1,btemp+0
    movlw   result_offset+1
    movff   PLUSW1,btemp+1
    restore_stack
    return
```

5.11.3.3 Equivalent Assembly Symbols

Most C symbols map to an corresponding assembly equivalent.

This mapping is such that an “ordinary” symbol defined in the assembly domain cannot interfere with an “ordinary” symbol in the C domain. So for example, if the symbol `main` is defined in the assembly domain, it is quite distinct to the `main` symbol used in C code and they refer to different locations.

The name of a C function maps to an assembly label that will have the same name, but with an underscore prepended. So the function `main()` will define an assembly label `_main`.

Baseline PIC devices can use alternate assembly domain symbols for functions. The destinations of call instructions on these devices are limited to the first half of a program memory page. The compiler, thus, encodes functions in two parts, as illustrated in the following example of a C function, `add()`, compiled for a Baseline device.

```
entry__add:
ljmp     _add
```

The label `entry__add` is the function's entry point and will always be located in the first half of a program memory page. The code associated with this label is simply a long jump (see 6.1.1.7 Long Jumps And Calls) to the actual function body located elsewhere and identified by the label `_add`.

If you plan to call Baseline routines from assembly code, you must be aware of this limitation in the device and the way the compiler works around it for C functions. Hand-written assembly code should always call the `entry__`*`funcName`* label rather than the usual assembly-equivalent function label.

If a C function is qualified `static` and there is more than one static function in the program with the exact same name, the name of the first function will map to the usual assembly symbol and the subsequent functions will map to a special symbol with the form: `_`*`functionName`*`@`*`fileName`*`$F`*`number`*, where *`functionName`* is the name of the function, *`fileName`* is the name of the file that contains the function, and *`number`* is a unique number sequence. The definition of "first" in this situation is complex. Typically, if the symbol is contained in the source module that defines `main()`, it will be processed first. If it is not in this module, then the order in which the source files are listed on the compiler command line determines which is considered first.

For example, a program contains the definition for two `static` functions, both called `add()`. One lives in the file `main.c` and the other in `lcd.c`. The first function will generate an assembly label `_add`. The second might generate the label `_add@lcd$F38`, for example.

The name of a C variable with static storage duration also maps to an assembler label that will have the same name, but with an underscore prepended. So the variable `result` will define an assembly label: `_result`.

If the C variable is qualified `static`, there is a chance that there could be more than one variable in the program with exactly the same C name. The rules that apply to `static` variables defined outside of functions are similar to those that apply to `static` functions. The name of the first variable will map to a symbol prepended with an underscore; the subsequent symbols will have the form: `_`*`variableName`*`@`*`fileName`*`$F`*`number`*, where *`variableName`* is the name of the variable, *`fileName`* is the name of the file that contains the variable, and *`number`* is a unique number sequence.

All local `static` variables (i.e., defined inside a function definition) have an assembly name of the form: *`functionName`*`@`*`variableName`*. If there is a `static` variable called `output` in the function `read()` and another `static` variable with the same name defined in the function `update()`, then the symbols in the assembly can be accessed using the symbols `read@output` and `update@output`, respectively.

Functions that use the reentrant model do not define any symbols that allow you to access auto and parameter variables. You should not attempt to access these in assembly code. Special symbols for these variables are defined, however, by functions that use the nonreentrant model. These symbols are described in the following paragraphs.

To allow easy access to parameter and auto variables on the compiled stack, special equates are defined which map a unique symbol to each variable. The symbol has the form: *`functionName`*`@`*`variableName`*. Thus, if the function `main()` defines an auto variable called `foobar`, the symbol `main@foobar` can be used in assembly code to access this C variable.

Function parameters use the same symbol mapping as auto variables. If a function called `read()` has a parameter called `channel`, then the assembly symbol for that parameter is `read@channel`.

Function return values have no C identifier associated with them. The return value for a function shares the same memory as that function's parameter variables, if they are present. The assembly symbol used for return values has the form `?_`*`functionName`*, where *`functionName`* is the name of the function returning the value. Thus, if a function, `getPort()` returns a value, it will be located the address held by the assembly symbol `?_getPort`. If this return value is more than one byte in size, then an offset is added to the symbol to access each byte, e.g., `?_getPort+1`.

If the compiler creates temporary variables to hold intermediate results, these will behave like auto variables. As there is no corresponding C variable, the assembly symbol is based on the symbol that represents the auto block for the function plus an offset. That symbol is `??_functionName`, where *`functionName`* is the function in which the symbol is being used. So for example, if the function `main()` uses temporary variables, they will be accessed as an offset from the symbol `??_main`.

5.11.3.4 Accessing Registers From Assembly Code

In separate assembly modules, SFR definitions are not automatically accessible. The assembly header file `<xc.inc>` can be used to gain access to these register definitions. Do not use this file for assembly in-line with C code as it will clash with definitions in `<xc.h>`.

Include the file using the assembler's `INCLUDE` directive, (see 6.1.9.23 Include Directive) or use the C preprocessor's `#include` directive. If you are using the latter method, make sure you use a `.S` extension for the assembly source file.

The symbols for registers in this header file look similar to the identifiers used in the C domain when including `<xc.h>`, e.g., `PORTA`, `EECON1`, etc. They are different symbols in different domains, but will map to the same memory location.

Names of bits within registers are defined as *`registerName,bitNumber`*. So, for example, `RA0` is defined as `PORTA,0`.

Here is an example of a Mid-range assembly module that uses SFRs.

```
#include <xc.inc>
GLOBAL _setports
PSECT text,class=CODE,local,delta=2
_setports:
    movlw 0xAA
    BANKSEL (PORTA)
    movwf BANKMASK(PORTA)
    BANKSEL (PORTB)
    bsf RB1
```

If you wish to access register definitions from assembly that is in-line with C code, ensure that the `<xc.h>` header is included into the C module. Information included by this header will define in-line assembly symbols as well as the usual symbols accessible from C code.

The symbols used for register names will be the same as those defined by `<xc.inc>`. So for example, the example given previously could be rewritten as in-line assembly as follows.

```
#include <xc.h>

int main(void) {
        asm("movlw 0xAA");
        asm("BANKSEL (PORTA)");
        asm("movwf " ___mkstr(BANKMASK(PORTA)));
        asm("BANKSEL (PORTB)");
        asm("bsf " ___mkstr(BANKMASK(PORTB)) "," ___mkstr(_PORTB_RB1_POSN));
}
```

Note that `BANKMASK()` is a preprocessor macro, and thus it will not be expanded inside a C string literal. Convert the macro's replacement to a string using the `___mkstr()` macro (note: *three* leading underscore characters), defined once you include `<xc.h>`, if you want to use the `BANKMASK()` macro within in-line assembly. The macro which defines RB1's bit position is similarly expanded.

The code generator does not detect when SFRs have been written to in in-line assembly, so these must be preserved. The list of registers used by the compiler and further information can be found in 5.6 Register Usage.

5.11.3.5 Absolute Psects

MPLAB XC8 is able to determine the address bounds of absolute psects (defined using the `abs` and `ovrld`, `PSECT` flags) and reserves that data memory prior to the compilation stage so it is not used by C source. Any data memory required by assembly code must use an absolute psect, but these do not need to be used for psects to be located in program memory.

The following example code contained in an assembly code file defines a table that must be located at address 0x110 in the data space.

```
PSECT lkuptbl,class=RAM,space=1,abs,ovrld
ORG 110h
lookup:
    DS 20h
```

When the project is compiled, the memory range from address 0x110 to 0x12F in memory space 1 (data memory) is recorded as being used and is reserved before compiling the C source. An absolute psect always starts at address 0. For such psects, you can specify a non-zero starting address by using the `ORG` directive. See 6.1.9.30 Org Directive for important information on this directive.

5.11.3.6 Undefined Symbols

If an object is defined in C source code, but is only accessed in assembly code, the compiler might ordinarily remove the object believing it is unused, resulting in an undefined symbol error.

To work around this issue, MPLAB XC8 searches for symbols in assembly code that have no assembly definition (which would typically be a label). If these symbols are encountered in C source they are automatically treated as being `volatile` (see 5.3.8.2 Volatile Type Qualifier), which will prevent them from being removed.

For example, if a C program defines a variable as follows:

```
int input;
```

but this variable is never used in C code. The assembly module(s) can simply declare this symbol, with the leading underscore character (see 5.11.3 Interaction between Assembly and C Code), using the `GLOBAL` assembler directive and then use it as follows.

```
GLOBAL _input, _raster
PSECT text,local,class=CODE,reloc=2
_raster:
    movf   _input,w
```

5.12 Optimizations

The optimizations in the MPLAB XC8 compiler can be broadly grouped into C-level optimizations performed on the source code before conversion into assembly and assembly-level optimizations performed on the assembly code generated by the compiler.

The C-level optimizations are performed early during the code generation phase and so have flow-on benefits: performing one optimization might mean that another can then be applied. As these optimizations are applied before the debug information has been produced, they have less impact on source-level debugging of programs.

Some of these optimizations are integral to the code generation process and so cannot be disabled via an option. Suggestions as to how specific optimizations can be defeated are given in the sections below.

If your compiler is unlicensed, some of the optimization levels are disabled (see 4.6.6 Options for Controlling Optimization). Even if they are enabled, optimizations can 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.

The optimization level determines the available optimizations, which are listed in the Table 5-13 table.

Table 5-13. Optimization Level Sets

Level	Optimization sets available
`O0`	• Rudimentary Optimization
`O1`	• Minimal code generator optimizations
`O2`	• All generic code generator optimizations • Minimal assembly optimizations

...........continued	
Level	**Optimization sets available**
`O3` (Licensed only)	• All generic and speed-specific code generator optimizations • All generic and speed-specific assembler optimizations
`Os` (Licensed only)	• All generic and space-specific code generator optimizations • All generic and space-specific assembler optimizations

Assembly-level optimizations are described in 6.2 Assembly-Level Optimizations.

The minimal code generator optimizations consist of the following.

- Whole-program analysis for object allocation into data banks without having to use non-standard keywords or compiler directives.
- 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 can be determined. 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 code generation (C-level) optimizations, which simplify C expressions or code produced from C expressions. These are applied across the entire program, not just on a module-by-module basis.

- Tracking of the current data bank is performed by the compiler as it generates assembly code. This allows the compiler to reduce the number of bank-selection instructions generated.
- 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. Global objects qualified `volatile` will never be removed (see 5.3.8.2 Volatile Type Qualifier). 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.
- Unused return expressions in a function are removed. The return value is considered unused if the result of all calls to that function discard the return value. The code associated with calculation of the return value will be removed and the function will be encoded as if its return type was `void`.
- Variables assigned a value before being read are not cleared or initialized by the runtime startup code. Only non-auto variables are considered and if they are assigned a value before other code can read their value, they are treated as being `__persistent` (see)5.3.9.1 Bank Type Qualifier. All `__persistent` objects are not cleared by the runtime startup code, so this optimization will speed execution of the program startup.
- Pointer sizes are optimized to suit the target objects they can access. The compiler tracks all assignments to pointer variables and keeps a list of targets each pointer can access. As the memory space of each target is known, the size and dereference method used can be customized for each pointer.

- Dereferencing pointers with only target can be replaced with direct access of the target object. This applies to data and function pointers.
- Objects qualified `const` are considered for placement into data memory if any such memory is otherwise unused by the program.

MPLAB X IDE or other IDEs can indicate incorrect values when watching variables if optimizations hold a variable in a register. Try to use the ELF/DWARF debug file format to minimize such occurrences. Check the assembly list file to see if registers are used in the routine that is being debugged.

5.13 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 4.6.2.2 E: Preprocess Only).

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

5.13.1 Preprocessor Directives

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

Table 5-14. Preprocessor Directives

Directive	Meaning	Example
`#`	Preprocessor null directive, do nothing.	`#`

`#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`

...........continued		
Directive	**Meaning**	**Example**
`#nn filename`	(where *nn* is a number, and *filename* is the name of the source file) the following content originated from the specified file and line number.	`#20 init.c`
`#pragma`	Compiler specific options.	See the Pragma Directives section in this guide.
`#undef`	Undefines preprocessor symbol.	`#undef FLAG`
`#warning`	Generate a warning message.	`#warning Length not set`

Macro expansion using arguments can use the `#` character to convert an argument to a string and the `##` sequence to concatenate arguments. If two expressions are being concatenated, consider using two macros in case either expression requires substitution itself; for example

```
#define __paste1(a,b)    a##b
#define __paste(a,b)     __paste1(a,b)
```

lets you use the `paste` macro to concatenate two expressions that themselves can require further expansion. Remember, that once a macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

5.13.1.1 Preprocessor Arithmetic

Preprocessor macro replacement expressions are textual and do not utilize types. Unless they are part of the controlling expression to the inclusion directives (discussed below), macros are not evaluated by the preprocessor. Once macros have been textually expanded and preprocessing is complete, the expansion forms a C expression which is evaluated by the code generator along with other C code. Tokens within the expanded C expression inherit a type, with values then subject to integral promotion and type conversion in the usual way.

If a macro is part of the controlling expression to a conditional inclusion directive (`#if` or `#elif`), the macro must be evaluated by the preprocessor. The result of this evaluation is often different to the C-domain result for the same sequence. The preprocessor assigns sizes to literal values in the controlling expression that are equal to the largest integer size accepted by the compiler, as specified by the size of `intmax_t` defined in `<stdint.h>`.

For the MPLAB XC8 C compiler, this size is 32 bits, unless you are compiling for a PIC18 device with C99 in which case it is 64 bits.

The following code might not work as expected. The preprocessor will evaluate `MAX` to be the result of a 32-bit multiplication, 0xF4240. However, the definition of the `long int` variable, `max`, will be assigned the value 0x4240 (since the constant 1000 has a `signed int` type, and therefore the C-domain multiplication will also be performed using a 16-bit `signed int` type).

```
#define MAX 1000*1000
...
#if MAX > INT16_MAX       // evaluation of MAX by preprocessor
    long int max = MAX;   // evaluation of MAX by code generator
#else
    int max = MAX;        // evaluation of MAX by code generator
#endif
```

Overflow in the C domain can be avoided by using a constant suffix in the macro (see 5.3.7 Constant Types and Formats). For example, an `L` after a number in a macro expansion indicates it should be interpreted by the C compiler as a `long`, but this suffix does not affect how the preprocessor interprets the value, if it needs to evaluate it.

For example, the following will evaluate to 0xF4240 in C expressions.

```
#define MAX 1000*1000L
```

5.13.2 Predefined Macros

The compiler drivers define certain symbols to the preprocessor, allowing conditional compilation based on chip type, etc. The symbols listed in the table below show the more common symbols defined by the drivers.

Each symbol, if defined, is equated to 1 (unless otherwise stated).

Table 5-15. Predefined Macros

Symbol	Set
`ERRATA_4000_BOUNDARY`	When the ERRATA_4000 applies.
`HI_TECH_C`	When the C language variety is HI-TECH C compatible.
`MPLAB_ICD`	When building for a non-PIC18 device and an MPLAB ICD debugger. Assigned 2 to indicate an MPLAB ICD 2, assigned 3 for the MPLAB ICD 3.
`_CHIPNAME`	When the specific chip type selected, e.g., `_16F877`.
`_BANKBITS_`	When building for non-PIC18 devices. Assigned 0, 1, or 2 to indicate 1, 2, or 4 available banks or RAM.
`_BANKCOUNT_`	When building for non-PIC18 devices. Indicates the number of banks of data memory implemented.
`_COMMON_`	When common RAM is present.
`_COMMON_ADDR_`	When common memory is present. Indicates common memory starting address.
`_COMMON_SIZE_`	When common memory is present. Indicates the common memory size.
`_EEPROMSIZE`	When building for non-PIC18 devices. Indicates how many bytes of EEPROM are available.
`_EEPROM_INT`	When building for non-PIC18 devices. Assigned a value of 2 (`_NVMREG_INT`), 1 (`_EEREG_INT`), or 0 (`_NOREG_INT`) to indicate the device uses the NVMREG, EEREG, or no register interface to access EEPROM.
`_ERRATA_TYPES`	When the errata workaround is being applied, see `-merrata` option, 4.6.1.10 Errata Option.
`_`*`FAMILY`*`_FAMILY_`	When building for PIC18 devices and indicates the PIC18 family as indicated by the `FAMILY` field in the relevant `.ini` file in the compiler's `pic/dat/ini` directory.
`_FLASH_ERASE_SIZE`	Always. Indicates the size of the Flash program memory erase block. They do not represent Flash data memory.
`_FLASH_WRITE_SIZE`	Always. Indicates the size of the Flash program memory write block. They do not represent Flash data memory.
`_GPRBITS_`	When building for non-PIC18 devices. Assigned 0, 1, or 2 to indicate 1, 2, or 4 available banks or general purpose RAM.
`_GPRCOUNT_`	When building for non-PIC18 devices. Assigned a value which indicates the number of banks that contain general-purpose RAM.
`_HAS_FUNCTIONLEVELPROF_`	When `-finstrument-functions` is specified and target supports profiling.
`_HAS_INT24`	Always.
`_HAS_OSCVAL_`	When the target device has an oscillator calibration register.
`_MPC_`	When compiling for Microchip PIC MCU family.
`_OMNI_CODE_`	When compiling using an OCG compiler.
`_PIC12`	When building for a Baseline device (12-bit instruction).

...........continued	
Symbol	**Set**
_PIC12E	When building for an Enhanced Baseline device (12-bit instruction).
_PIC12IE	When building for am Enhanced Baseline device with interrupts.
_PIC14	When building for an Mid-range device (14-bit instruction).
_PIC14E	When building for an Enhanced Mid-range device (14-bit instruction).
_PIC14EX	When building for an extended-bank Enhanced Mid-range PIC device (14-bit instruction).
_PIC18	When building for a PIC18 device (16-bit instruction).
PROGMEM	When building for a Mid-range device with flash memory, and indicates the type of flash memory employed by the target device: values 0xFF (unknown) 0xF0 (none) 0 (read-only) 1 (word write with auto erase) 2 (block write with auto erase) 3 (block write with manual erase).
_RAMSIZE	When building for a PIC18 device. Indicates how many bytes of data memory are available.
_ROMSIZE	Always. Indicates how much program memory is available (byte units for PIC18 devices; words for other devices).
__CHIPNAME and __CHIPNAME__	When the specific chip type selected, e.g., __16F877.
__CLANG__	When the Clang frontend is in use (`-std=c99`).
__CODECOV	Always, with value `__CC_NONE` (0) or `__CC_RAM` (1).
__DATABANK	When eeprom or flash memory is implemented, and identifies in which bank the EEDATA/PMDATA register is found.
__DATE__	Always. Indicates the current date as a string literal, e.g., `"May 21 2004"`.
__DEBUG	When performing a debug build and you are using the MPLAB X IDE.
__EXTMEM	When device has external memory. Indicates the size of this memory.
__FILE__	Always. Indicates the source file being preprocessed.
__FLASHTYPE	When building for non-PIC18 devices with flash memory. Indicates the type of flash memory employed by the target device, see `_PROGMEM` below.
__LINE__	Always. Indicates this source line number.
__J_PART	When building for a PIC18 ‘J’ series part.
__MPLAB_ICDX__	When compiling for an ICD debugger. *X* can be `2`, `3`, or `4` indicating a Microchip MPLAB ICD 2, ICD 3, or ICD 4, respectively.
__MPLAB_ICE__	To indicate any In-circuit Emulator
__MPLAB_PICKITX__	When compiling for a PICkit™. *X* can be `2`, `3`, or `4` indicating a Microchip MPLAB PICkit 2, PICkit 3, or PICkit 4, respectively.
__MPLAB_REALICE__	When compiling for a Microchip MPLAB REAL ICE™ In-Circuit Emulator.
__MPLAB_SNAP__	When compiling for a Microchip MPLAB Snap In-Circuit Debugger.

...........continued

Symbol	Set
`__OPTIMIZE_SPEED__`	When using speed-orientated optimizations.
`__OPTIMIZE_SPACE__` and `__OPTIMIZE_SIZE__`	When using space-orientated optimizations.
`__OPTIMIZE_NONE__`	When no optimizations are in effect.
`__OPTIM_FLAGS`	Always. Indicates the optimizations in effect (see text following this table).
`__PICCPRO__` and `__PICC__`	When building for any PIC10/12/14/16 device.
`__PICC18__`	When not in C18 compatibility mode.
`__RESETBITS_ADDR`	When the STATUS register will be preserved. Indicates the address at which this register will be saved.
`__SIZEOF_TYPE__`	Always. Indicates the size in bytes of the specified type, e.g., `__SIZEOF_INT__` or `__SIZEOF___INT24__`.
`__STACK`	Always. Assigned with `__STACK_COMPILED` (0x1), `__STACK_HYBRID` (0x2) or `__STACK_REENTRANT` (0x4) to indicate the global stack setting: compiled, hybrid or software, respectively.
`__STRICT`	When the `-Wpedantic` option is enabled.
`__TIME__`	Always. Indicates the current time as a string literal, e.g., “08:06:31”
`__TRADITIONAL18__`	When building for a PIC18 device. Indicates the non-extended instruction set is selected.
`__XC`	Always. Indicates MPLAB XC compiler for Microchip is in use.
`__XC8`	Always. Indicates MPLAB XC compiler for Microchip 8-bit devices is in use.
`__XC8_VERSION`	Always, and indicates the compiler’s version number multiplied by 1000, e.g., v1.00 will be represented by 1000.

The compiler-defined macros shown in the Table 5-16 table can be used as bitwise AND masks to determine the value held by `__OPTIM_FLAGS`, hence the optimizations used.

Table 5-16. Optimization Flags

Macro	Value	Meaning
`__OPTIM_NONE`	0x0	No optimizations applied (on equality).
`__OPTIM_ASM`	0x1	Assembler optimizations on C code.
`__OPTIM_ASMFILE`	0x2	Assembler optimizations on assembly source code.
`__OPTIM_SPEED`	0x20000	Optimized for speed.
`__OPTIM_SPACE`	0x40000	Optimized for size.
`__OPTIM_SIZE`	0x40000	Optimized for size.
`__OPTIM_DEBUG`	0x80000	Optimized for accurate debug.
`__OPTIM_LOCAL`	0x200000	Local optimizations applied.

5.13.3 Pragma Directives

There are certain compile-time directives that can be used to modify the behavior of the compiler. These are implemented through the use of the C Standard's `#pragma` facility. The format of a pragma is:

```
#pragma directive options
```

where *`directive`* is one of a set of keywords, some of which are followed by `options`. A list of the keywords is given in the Table 5-17 table. Those keywords not discussed elsewhere are detailed below.

Table 5-17. Pragma Directives

Directive	Meaning	Example
`addrqual`	Specify action of qualifiers.	`#pragma addrqual require`
`config`	Specify configuration bits.	`#pragma config WDT=ON`
`inline`	Inline function if possible.	`#pragma inline(getPort)`
`intrinsic`	Specify function is inline.	`#pragma intrinsic(_delay)`
`interrupt_level`	Allow call from interrupt and main-line code.	`#pragma interrupt_level 1`
`pack`	Specify structure packing.	`#pragma pack 1`
`printf_check`	Enable printf-style format string checking.	`#pragma printf_check(printf) const`
`psect`	Rename compiler-generated psect.	`#pragma psect nvBANK0=my_nvram`
`regsused`	Specify registers used by function.	`#pragma regsused myFunc wreg,fsr`
`switch`	Specify code generation for switch statements.	`#pragma switch direct`
`warning`	Control messaging parameters.	`#pragma warning disable 299,407`

5.13.3.1 The #pragma Addrqual Directive

This directive allows you to control the compiler's response to non-standard memory qualifiers. This pragma is an in-code equivalent to the `-maddrqual` option and both use the same arguments (see 4.6.1.1 Addrqual Option).

The pragma has an effect over the entire C program and should be issued once, if required. If the pragma is issued more than once, the last pragma determines the compiler's response.

For example:

```
#pragma addrqual require
__bank(2) int foobar;
```

5.13.3.2 The #pragma Config Directive

This directive allows the device Configuration bits to be specified for PIC18 target devices. See 5.2.5 Configuration Bit Access for full details.

5.13.3.3 The #pragma Inline Directive

The `#pragma inline` directive indicates to the compiler that calls to the specified function should be as fast as possible. This pragma has the same effect as using the `inline` function specifier.

5.13.3.4 The #pragma Intrinsic Directive

The `#pragma intrinsic` directive is used to indicate to the compiler that a function will be inlined intrinsically by the compiler. This directive should never be used with user-defined code.

You should not attempt to redefine an existing library function that uses the `intrinsic` pragma. If you need to develop your own version of such a routine, it must not use the same name as the intrinsic function. For example, if you need to develop your own version of `memcpy()`, give this a unique name, such as `sp_memcpy()`. Check the standard header files to determine which library functions use this pragma.

5.13.3.5 The #pragma Interrupt_level Directive

The `#pragma interrupt_level` directive can be used to prevent duplication of functions called from main-line and interrupt code (see 5.8.7.1 Disabling Duplication).

5.13.3.6 The #pragma Pack Directive

All 8-bit PIC devices can only perform byte accesses to memory and so do not require any alignment of memory objects within structures. This pragma will have no effect when used.

5.13.3.7 The #pragma Printf_check Directive

Certain library functions accept a format string followed by a variable number of arguments in the manner of `printf()`. Although the format string is interpreted at runtime, it can be compile-time checked for consistency with the remaining arguments.

This directive enables this checking for the named function, for example the system header file `<stdio.h>` includes the directive:

```
#pragma printf_check(printf) const
```

to enable this checking for `printf()`. You can also use this for any user-defined function that accepts printf-style format strings.

The qualifier following the function name is to allow automatic conversion of pointers in variable argument lists. The above example would cast any pointers to strings in RAM to be pointers of the type (`const char *`).

Note that the warning level must be set to -1 or below for this option to have any visible effect (see 4.6.4.2 Warn Option).

5.13.3.8 The #pragma Psect Directive

The `#pragma psect` was used to redirect objects and functions to a new psect (section). It has been replaced by the `__section()` specifier (see 5.14.3 Changing and Linking the Allocated Section), which not only performs the same task, but is easier to use, and has fewer restrictions as to where the psects can be linked. It is recommended you always use the `__section()` specifier to location variables and function in unique psects.

The general form of this pragma looks like:

```
#pragma psect standardPsect=newPsect
```

and instructs the code generator that anything that would normally appear in the standard psect `standardPsect`, will now appear in a new psect called `newPsect`. This psect will be identical to `standardPsect` in terms of its flags and attributes; however, it will have a unique name. Thus, you can explicitly position this new psect without affecting the placement of anything in the original psect.

If the name of the standard psect that is being redirected contains a counter (e.g., `text0`, `text1`, `text2`, etc.), the placeholder `%%u` should be used in the name of the psect at the position of the counter, e.g., `text%%u`.

5.13.3.9 The #pragma Regsused Directive

The `#pragma regsused` directive allows the programmer to indicate register usage for functions that will not be "seen" by the code generator; for example, if they were written in assembly code. It has no effect when used with functions defined in C code, but in those cases the register usage of these functions can be accurately determined by the compiler and the pragma is not required. This pragma has no effect when used with devices that can save registers into shadow registers.

The general form of the pragma is:

```
#pragma regsused routineName registerList
```

where *`routineName`* is the C-equivalent name of the function or routine whose register usage is being defined, and *`registerList`* is a space-separated list of registers' names, as shown in the Table 5-10 table.

Those registers that are not listed are assumed to be unused by the function or routine. The code generator can use any of these registers to hold values across a function call. If the routine does in fact use these registers, unreliable program execution can happen.

The register names are not case sensitive and a warning will be produced if the register name is not recognized. An empty register list indicates that the specified function or routine uses no registers. If this pragma is not used, the compiler will assume that the external function uses all registers.

For example, a routine called _search is written in PIC18 assembly code. In the C source, we can write:

```
extern void search(void);
#pragma regsused search wreg status fsr0
```

to indicate that this routine used the W register, STATUS and FSR0. Here, `fsr0` expands to both FSR0L and FSR0H. These could be listed individually, if required.

The compiler can determine those registers and objects that need to be saved by an interrupt function, so this pragma could be used, for example, to allow you to customize the context switch code in cases where an interrupt routine calls an assembly routine.

5.13.3.10 The #pragma Switch Directive

Normally, the compiler encodes `switch` statements to produce the smallest possible code size. The `#pragma switch` directive can be used to force the compiler to use a different coding strategy.

The switch pragma affects all subsequent code and has the general form:

```
#pragma switch switchType
```

where *`switchType`* is one of the available selections that are listed in the Table 5-18 tables. The only switch type currently implemented for PIC18 devices is space.

Table 5-18. Switch Types

Switch Type	Description
`speed`	Use the fastest switch method.
`space`	Use the smallest code size method.
`time`	Use a fixed delay switch method.
`auto`	Use smallest code size method (default).
`direct` (deprecated)	Use a fixed delay switch method.
`simple` (deprecated)	Sequential xor method.

Specifying the `time` option to the `#pragma switch` directive forces the compiler to generate a table look-up style switch method. The time taken to execute each case is the same, so this is useful where timing is an issue, e.g., with state machines.

The `auto` option can be used to revert to the default behavior.

Information is printed in the assembly list file for each `switch` statement, showing the chosen strategy (see 6.3.4 Switch Statement Information).

5.13.3.11 The #pragma Warning Directive

This pragma allows control over some of the compiler's messages, such as warnings and errors. The pragmas have no effect when the Clang front end is used to compile C99 projects. For full information on the messaging system employed by the compiler see 4.5 Compiler Messages.

5.13.3.11.1 The Warning Disable Pragma

Some warning messages can be disabled by using the `warning disable` pragma.

This pragma will only affect warnings that are produced by the parser or the code generator; i.e., errors directly associated with C code. The position of the pragma is only significant for the parser; i.e., a parser warning number can be disabled for one section of the code to target specific instances of the warning. Specific instances of a warning produced by the code generator cannot be individually controlled and the pragma will remain in force during compilation of the entire C program, even across modules.

Those warnings that have been disabled can be preserved and recalled using the `warning push` and `warning pop` pragmas. The `warning push` and `warning pop` pragmas can be nested.

The following example normally produces the warning associated with assignment of a `const` object address to a pointer to non-const objects (359).

```
int readp(int * ip) {
    return *ip;
}
const int i = 'd';
int main(void) {
    unsigned char c;
#pragma warning disable 359
    readp(&i);
#pragma warning enable 359
}
```

This same effect would be observed using the following code.

```
#pragma warning push
#pragma warning disable 359
    readp(&i);
#pragma warning pop
```

Here, the state of the messaging system is saved by the `warning push` pragma. Warning 359 is disabled, then after the source code which triggers the warning, the state of the messaging system is retrieved by using the `warning pop` pragma.

5.13.3.11.2 The Warning Error/Warning Pragma

It is possible to change the types of some messages.

A message type can only be changed by using the `warning` pragma and this only affects messages generated by the parser or code generator. The position of the pragma is only significant for the parser; i.e., a parser message number can have its type changed for one section of the code to target specific instances of the message. Specific instances of a message produced by the code generator cannot be individually controlled and the pragma will remain in force during compilation of the entire program.

The following example shows the warning produced in the previous example being converted to an error for the instance in the function `main()`.

```
int main(void) {
    unsigned char c;
    #pragma warning error 359
    readp(&i);
}
```

Building this code would result in an error, not the usual warning.

5.14 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. These options specify the memory of the device and how the sections should be placed in the memory. No linker scripts are used.

The linker will run with options that are obtained from the command-line driver and use linker scripts, which specify memory areas and where sections are to be placed.

The linker options passed to the linker can be adjusted by the user, but this is only required in special circumstances (see 4.6.11.8 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.

Related Links

7.3 Map Files

5.14.1 Compiler-Generated Psects

The code generator places code and data into psects, or sections, with standard names, which are subsequently positioned by the default linker options. The linker does not treat these compiler-generated psects any differently to a psect that has been defined by yourself. A psect can be created in assembly code by using the `PSECT` assembler directive (see 6.1.9.36 Psect Directive).

The names of psects are case sensitive. Some psects use special naming conventions, for example, the `bss` psect, which holds uninitialized objects. There may be uninitialized objects that will need to be located in bank 0 data memory; others may need to be located in bank 1 memory. As these two groups of objects will need to be placed into different memory banks, they will need to be in separate psects so they can be independently controlled by the linker.

The general form of data psect names is:

```
[bit]psectBaseNameCLASS[div]
```

where *`psectBaseName`* is the base name of the psect (listed in 5.14.1.2 Data Space Psects). The *`CLASS`* is a name derived from the linker class (see 5.14.2.2 Data Memory Classes) in which the psect will be linked, e.g., BANK0. The prefix bit is used if the psect holds `__bit` variables. So there can be psects like: bssBANK0, bssBANK1 and bitbssBANK0 defined by the compiler to hold the uninitialized variables.

Note that `__eeprom`-qualified variables can define psects called `bssEEDATA` or `dataEEDATA`, for example, in the same way. Any psect using the class suffix `EEDATA` is placed in the HEX file and is burnt into the EEPROM space when you program the device.

If a `data` psect needs to be split around a reserved area, it will use the letters `l` and `h` (as *`div`* in the above form) in the psect name to indicate if it is the lower or higher division. Thus you might see `bssBANK0l` and `bssBANK0h` psects if a split took place.

If you are unsure which psect holds an object or code in your project, check the assembly list file (see 6.3.1 General Format).

5.14.1.1 Program Space Psects

The following psects reside in program memory.

`checksum`	A psect that is used to mark the position of a checksum that has been requested using the `-mchecksum` option. See 4.6.1.2 Checksum Option for more information. The checksum value is added after the linker has executed so you will not see the contents of this psect in the assembly list file, nor specific information in the map file. Do not change the default linker options relating to this psect.
`cinit`	The psect used by the C initialization runtime startup code. Code in this psect is output by the code generator along with the generated code for the C program. This psect can be linked anywhere within a program memory page, provided it does not interfere with the requirements of other psects.
`config`	The psect used to store the Configuration Words. This psect must be stored in a special location in the HEX file. Do not change the default linker options relating to this psect.
`const`	The PIC18-only psect used to hold objects that are declared `const` and string literals which are not modifiable. Used when the total amount of `const` data in a program exceeds 64k. This psect can be linked anywhere within a program memory page, provided it does not interfere with the requirements of other psects.
`eeprom` (PIC18: `eeprom_data`)	The psect used to store initial values in EEPROM memory. Do not change the default linker options relating to this psect.
`end_init`	The psect used by the C initialization runtime startup code for Baseline and Mid-range devices. This psect holds code which transfers control to the `main()` function.

idata	The psect containing the ROM image of any initialized variables. The class name associated with these psects represents the class of the corresponding RAM-based data psects, to which the content of these psects will be copied. These psects can be linked at any address within a program memory page, provided that they do not interfere with the requirements of other psects.
idloc	This psect is used to store the ID location words. This psect must be stored in a special location in the HEX file. Do not change the default linker options relating to this psect.
init	The psect used by assembly code in the runtime startup assembly module. The code in this and `cinit` define the runtime startup code. If no interrupt code is defined, the Reset vector code can “fall through” into this psect. It is recommended that the linker options for this psect are not changed.
intcode and **intcodelo**	The psect that contains the executable code for the high-priority (default) and low-priority interrupt service routines, respectively, linked to interrupt vector at address 0x8 and 0x18. Do not change the default linker options relating to these psects. See 4.6.1.3 Codeoffset Option if moving code when using a bootloader.
intentry	The psect that contains the entry code for the interrupt service routine which is linked to the interrupt vector. This code saves the necessary registers and jumps to the main interrupt code in the case of Mid-range devices; for Enhanced Mid-range devices this psect will contain the interrupt function body. PIC18 devices use the `intcode` psects. This psect must be linked at the interrupt vector. Do not change the default linker options relating to this psect. See the `-mcodeoffset` option 4.6.1.3 Codeoffset Option if you want to move code when using a bootloader.
ivt0xn	The psect that contains the vector table located at address *n* for devices that use interrupt vector tables or that are operating in legacy mode, see 5.8.1 Writing an Interrupt Service Routine.
jmp_tab	The Baseline only psect used to store jump addresses and function return values. Do not change the default linker options relating to this psect.
maintext	The psect used to hold the assembly code for the `main()` function. The code for `main()` is segregated as it contains the program entry point. Do not change the default linker options relating to this psect as the runtime startup code can “fall through” into this psect.
mediumconst	The PIC18-only psect used to hold objects that are declared `const` and string literals. Used when the total amount of `const` data in a program exceeds 255 bytes, but is less than 64k. This psect can be linked anywhere in the lower 64k of program memory, provided it does not interfere with the requirements of other psects. For PIC18 devices, the location of the psect must be above the highest RAM address.
powerup	The psect that contains executable code for a user-supplied powerup routine. Do not change the default linker options relating to this psect.
reset_vec	The psect used to hold code associated with the Reset vector. Do not change the default linker options relating to this psect. See the `-mcodeoffset` option 4.6.1.3 Codeoffset Option, if you want to move code when using a bootloader.
reset_wrap	The Baseline psect that contains code that is executed after the oscillator calibration location at the top of program memory has been loaded. Do not change the default linker options relating to this psect.
smallconst	The psect that holds objects that are declared `const` and string literals. Used when the total amount of `const` data in a program is less than 255 bytes. This psect can be linked anywhere in the program memory, provided it does not cross a 0x100 boundary and it does not interfere with the requirements of other psects. For PIC18 devices, the location of the psect must be above the highest RAM address.

strings The psect used for `const` objects.
It also includes all unnamed string literals.
This psect can be linked anywhere in the program memory, provided it does not cross a 0x100 boundary or interfere with the requirements of other psects.

stringtext*n* The `stringtext`*n* psects (where *n* is a decimal number) are used for `const` objects when compiling for Enhanced Mid-range devices. These psects can be linked anywhere in the program memory, provided they do not interfere with the requirements of other psects.

textn These psects (where *n* is a decimal number) contain all other executable code that does not require a special link location. These psects can be linked anywhere within any program memory page, and provided they do not interfere with the requirements of other psects. Note that the compiler imposes pseudo page boundaries on some PIC18 devices to work around published errata. Check the default `CODE` linker class for the presence of pages, and their size, in the executable memory.

temp The psect that contains compiler-defined temporary variables. This psect must be linked in common memory, but can be placed at any address in that memory, provided it does not interfere with other psects.

***xxx*_text** The psects for functions that have been made absolute; i.e., placed at an address. *xxx* will be the assembly symbol associated with the function. For example if the function `rv()` is made absolute, code associated with it will appear in the psect called `_rv_text`.
As these psects are already placed at the address indicated in the C source code, the linker options that position them should not be changed.

***xxx*_const** The psects used for `const` objects that has been made absolute; i.e., placed at an address. *xxx* will be the assembly symbol associated with the object. For example, if the array `nba` is made absolute, values stored in this array will appear in the psect called `_nba_const`.
As these psects are already placed at the address indicated in the C source code, the linker options that position them should not be changed.

5.14.1.2 Data Space Psects

The following psects reside in data memory.

nv The psects used to store variables qualified `__persistent`. They are not cleared or otherwise modified at startup. These psects can be linked anywhere in their targeted memory bank and should not overlap any common (unbanked) memory that the device supports.

bss The psects that contain any uninitialized variables.
These psects can be linked anywhere in their targeted memory bank and should not overlap any common (unbanked) memory that the device supports.

data The psects contain the RAM image of any initialized variables. These psects can be linked anywhere in their targeted memory bank and should not overlap any common (unbanked) memory that the device supports.

cstack The psects that contain the compiled stack. On the stack are auto and parameter variables for the entire program. These psects can be linked anywhere in their targeted memory bank and should not overlap any common (unbanked) memory that the device supports.

stack This psect is used as a placeholder for the software stack. This stack is dynamic and its size is not known by the compiler. As described in 5.2.4.2 Data Stacks this psect is typically allocated the remainder of the free data space so that the stack may grow as large as possible. This psect may be linked anywhere in the data memory, but adjusting the default linker options for this psect may limit the size of the software stack. Any overflow of the software stack may cause code failure.

5.14.2 Default Linker Classes

The linker uses classes to represent memory ranges in which psects can be linked.

Classes are defined by linker options (see 7.1.1 A: Define Linker Class). The compiler driver passes a default set of such options to the linker, based on the selected target device. The names of linker classes are case sensitive.

Psects are typically allocated free memory from the class they are associated with. The association is made using the `class` flag with the `PSECT` directive (see 6.1.9.36.3 Class Flag). Alternatively, a psect can be explicitly placed into the memory associated with a class using a linker option (see 7.1.17 P: Position Psect).

Classes can represent a single memory range, or multiple ranges. Even if two ranges are contiguous, the address where one range ends and the other begins, forms a boundary, and psects placed in the class can never cross such boundaries. You can create classes that cover the same addresses, but which are divided into different ranges and have different boundaries. This allows you to accommodate psects whose contents makes assumptions about where it or the data it accesses would be located in memory. Memory allocated from one class will also be reserved from other classes that specify the same memory addresses.

To the linker, there is no significance to a class name or the memory it defines.

Memory can be removed from these classes if using the `-mreserve` option (see 4.6.1.18 Reserve Option), or when subtracting memory ranges using the `-mram` and `-mrom` options (see 4.6.1.17 Ram Option and 4.6.1.20 Rom Option).

Other than reserve memory from classes, never change or remove address boundaries specified by a class.

5.14.2.1 Program Memory Classes

The following linker classes defined by the compiler represent program space memory. Not all classes will be present for each device.

CODE Consists of ranges that map to the program memory pages on the target device and are used for psects containing executable code.
On Baseline devices, it can only be used by code that is accessed via a jump table.

ENTRY Is relevant for Baseline device psects containing executable code that is accessed via a `call` instruction. Calls can only be to the first half of a page on these devices.
The class is defined in such a way that it spans a full page, but the psects it holds will be positioned so that they start in the first half of the page.
This class is also used in Mid-range devices and will consist of many 0x100 word-long ranges, aligned on a 0x100 boundary.

STRING Consists of ranges that are 0x100 words long and aligned on a 0x100 boundary. Thus, it is useful for psects whose contents cannot span a 0x100 word boundary.

STRCODE Defines a single memory range that covers the entire program memory. It is useful for psects whose content can appear in any page and can cross page boundaries.

CONST Consists of ranges that are 0x100 words long and aligned on a 0x100 boundary. Thus, it is useful for psects whose contents cannot span a 0x100 word boundary.

5.14.2.2 Data Memory Classes

The following linker classes defined by the compiler represent data space memory. Not all classes will be present for each device.

RAM Consists of ranges that cover all the general purpose RAM memory of the target device, but excluding any common (unbanked) memory.
Thus, it is useful for psects that must be placed within any general-purpose RAM bank.

BIGRAM Consists of a single memory range that is designed to cover the linear data memory of Enhanced Mid-range devices, or the entire available memory space of PIC18 devices.
It is suitable for any psect whose contents are accessed using linear addressing or which does not need to be contained in a single data bank.

ABS1 Consists of ranges that cover all the general purpose RAM memory of the target device, including any common (unbanked) memory.
Thus, it is useful for psects that must be placed in general purpose RAM, but can be placed in any bank or the common memory,

BANK*x* (where *x* is a bank number) — each consist of a single range that covers the general purpose RAM in that bank, but excluding any common (unbanked) memory.

COMMON Consists of a single memory range that covers the common (unbanked) RAM, if present, for all Mid-range devices.

COMRAM Consists of a single memory range that covers the common (unbanked) RAM, if present, for all PIC18 devices.

SFR*x* (where *x* is a number) — each consists of a single range that covers the SFR memory in bank x. These classes would not typically be used by programmers as they do not represent general purpose RAM.

5.14.2.3 Miscellaneous Classes

The following linker classes defined by the compiler represent memory for special purposes. Not all classes will be present for each device.

CONFIG Consists of a single range that covers the memory reserved for configuration bit data. This class would not typically be used by programmers as it does not represent general purpose RAM.

IDLOC Consists of a single range that covers the memory reserved for ID location data in the hex file. This class would not typically be used by programmers as it does not represent general purpose RAM.

EEDATA Consists of a single range that covers the EEPROM memory of the target device, if present. This class is used for psects that contain data that is to be programmed into the EEPROM.

5.14.3 Changing and Linking the Allocated Section

The `__section()` specifier allows you to have a object or function redirected into a user-define psect, or section.

5.14.1 Compiler-Generated Psects lists the default sections the compiler uses to hold objects and code. The default section used by a function or object can be changed if the object has unique linking requirements that cannot be addressed by existing compiler features.

New sections created by the specifier for objects will have no linker class associated with them. In addition, the compiler will not make assumptions about the final location of the new section. You can link objects specified with `__section()` into any data bank. However, since the compiler cannot know where the new section will be placed until the linker has executed, the code to access the relocated object will be less efficient than the code used to access the object without the specifier.

Since the new section is linked after other objects have been allocated memory, you might also receive memory errors when using the `__section()` specifier. If this is the case, you will need to reserve memory for the new section (see 4.6.1.18 Reserve Option).

New sections created by the specifier for functions will inherit the same flags. However, there are fewer linking restrictions relating to functions and this has minimal impact on the generated code.

The name of the new section you specify must be a valid assembly-domain identifier. The name must contain only alphabetic, numeric characters, or the underscore character, _. It cannot have a name which is the same as that of an assembler directive, control, or directive flag. If the new section contains executable code and you wish this code to be optimized by the assembler, ensure that the section name contains the substring `text`, e.g., `usb_text`. Sections named otherwise will not be modified by the assembler optimizer.

Objects that use the `__section()` specifier will be cleared or initialized in the usual way by the runtime startup code (see 4.3.2 Startup and Initialization). For the case of initialized objects, the compiler will automatically allocate an additional new section (whose name will be the same as the section specified, prefixed with the letter `i`), which will contain the initial values. This section must be stored in program memory, and you might need to locate this section explicitly with a linker option.

The following are examples of a object and function allocated to a non-default section.

```
int __section("myData") foobar;
int __section("myCode") helper(int mode) { /* ... */ }
```

Typically you locate new sections you create with an explicit linker option. So, for example, if you wanted to place the sections created in the above example, you could use the following driver options:

```
-Wl,-PmyData=0200h
-Wl,-AMYCODE=50h-3ffh
-Wl,-PmyCode=MYCODE
```

which will place the section `myData` at address 0x200, and the section `myCode` anywhere in the range 0x50 to 0x3ff represented by the linker class, `MYCODE`. See 7.1 Operation for linker options that can be passed using the `-Wl` driver option (4.6.11.8 Wl: Pass Option To The Linker, Option).

If you are creating a new class for a memory range in program memory and your target is a Baseline or Mid-range PIC device, then you will need to inform the linker that this class defines memory that is word addressable. Do this by using the linker's `-D` option, which indicates a delta value for a class. For example:

```
-Wl,-DMYCODE=2
```

Do not use this option for PIC18 devices, which have byte-addressable program memory.

If you would like to set special flags with the new section, you can do this by providing a definition of the new section in your source code. For example, if you wanted the `myCode` section to be placed at an address that is a multiple of 100h, then you can place the following in your source file:

```
asm("PSECT mycode,reloc=100h");
int __section("myCode") helper(int mode) { /* ... */ }
```

The `reloc`, `size` and `limit` psect flags (see for example 6.1.9.36.16 Reloc Flag) can all be redefined in this way. Redefinitions might trigger assembler warning messages; however, these can be ignored in this circumstance.

5.14.4 Replacing Library Modules

You can easily replace a library routine with your own without having to using the librarian, `xc8-ar` (see 8.1 Archiver/Librarian).

If a source file in your project contains the definition for a routine or object with the same name as a library routine or object, the definition from the source will replace the library definition. This is because the linker scans all the source modules for definitions before it search library files.

You cannot replace a C library function with an equivalent written in assembly code using the above method. If this is required, you will need to use the librarian to edit or create a new library file.

5.14.5 Signature Checking

A signature is a 16-bit value computed from a combination of the function's return type, the number of its parameters and other information affecting how the function is called. This signature is automatically generated and placed the output whenever a C function is referenced or defined. The linker will report any mismatch of signatures, which will indicate a discrepancy between how the function is defined and how it is called.

If it is necessary to write an assembly language routine, it is desirable to include a signature with that routine that is compatible with its equivalent C prototype. The simplest method of determining the correct signature for a function is to write a dummy C function with the same prototype and check the assembly list file using the `-Wa,-a` option.

For example, suppose you have an assembly language routine called `_widget` whose equivalent C prototype takes a `char` argument and returns a `char`. To find the correct signature for such a function, write a test function, such as the following, and check the assembly list file after building this function as part of a test program.

```
char myTestFunction(char arg1) {
}
```

The resultant assembler code seen in the assembly list file includes the following line, indicating that the signature value is 4217:

```
SIGNAT  _myTestFunction,4217
```

Include a similar `SIGNAT` directive in your assembly source code that contains `_widget`.

If you mistakenly declare this assembly routine in a C source file as:

```
extern char widget(long);
```

then the signature generated by this declaration will differ to that specified in your assembly routine and trigger an error.

5.14.6 Linker-Defined Symbols

The linker defines special symbols that can be used to determine where sections that were explicitly linked via an option where located in memory. These symbols can be used in C or assembly code, if required.

The link address of a section can be obtained from the value of a global symbol with name `__L`*`name`* (two leading underscores) where *`name`* is the name of the section. For example, `__LbssBANK0` is the low bound of the `bssBANK0` section. The highest address of a section (i.e., the link address plus the size) is represented by the symbol `__H`*`name`*. If the section has different load and link addresses, the load start address is represented by the symbol `__B`*`name`*; however these are rarely used with PIC devices.

If a section is implicitly linked via a linker class, that is, it is not placed in memory without the explicit use of a `-P` linker option (see 7.1.17 P: Position Psect), the special symbols associated with it are not assigned an address and will have the value 0. Addresses are only assigned to the symbols when the section is linked using a `-P` linker option. That option can simply place the section anywhere in a linker class, for example `-PmyConstData=CONST`.

Assembly code can use these symbol by globally declaring them (noting the two leading underscore characters in the names), for example:

```
GLOBAL __Lidata
```

and C code could use them by declaring a symbol such as the following.

```
extern char * _Lidata;
```

Note that there is only one leading underscore in the C domain. As the C identifier represents an address, a pointer is the typical type choice.

6. Macro Assembler

An assembler is included with the MPLAB XC8 C Compiler to assemble source files for all 8-bit PIC devices. The assembler is automatically run by the compiler driver, `xc8-cc`, for any assembly source files in your project.

This chapter describes the directives (assembler pseudo-ops and controls) accepted by the assembler in the assembly source files or assembly inline with C code.

Although the term “assembler” is almost universally used to describe the tool that converts human-readable mnemonics into machine code, both “assembler” and “assembly” are used to describe the source code which such a tool reads. The latter is more common and is used in this manual to describe the language. Thus you will see the terms assembly language (or just assembly), assembly listing, assembler options, assembler directive, assembler optimizer, and other assembly terms.

The following topics are examined in this chapter of the user’s guide:

- MPLAB XC8 Assembly Language
- Assembly-Level Optimizations
- Assembly List Files

6.1 MPLAB XC8 Assembly Language

Information about the source language accepted by the macro assemblers is described in this section.

All opcode mnemonics and operand syntax are specific to the target device, and you should consult your device data sheet. Additional mnemonics, deviations from the instruction set, and assembler directives are documented in this section.

The same assembler application is used for compiler-generated intermediate assembly and hand-written assembly source code, and for hand-written assembly modules and assembly inline with C code.

6.1.1 Assembly Instruction Deviations

The MPLAB XC8 assembler can use a slightly modified form of assembly language to that specified by the Microchip data sheets. This form is generally easier to read, but the form specified on the data sheet can also be used. The following information details allowable deviations to the instruction format as well as pseudo instructions that can be used in addition to the device instruction set.

These deviations can be used in assembly code in-line with C code or in hand-written assembly modules.

6.1.1.1 Destination And Access Operands

To specify the destination for byte-orientated file register instructions, you may use the operands from either style shown in Table 6-1. The wreg destination indicates that the instruction result will be written to the W register and the file register destination indicates that the result will be written to the register specified by the instruction's file register operand. This operand is usually represented by `,d` in the device data sheet.

Table 6-1. Destination Operand Styles

Style	Wreg destination	File register destination
XC8	`,w`	`,f`
MPASM	`,0`	`,1`

For example (ignoring bank selection and address masking for this example):

```
addwf   foo,w        ;add wreg to foo, leaving the result in wreg
addwf   foo,f        ;add wreg to foo, updating the content of foo
addwf   foo,0        ;add wreg to foo, leaving the result in wreg
addwf   foo,1        ;add wreg to foo, updating the content of foo
```

It is highly recommended that the destination operand is always specified with those instructions where it is needed. If the destination operand is omitted, the destination is assumed to be the file register.

To specify the RAM access bit for PIC18 devices, you may use operands from either style shown in Table 6-2. Banked access indicates that the file register address specified in the instruction is just an offset into the currently selected bank. Unbanked access indicates that the file register address is an offset into the Access bank, or common memory.

Table 6-2. RAM Access Operand Styles

Style	Banked access	Unbanked access
XC8	`,b`	`,c` or `,a`
MPASM	`,1`	`,0`

This operand is usually represented by `,a` in the device data sheet.

Alternatively, an instruction operand can be preceded by the characters “`c:`” to indicate that the address resides in the Access bank. For example:

```
addwf   bar,f,c         ;add wreg to bar in common memory
addwf   bar,f,a         ;add wreg to bar in common memory
addwf   bar,1,0         ;add wreg to bar in common memory
addwf   bar,f,b         ;add wreg to bar in banked memory
addwf   bar,1,1         ;add wreg to bar in banked memory
btfsc   c:bar,3            ;test bit three in the common memory symbol bar
```

It is recommended that you always specify the RAM access operand or the common memory prefix. If these are not present, the instruction address is absolute and the address is within the upper half of the access bank (which dictates that the address must not masked), the instruction will use the access bank RAM. In all other situations, the instruction will access banked memory.

If you use the XC8 style, the destination operand and the RAM access operand can be listed in any order for PIC18 instructions. For example, the following two instructions are identical:

```
addwf   foo,f,c
addwf   foo,c,f
```

Always be consistent in the use of operand style for each instruction, and preferably, that style should remain consistent through the program. For example, the instruction `addwf bar,1,c` is illegal.

For example, the following instructions show the W register being moved to first, an absolute location; and then to an address represented by an identifier. Bank selection and masking has been used in this example. The PIC18 opcodes for these instructions, assuming that the address assigned to `foo` is 0x516 and to `bar` is 0x55, are shown below.

```
6EE5  movwf 0FE5h              ;write to access bank location 0xFE5
6E55  movwf bar,c              ;write to access bank location 0x55
0105  BANKSEL(foo)             ;set up BSR to access foo
6F16  movwf BANKMASK(foo),b    ;write to foo (banked)
6F16  movwf BANKMASK(foo)      ;defaults to banked access
```

Notice that the first two instruction opcodes have the RAM access bit (bit 8 of the op-code) cleared, but that the bit is set in the last two instructions.

6.1.1.2 Bank And Page Selection

The `BANKSEL()` pseudo instruction can be used to generate instructions to select the bank of the operand specified. The operand should be the symbol or address of an object that resides in the data memory.

Depending on the target device, the generated code will either contain one or more bit instructions to set/clear bits in the appropriate register, or use a `movlb` instruction (in the case of enhanced mid-range or PIC18 devices). As this pseudo instruction can expand to more than one instruction on mid-range or baseline parts, it should not immediately follow a `btfsx` instruction on those devices.

For example:

```
movlw 20
BANKSEL(_foobar)     ;select bank for next file instruction
movwf BANKMASK(_foobar)  ;write data and mask address
```

In the same way, the `PAGESEL()` pseudo instruction can be used to generate code to select the page of the address operand. For the current page, you can use the location counter, `$`, as the operand.

Depending on the target device, the generated code will either contain one or more instructions to set/clear bits in the appropriate register, or use a `movlp` instruction in the case of enhanced mid-range PIC devices. As the directive could expand to more than one instruction, it should not immediately follow a `btfsx` instruction.

For example:

```
fcall _getInput
PAGESEL $          ;select this page
```

This directive is accepted when compiling for PIC18 targets but has no effect and does not generate any code. Support is purely to allow easy migration across the 8-bit devices.

6.1.1.3 Address Masking

A file register address used with most instructions should be masked to remove the bank information from the address. Failure to do this might result in a linker fixup error.

All MPLAB XC8 assembly identifiers represent a full address. This address includes the bank information for the object it represents. Virtually all instructions in the 8-bit PIC instruction sets that take a file register operand expect this operand value to be an offset into the currently selected bank. As the device families have different bank sizes, the width of this offset is different for each family.

The `BANKMASK()` macro can be used with identifier or address operands. The macro ANDs out the bank information in the address using a suitable mask. It is available once you include `<xc.inc>`. Use of this macro increases assembly code portability across Microchip devices, since it adjusts the mask to suit the bank size of the target device. An example of this macro is given in 6.1.1.2 Bank And Page Selection.

Do not use this macro with any instruction that expects its operand to be a full address, such as the PIC18's `movff` instruction.

6.1.1.4 Movfw Pseudo Instruction

The `movfw` pseudo instruction implemented by MPASM is not implemented in the MPLAB XC8 assemblers. You will need to use the standard PIC instruction that performs an identical function. Note that the MPASM instruction:

```
movfw foobar
```

maps directly to the standard PIC instruction:

```
movf foobar,w
```

6.1.1.5 Movff/movffl Instructions

The `movff` instruction is a physical device instruction, but for PIC18 devices that have extended data memory, it also serves as a placeholder for the `movffl` instruction.

For these devices, when generating output for the `movff` instruction, the assembler checks the psects that hold the operand symbols. If the psect containing the source operand and the psect containing the destination operand both specify the `lowdata` psect flag, the instruction is encoded as the two-word `movff` instruction. If an operand is an absolute address and that address is in the lower 4kB of memory, then that is also considered acceptable for the shorter form of the instruction. In all other situations, the instruction is encoded as a three-word `movffl` instruction.

Note that assembly list files will always show the `movff` mnemonic, regardless of how it is encoded. Check the number of op-code words to determine which instruction was encoded.

6.1.1.6 Interrupt Return Mode

The `retfie` PIC18 instruction can be followed by “`f`” (no comma) to indicate that the shadow registers should be retrieved and copied to their corresponding registers on execution. Without this modifier, the registers are not updated from the shadow registers. This syntax is not relevant for Baseline and Mid-range devices.

The following examples show both forms and the opcodes they generate.

```
0011  retfie f    ;shadow registers copied
0010  retfie      ;return without copy
```

The “`0`” and “`1`” operands indicated in the device data sheet can be alternatively used if desired.

6.1.1.7 Long Jumps And Calls

The assembler recognizes several mnemonics that expand into regular PIC MCU assembly instructions. The mnemonics are `fcall` and `ljmp`.

On baseline and mid-range parts, these instructions expand into regular `call` and `goto` instructions respectively, but also ensure the instructions necessary to set the bits in PCLATH (for mid-range devices) or STATUS (for baseline devices) will be generated, should the destination be in another page of program memory. Page selection instructions can appear immediately before the `call` or `goto`, or be generated as part of, and immediately after, a previous `fcall`/`ljmp` mnemonic.

On PIC18 devices, these mnemonics are present purely for compatibility with smaller 8-bit devices and are always expanded as regular PIC18 `call` and `goto` instructions.

These special mnemonics should be used where possible, as they make assembly code independent of the final position of the routines that are to be executed. Whether the page selection instructions are generated, and exactly where they will be located, is dependent on the surrounding source code. If the `call` or `goto` is determined to be within the current page, the additional code to set the PCLATH bits can be optimized away.

The operand to the `fcall` and `ljmp` mnemonics should not be masked, regardless of target device. The full address is required to determine the destination address and page (where applicable). When the mnemonic is expanded, the address used with either the `call` or `goto` instruction in the expansion will be automatically masked. When using a `call` or `goto` instruction directly in your source code, always apply the appropriate mask to the operand.

The following mid-range PIC example shows an `fcall` instruction in the assembly list file. You can see that the `fcall` instruction has expanded to five instructions. In this example, there are two bit instructions that set/clear bits in the PCLATH register. Bits are also set/cleared in this register after the call to reselect the page that was selected before the `fcall`.

```
13  0079  3021                      movlw   33
14  007A  120A  158A  2000          fcall   _phantom
          120A  118A
15  007F  3400                      retlw   0
```

Since `fcall` and `ljmp` instructions can expand into more than one instruction, they should never be preceded by an instruction that can skip, e.g., a `btfsc` instruction.

The `fcall` and `ljmp` instructions assume that the psect that contains them is smaller than a page. Do not use these instructions to transfer control to a label in the current psect if it is larger than a page. The default linker options will not permit code psects to be larger than a page.

On PIC18 devices, the regular `call` instruction can be followed by a “`,f`” to indicate that the W, STATUS and BSR registers should be pushed to their respective shadow registers. This replaces the “`,1`” syntax indicated on the device data sheet.

6.1.1.8 Relative Branches

The PIC18 devices implement conditional relative branch instructions, e.g., `bz`, `bnz`. These instructions have a limited jump range compared to the `goto` instruction.

Note that in some instances, the assembler can change a relative branch instruction to be a relative branch with the opposite condition over a `goto` instruction. For example:

```
  bz error
  ;next
```

can become:

```
  bnz l18
  goto error
l18:
  ;next
```

This transformation is made so that a conditional branch can be made to span the same range as a `goto` instruction.

This substitution will not take place for hand-written assembly placed in-line with C code or in separate modules, or for assembly source that is protected with the `ASMOPT` directive (see 6.1.9.2 Asmopt Directive).

6.1.2 Statement Formats

Valid statement formats are shown in Table 6-3.

The *`label`* field is optional and, if present, should contain one identifier. A label can appear on a line of its own, or precede a mnemonic as shown in the second format.

The third format is only legal with certain assembler directives, such as `MACRO`, `SET` and `EQU`. The *`name`* field is mandatory and should contain one identifier.

If the assembly source file is first processed by the preprocessor, then it can also contain lines that form valid preprocessor directives. See 5.13.1 Preprocessor Directives for more information on the format for these directives.

There is no limitation on what column or part of the line in which any part of the statement should appear.

Table 6-3. Assembler Statement Formats

Format #	Field1	Field2	Field3	Field4
Format 1	*`label`*`:`			
Format 2	*`label`*`:`	*`mnemonic`*	*`operands`*	`;` *`comment`*
Format 3	`name`	*`pseudo-op`*	*`operands`*	`;` *`comment`*
Format 4	`;` *`comment only`*			
Format 5	empty line			

6.1.3 Characters

The character set used is standard 7 bit ASCII. Alphabetic case is significant for identifiers, but not mnemonics and reserved words. Tabs are equivalent to spaces.

6.1.3.1 Delimiters

All numbers and identifiers must be delimited by white space, non-alphanumeric characters or the end of a line.

6.1.3.2 Special Characters

There are a few characters that are special in certain contexts. Within a macro body, the character `&` is used for token concatenation. To use the bitwise `&` operator within a macro body, escape it by using `&&` instead or use the `and` form of this operator. In a macro argument list, the angle brackets `<` and `>` are used to quote macro arguments.

6.1.4 Comments

An assembly comment is initiated with a semicolon that is not part of a string or character constant, for example:

```
  movlw 22   ;this value will ensure there is a good safety margin
```

If the assembly file is first processed by the C preprocessor, then the file can also contain C or C++ style comments using the standard `/* ... */` and `//` syntax.

Avoid using assembly comments (`;comment`) in preprocessor directives, especially the `#define` directive. Assembly comments are not removed by the C preprocessor prior to macro substitution and so will appear in the substituted text, possibly resulting in build errors. Always use C or C++ style comments in these situations.

6.1.4.1 Special Comment Strings

Several comment strings are appended to compiler-generated assembly instructions by the code generator. These comments are typically used by the assembler optimizer.

The comment string `;volatile` is used to indicate that the memory location being accessed in the instruction is associated with a variable that was declared as `volatile` in the C source code. Accesses to this location which appear to be redundant will not be removed by the assembler optimizer if this string is present.

This comment string can also be used in hand-written assembly source to achieve the same effect for locations defined and accessed in assembly code.

The comment string `;wreg free` is placed on some `call` instructions. The string indicates that the W register was not loaded with a function parameter; i.e., it is not in use. If this string is present, optimizations can be made to assembler instructions before the function call, which loads the W register redundantly.

6.1.5 Constants

6.1.5.1 Numeric Constants

The assembler performs all arithmetic with signed 32-bit precision.

The default radix for all numbers is 10. Other radices can be specified by a trailing base specifier, as given in the Table 6-4 table.

Table 6-4. Numbers And Bases

Radix	Format
Binary	Digits 0 and 1 followed by `B`
Octal	Digits 0 to 7 followed by `O`, `Q`, `o` or `q`
Decimal	Digits 0 to 9 followed by `D`, `d` or nothing
Hexadecimal	Digits 0 to 9, A to F preceded by `0x` or followed by `H` or `h`

Hexadecimal numbers must have a leading digit (e.g., `0ffffh`) to differentiate them from identifiers. Hexadecimal digits are accepted in either upper or lower case.

Note that a binary constant must have an upper case `B` following it, as a lower case `b` is used for temporary (numeric) label backward references.

In expressions, real numbers are accepted in the usual format, and are interpreted as IEEE 32-bit format.

6.1.5.2 Character Constants And Strings

A character constant is a single character enclosed in single quotes `'`.

Multi-character constants, or strings, are a sequence of characters, not including carriage return or newline characters, enclosed within matching quotes. Either single quotes `'` or double quotes `"` can be used, but the opening and closing quotes must be the same.

6.1.6 Identifiers

Assembly identifiers are user-defined symbols representing memory locations or numbers. A symbol can contain any number of characters drawn from alphabetics, numerics, as well as special characters: dollar, `$`; question mark, `?`; and underscore, `_`.

The first character of an identifier cannot be numeric nor the `$` character. The case of alphabetics is significant, e.g., `Fred` is not the same symbol as `fred`. Some examples of identifiers are shown here:

```
An_identifier
an_identifier
```

```
an_identifier1
?$_12345
```

An identifier cannot have the same symbol (any case) as any of the assembly code mnemonics (e.g. `movlw` or `return`) assembler directives (e.g. `SET` or `LIST`), directive argument tokens (e.g. `hex` or `push`), or operators (e.g. `mod` or `nul`).

An identifier that begins with at least one underscore character can be accessed from C code. Care must be taken with such symbols that they do not interact with C code identifiers. Identifiers that do not begin with an underscore can only be accessed from the assembly domain. See the Equivalent Assembly Symbols section for the mapping between the C and assembly domains.

6.1.6.1 Significance Of Identifiers

Users of other assemblers that attempt to implement forms of data typing for identifiers should note that this assembler attaches no significance to any symbol, and places no restrictions or expectations on the usage of a symbol.

The names of psects (program sections) and ordinary symbols occupy separate, overlapping name spaces, but other than this, the assembler does not care whether a symbol is used to represent bytes, words or sports cars. No special syntax is needed or provided to define the addresses of bits or any other data type, nor will the assembler issue any warnings if a symbol is used in more than one context. The instruction and addressing mode syntax provide all the information necessary for the assembler to generate correct code.

6.1.6.2 Assembler-generated Identifiers

Where a `LOCAL` directive is used in a macro block, the assembler will generate a unique symbol to replace each specified identifier in each expansion of that macro. These unique symbols will have the form `??`*`nnnn`* where *`nnnn`* is a 4-digit number. The user should avoid defining symbols with the same form.

6.1.6.3 Location Counter

The current location within the active program section is accessible via the symbol `$`. This symbol expands to the address of the currently executing instruction (which is different than the address contained in the program counter (PC) register when executing this instruction). Thus:

```
goto $   ;endless loop
```

will represent code that will jump to itself and form an endless loop. By using this symbol and an offset, a relative jump destination can be specified.

Any address offset added to `$` has the native addressability of the target device. So, for baseline and mid-range devices, the offset is the number of instructions away from the current location, as these devices have word-addressable program memory. For PIC18 instructions, which use byte addressable program memory, the offset to this symbol represents the number of bytes from the current location. As PIC18 instructions must be word aligned, the offset to the location counter should be a multiple of 2. All offsets are rounded down to the nearest multiple of 2.

For example:

```
goto       $+2   ;skip...
movlw      8     ;to here for PIC18 devices, or
movwf      _foo  ;to here for baseline and mid-range devices
```

will skip the `movlw` instruction on baseline or mid-range devices. On PIC18 devices, `goto $+2` will jump to the following instruction; i.e., act like a `nop` instruction.

6.1.6.4 Register Symbols

Code in assembly modules can gain access to the special function registers by including pre-defined assembly header files. The appropriate file can be included by add the line:

```
#include <xc.inc>
```

to the assembler source file and using a `.S` extension with the source filename to ensure it is preprocessed.This header file contains appropriate commands to ensure that the header file specific for the target device is included into the source file.

These header files contain `EQU` declarations for all byte or multi-byte sized registers and `#define` macros for named bits within byte registers.

6.1.6.5 Symbolic Labels

A label is a symbolic alias that is assigned a value that is equal to the current address within the current psect. Labels are not assigned a value until link time.

A label definition consists of any valid assembly identifier that must be followed by a colon, `:`. The definition can appear on a line by itself or it can be positioned to the left of an instruction or assembler directive. Here are two examples of legitimate labels interspersed with assembly code.

```
frank:
  movlw   1
  goto        fin
simon44: clrf _input
```

Here, the label `frank` will ultimately be assigned the address of the `movlw` instruction and `simon44` the address of the `clrf` instruction. Regardless of how they are defined, the assembler list file produced by the assembler will always show labels on a line by themselves.

Labels can be used (and are preferred) in assembly code, rather than using an absolute address with other instructions. In this way, they can be used as the target location for jump-type instructions or to load an address into a register.

Like variables, labels have scope. By default, they can be used anywhere in the module in which they are defined. They can be used by code located before their definition. To make a label accessible in other modules, use the `GLOBAL` directive (see 6.1.9.21 Global Directive for more information).

6.1.7 Expressions and Operators

Expressions can consist of unary operators (one operand, e.g., `not`) or binary operators (two operands, e.g., `+`). The operators allowable in expressions are listed in the table below.

Operators within expressions are evaluated left to right, thus the expression `5 + 1 * 2` will yield the value 12. Use parenthesis to change the order of operator execution, for example `5 + (1 * 2)` will yield the value 7.

With the exception of the equality (e.g. `=`) and relational operators (e.g. `>=`), all the operators listed can be freely combined in both constant and relocatable expressions. Relocatable expressions will be evaluated by the linker at link time, after psects have been positioned and symbol values have been determined.

Table 6-5. Assembly Operators

Operator	Purpose	Example
`*`	multiplication	`movlw 4*33,w`
`+`	addition	`bra $+1`
`-`	subtraction	`DB 5-2`
`/`	division	`movlw 100/4`
= or `eq`	equality	`IF inp eq 66`
> or `gt`	signed greater than	`IF inp > 40`
>= or `ge`	signed greater than or equal to	`IF inp ge 66`
< or `lt`	signed less than	`IF inp < 40`
<= or `le`	signed less than or equal to	`IF inp le 66`
<> or `ne`	signed not equal to	`IF inp <> 40`
`low`	low byte of operand	`movlw low(inp)`
`high`	high byte of operand	`movlw high(1008h)`

...........continued		
Operator	**Purpose**	**Example**
`highword`	high 16 bits of operand	`DW highword(inp)`
`mod`	modulus	`movlw 77mod4`
`&` or `and`	bitwise AND	`clrf inp&0ffh`
`^`	bitwise XOR (exclusive or)	`movf inp^80,w`
`|`	bitwise OR	`movf inp|1,w`
`not`	bitwise complement	`movlw not 055h,w`
`<<` or `shl`	shift left	`DB inp>>8`
`>>` or `shr`	shift right	`movlw inp shr 2,w`
`rol`	rotate left	`DB inp rol 1`
`ror`	rotate right	`DB inp ror 1`
`float24`	24-bit version of real operand	`DW float24(3.3)`
`nul`	tests if macro argument is null	

6.1.8 Program Sections

Program sections, or psects, are simply a section of code or data. They are a way of grouping together parts of a program (via the psect's name) even though the source code cannot be physically adjacent in the source file, or even where spread over several modules.

A psect is identified by a name and has several attributes. The `PSECT` assembler directive is used to define a psect. It requires a name argument, which may be followed an comma-separated list of flags which define its attributes. Linker options that can be used to control psect placement in memory are described in 7. Linker. These options can be accessed from the driver using the `-Wl` driver option (see 4.6.11.8 Wl: Pass Option To The Linker, Option), negating the need for you to run the linker explicitly.

See 5.14.1 Compiler-Generated Psects for a list of all psects that the code generator defines.

Unless defined as `abs` (absolute), psects are relocatable.

Code or data that is not explicitly placed into a psect using the `PSECT` directive will become part of the default (unnamed) psect. As you have no control over where this psect is linked, it recommended that a `PSECT` directive always be placed before the code and objects.

When writing assembly code, you can use the existing compiler-generated psects or create your own. You will not need to adjust the linker options if you are using compiler-generated psects.

If you create your own psects, try to associate them with an existing linker class (see 5.14.2 Default Linker Classes and 7.1.2 C: Associate Linker Class To Psect) otherwise you can need to specify linker options for them to be allocated correctly.

Note, that the length and placement of psects is important. It is easier to write code if all executable code is located in psects that do not cross any device pages boundaries; so, too, if data psects do not cross bank boundaries. The location of psects (where they are linked) must match the assembly code that accesses the psect contents.

6.1.9 Assembler Directives

Assembler directives, or pseudo-ops, are used in a similar way to instruction mnemonics. With the exception of `PAGESEL` and `BANKSEL`, these directives do not generate instructions. The `DB`, `DW` and `DDW` directives place data bytes into the current psect. The directives are listed in the following sections.

Table 6-6. Assembler Directives

Directive	Purpose
ALIGN	Aligns output to the specified boundary.
ASMOPT	Controls whether subsequent code is optimized by the assembler.
BANKISEL	Generates code to select bank of operand for indirect access on some devices.
BANKSEL	Generates code to select bank of operand.
CALLSTACK	Indicates the call stack depth remaining.
[NO]COND	Controls inclusion of conditional code in the listing file.
CONFIG	Specifies configuration bits.
DABS	Defines absolute storage.
DB	Defines constant byte(s).
DDW	Defines double-width constant word(s).
DEBUG_SOURCE	Controls debug information.
DLABS	Define linear-memory absolute storage.
DS	Reserves storage.
DW	Defines constant word(s).
ELSE	Alternates conditional assembly.
ELSIF	Alternates conditional assembly.
ENDIF	Ends conditional assembly.
END	Ends assembly.
ENDM	Ends macro definition.
EQU	Defines symbol value.
ERROR	Generates a user-defined error.
[NO]EXPAND	Controls expansion of assembler macros in the listing file.
EXTRN	Links with global symbols defined in other modules.
FILE	Indicates the source file that contains the assembly code following.
GLOBAL	Makes symbols accessible to other modules or allow reference to other global symbols defined in other modules.
IF	Conditional assembly.
INCLUDE	Textually includes the content of the specified file.
IRP	Repeats a block of code with a list.
IRPC	Repeats a block of code with a character list.
LINE	Indicates the line number of the current source file that contains the assembly code following.
[NO]LIST	Defines options for listing file.
LOCAL	Defines local tabs.
MACRO	Macro definition.
MESSG	Generates a user-defined advisory message.

...........continued	
Directive	**Purpose**
`ORG`	Sets location counter within current psect.
`PAGELEN`	Specifies the length of the listing file page.
`PAGESEL`	Generates set/clear instruction to set PCLATH bits for this page.
`PAGEWIDTH`	Specifies the width of the listing file page.
`PROCESSOR`	Defines the particular chip for which this file is to be assembled.
`PSECT`	Declares or resumes program section.
`RADIX`	Specifies radix for numerical constants.
`REPT`	Repeats a block of code n times.
`SET`	Defines or re-defines symbol value.
`SIGNAT`	Defines function signature.
`SUBTITLE`	Specifies the subtitle of the program for the listing file.
`TITLE`	Specifies the title of the program for the listing file.
`WARN`	Generates a user-defined warning.

6.1.9.1 Align Directive

The `ALIGN` directive aligns whatever is following, data storage or code etc., to the specified offset boundary within the current psect. The boundary is specified as a number of bytes following the directive.

For example, to align output to a 2-byte (even) address within a psect, the following could be used.

```
ALIGN 2
```

Note that what follows will only begin on an even absolute address if the psect begins on an even address; i.e., alignment is done within the current psect. See 6.1.9.36.16 Reloc Flag for psect alignment.

The `ALIGN` directive can also be used to ensure that a psect's length is a multiple of a certain number. For example, if the above `ALIGN` directive was placed at the end of a psect, the psect would have a length that was always an even number of bytes long.

6.1.9.2 Asmopt Directive

The `ASMOPT` *`action`* directive selectively controls the assembler optimizer when processing assembly code. The allowable actions are shown in Table 6-7.

Table 6-7. Asmopt Actions

Action	Purpose
`off`	Disables the assembler optimizer for subsequent code.
`on`	Enables the assembler optimizer for subsequent code.
`pop`	Retrieves the state of the assembler optimization setting.
`push`	Saves the state of the assembler optimization setting.

No code is modified after an `ASMOPT off` directive. Following an `ASMOPT on` directive, the assembler will perform allowable optimizations.

The `ASMOPT push` and `ASMOPT pop` directives allow the state of the assembler optimizer to be saved onto a stack of states and then restored at a later time. They are useful when you need to ensure the optimizers are disabled for a small section of code, but you do not know if the optimizers have previously been disabled.

For example:

```
ASMOPT PUSH  ;store the state of the assembler optimizers
ASMOPT OFF   ;optimizations must be off for this sequence
movlw 0x55
movwf EECON2
movlw 0xAA
movwf EECON2
ASMOPT POP   ;restore state of the optimizers
```

6.1.9.3 Bankisel Directive

The `BANKISEL` directive is used to generate code that ensures the correct data bank will be selected for indirect access of the operand's memory location. The operand should be the symbol or numeric address of an object that resides in data memory, for example

```
PROCESSOR 12F510

#include <xc.inc>

PSECT udata_bank1
myVar:
  DS  1

PSECT code
setMode:
  movlw myVar
  movwf FSR        ;Load the address of myVar info FSR
  BANKISEL myVar   ;Select the correct bank for myVar
  movlw 055h
  movwf INDF       ;Indirectly write to myVar
```

This directive will set the IRP bit (Mid-range) or STATUS bits (Baseline) appropriately for the symbol argument. For all other devices, this directive will be ignored.

6.1.9.4 Banksel Directive

The `BANKSEL` directive can be used to generate code to select the data bank of the operand. The operand should be the symbol or address of an object that resides in the data memory (see 6.1.1.2 Bank And Page Selection).

6.1.9.5 Callstack Directive

The `CALLSTACK` *`depth`* directive indicates to the assembler the number of call stack levels still available at that particular point in the program.

This directive is used by the assembler optimizers to determine if transformations like procedural abstraction can take place.

6.1.9.6 Cond Directive

The `COND` directive includes conditional code in the assembly listing file. The complementary `NOCOND` directive will not include conditional code in the listing file.

6.1.9.7 Config Directive

The `CONFIG` directive allows the configuration bits (or fuses) and id-location registers to be specified in the assembly source file. Configuration bits, or fuses, are used to set up fundamental device operation, such as the oscillator mode, watchdog timer, programming mode and code protection. These bits must be correctly set to ensure your program executes correctly.

The directive has the following forms:

```
CONFIG setting = value
CONFIG register = literal_value
```

Here, *`setting`* is a configuration setting descriptor (e.g., `WDT`) and *`value`* can be either a textual description of the desired state (e.g., `OFF`) or a numerical value. Either the *`setting`* or *`value`* tokens or the *`setting =`*

value expression can be surrounded by either double or single quotes to protect them from any macro substitution performed by the preprocessor, for example:

```
CONFIG "WDT = ON"          ;turn on watchdog timer
CONFIG "FEXTOSC" = "ECH"  ;external clock oscillator mode, high power PFM
CONFIG WDTPS = 0x1A       ;specify the timer postscale value
```

Some `value` tokens that appear to be purely numerical are in fact a textual description of the value, for example in the following:

```
config WDTPS = 32
```

the `32` token is a textual description for some devices that indicates a watchdog timer post-scale of 1:32, not the number 32. In such a case, it might not be the number 32 that is programmed into the relevant bits of the configuration register. The assembler will first check if *value* represents a predefined string and, only if that is not the case, assume it represents a numerical constant, which will then be subject to the same constraints as other numerical constant operands.

You should never assume that the `OFF` and `ON` tokens used in configuration macros equate to 0 and 1, respectively, as that is often not the case.

The *register* field is the name of a configuration or id-location register, and this must always be used with a *value* that is a numerical constant, for example:

```
CONFIG CONFIG1L = 0x8F
```

The available *setting*, *register* and *value* fields are documented in the chipinfo file relevant to your device (i.e. `pic_chipinfo.html` and `pic18_chipinfo.html`) and that are 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 `CONFIG` directive can be used to set each configuration setting; alternatively, several comma-separated configuration settings can be specified by the same directive. The directive can be used as many times as required to fully configure the device.

The following example shows a configuration register being programmed as a whole and programmed using the individual settings contained within that register.

```
; PIC18F67K22
; VREG Sleep Enable bit : Enabled
; LF-INTOSC Low-power Enable bit : LF-INTOSC in High-power mode during Sleep
; SOSC Power Selection and mode Configuration bits : High Power SOSC circuit selected
; Extended Instruction Set : Enabled
CONFIG RETEN = ON, INTOSCSEL = HIGH, SOSCSEL = HIGH, XINST = ON

; Alternatively
CONFIG CONFIG1L = 0x5D

; IDLOC @ 0x200000
CONFIG IDLOC0 = 0x15
```

All the bits in the Configuration Words should be programmed to prevent erratic program behavior. Do not leave them in their default/unprogrammed state. Not all Configuration bits have a default state of logic high; some have a logic low default state. Consult your device data sheet for more information.

If you are using MPLAB X IDE, take advantage of its built-in tools to generate the required pragmas, so that you can copy and paste them into your source code. See the *MPLAB® X IDE User's Guide* for a description and use of the Configuration Bits window.

6.1.9.8 Dabs Directive

The `DABS` directive allows one or more bytes of memory to be reserved at the specified address. The general form of the directive is:

```
DABS memorySpace, address, bytes [,symbol]
```

where *memorySpace* is a number representing the memory space in which the reservation will take place, *address* is the address at which the reservation will take place and *bytes* is the number of bytes that is to be reserved. The symbol is optional and refers to the name of the object at the address.

Specifying a symbol allows you to access the reserved memory in your code. This symbol is automatically made globally accessible and is equated to the address specified in the directive. For example, the symbol, `foo`, defined by the following directive:

```
DABS 1,0x100,4,foo
```

can be used in code, for example:

```
    movlw 20
    movwf BANKMASK(foo)
```

This directive differs to the `DS` directive in that it can be used to reserve memory at any location, not just within the current psect. Indeed, these directives can be placed anywhere in the assembly code and do not contribute to the currently selected psect in any way. Additionally, objects defined by the `DS` directive inside a psect are allocated free memory by the linker, whereas the allocation address must be specified and manged by the programmer when using the `DABS` directive.

The memory space number is the same as the number specified with the `space` flag option to psects (see 6.1.9.36.18 Space Flag).

The linker reads this `DABS`-related information from object files and ensures that the reserved addresses are not used for other memory placement.

6.1.9.9 Db Directive

The `DB` directive is used to initialize bytes of program memory.

The directive takes a comma-separated list of arguments. Each argument can be a numeric value (e.g., `55`), a character constant (e.g. `'T'`), or a string (e.g. `"level"` or `'level'`). Each argument or each character of a string argument is written as one byte into consecutive program memory locations within the current psect.

The encoding of the values written depend on the `delta` flag of the psect that contains the directive. In the following code:

```
PSECT myBytes,class=CODE,delta=2
alabel:
  DB  ’X’,1,2,3,4,
```

the size of the address unit in the program memory specified by the `delta` psect flag (see 6.1.9.36.4 Delta Flag) is 2 bytes (true for Baseline and Mid-range PIC devices). In this case, the `DB` directive will initialize each word of program memory with the upper byte set to zero, specifically (in hexadecimal):

```
0058 0001 0002 0003 0004
```

However, on PIC18 devices, which uses program memory psects with the `delta` flag set to 1, no padding will occur. For the code:

```
PSECT myBytes,class=CODE,delta=1
alabel:
  DB  'X',1,2,3,4
```

the following (hexadecimal) data will be defined in the HEX file for the program memory.

```
58 01 02 03 04
```

Strings consist of a sequence of characters enclosed within either single or double quotes. There is no termination character added by the assembler. If you need a nul-terminated string, then specify the termination character as a separate argument to the `DB` directive, for example:

```
PSECT myConst,class=CODE,delta=1
bytes:
        DB "a terminated string",0
```

will define:

```
61 20 74 65 72 6D 69 6E 61 74 65 64 20 73 74 72 69 6E 67 00
```

The `DB` directive cannot be used to created objects in data memory. For that, use the `DS` directive (see 6.1.9.13 Ds Directive).

6.1.9.10 DDW Directive

The `DDW` directive is used to initialize 32-bit words of program memory.

The directive takes a comma-separated list of arguments. Each argument can be a numeric value (e.g., `55`), a character constant (e.g. `'T'`), or a string (e.g. `"level"` or `'level'`). Each argument or each character of a string argument is written as four bytes into consecutive program memory locations within the current psect.

In the following code:

```
PSECT myWords,class=CODE,delta=2
alabel:
  DDW  'X',1,2,398,0x8005FFFF
```

the `DDW` directive will initialize each word of program memory with the supplied value, specifically (in hexadecimal):

```
00000058 00000001 00000002 0000018E 8005FFFF
```

Strings consist of a sequence of characters enclosed within either single or double quotes. There is no termination character added by the assembler. If you need a nul-terminated string, then specify the termination character as a separate argument to the `DDW` directive, for example:

```
PSECT myConst,class=CODE,delta=1
words:
        DDW "a terminated string",0
```

will define:

```
00000061 00000020 00000074 00000065 00000072 0000006D 00000069 0000006E 00000061 00000074
00000065 00000064 00000020 00000073 00000074 00000072 00000069 0000006E 00000067 00000000
```

The `DDW` directive cannot be used to created objects in data memory. For that, use the `DS` directive (see 6.1.9.13 Ds Directive).

6.1.9.11 Debug Source Directive

The `DEBUG_SOURCE` *`action`* directive controls whether the object-code generated by the assembler should favors debugging assembly or C sources. The allowable actions are shown in Table 6-8.

Table 6-8. Debug Source Actions

Action	Purpose
`C`	Favor debugging C source.
`asm`	Favor debugging assembly source.
`pop`	Retrieves the state of the assembler debug source setting.
`push`	Saves the state of the assembler debug source setting.

This directive controls the generation of information that might affect debugging, for example the debug information associated with labels, which can restrict the full expansion of assembly macros shown in the MPLAB X IDE Disassembly View. The directive only affects how the code is debugged; it does not affect the operation of assembled code.

If this directive is not specified, the `asm` setting is employed. Use this setting for projects built with the PIC Assembler or around assembly code using macros that is part of a C project. The `c` setting will automatically be used in assembly output from the MPLAB XC8 C compiler for assembly generated from, C code.

The `DEBUG_SOURCE push` and `DEBUG_SOURCE pop` directives allow the state of the debug source setting to be saved onto a stack of states and then restored at a later time. They are useful when you need to set a particular debug source state for a small section of code, but you do not know what state the debug source setting had previously been.

For example:

```
DEBUG_SOURCE push    ;store the state of the debug source setting
DEBUG_SOURCE asm     ;ensure proper debugging of the macro used below
    MY_UNLOCK_MACRO  ;this should expand in the Disassembly View
DEBUG_SOURCE pop     ;restore state of the debug source setting
```

6.1.9.12 Dlabs Directive

The `DLABS` directive must be used to reserve one or more bytes of memory at the specified linear address on those devices that support linear addressing. Typically, the directive is used to allocate memory for large objects that do not fit within one data bank and which must be accessed via linear addressing. The general form of the directive is:

```
DLABS memorySpace, address, bytes [,symbol]
```

The *`memorySpace`* argument is a number representing the linear memory space. This will be the same number as the banked data space. The *`address`* is the address at which the reservation will take place. This can be specified as either a linear or banked address. The *`bytes`* is the number of bytes that is to be reserved. The symbol is optional and refers to the name of the object at the address.

Specifying a symbol allows you to access the reserved memory in your code. The symbol is automatically made globally accessible and is equated to the address specified in the directive in linear addressing form. For example, the symbol, `foo`, defined by the following directive:

```
DLABS 1,0x120,128,foo
```

will be assigned the linear address 0x20A0 and can be used in code, for example:

```
    movlw low(foo)
    movwf FSR1L
    movlw high(foo)
    movwf FSR1H
```

The memory space number is the same as the number specified with the `space` flag option to psects (see 6.1.9.36.18 Space Flag).

The linker reads this `DLABS`-related information from object files and ensures that the reserved addresses are not used for other memory placement.

6.1.9.13 Ds Directive

The `DS` *`units`* directive reserves, but does not initialize, the specified amount of space. The single argument is the number of address units to be reserved. An address unit is determined by the flags used with the psect that holds the directive.

This directive is typically used to reserve bytes for RAM-based objects in the data memory (the enclosing psect's `space` flag set to 1). If the psect in which the directive resides is a bit psect (the psect's `bit` flag was set), the directive reserves the request number of bits. If used in a psect linked into the program memory, it will move the location counter, but not place anything in the HEX file output. Note that on Mid-range and Baseline devices, the size of an address unit in the program memory is 2 bytes (see 6.1.9.36.4 Delta Flag), so the `DS` pseudo-op will actually reserve words in that instance.

An object is typically defined by using a label and then the `DS` directive to reserve locations at the label location.

Examples:

```
PSECT myVars,space=1,class=BANK2
alabel:
  DS 23    ;reserve 23 bytes of memory
PSECT myBits,space=1,bit,class=COMRAM
xlabel:
  DS 2+3   ;reserve 5 bits of memory
```

6.1.9.14 Dw Directive

The DW directive is used to initialize 16-bit words of program memory.

The directive takes a comma-separated list of arguments. Each argument can be a numeric value (e.g., 55), a character constant (e.g. 'T'), or a string (e.g. "level" or 'level'). Each argument or each character of a string argument is written as two bytes into consecutive program memory locations within the current psect.

In the following code:

```
PSECT myWords,class=CODE,delta=2
alabel:
  DW  'X',1,2,398,5472
```

the DW directive will initialize each word of program memory with the supplied value, specifically (in hexadecimal):

```
0058 0001 0002 018E 1560
```

Strings consist of a sequence of characters enclosed within either single or double quotes. There is no termination character added by the assembler. If you need a nul-terminated string, then specify the termination character as a separate argument to the DW directive, for example:

```
PSECT myConst,class=CODE,delta=1
words:
        DW "a terminated string",0
```

will define:

```
0061 0020 0074 0065 0072 006D 0069 006E 0061 0074 0065 0064 0020 0073 0074 0072 0069 006E
0067 0000
```

The DW directive cannot be used to created objects in data memory. For that, use the DS directive (see 6.1.9.13 Ds Directive).

6.1.9.15 End Directive

The END *label* directive is optional, but if present should be at the very end of the program. It will terminate the assembly and not even blank lines should follow this directive.

If an expression is supplied as an argument, that expression will be used to define the entry point of the program. This is stored in a start record in the object file produced by the assembler. Whether this is of any use will depend on the linker.

For example:

```
END  start_label  ;defines the entry point
```

or

```
END               ;do not define entry point
```

6.1.9.16 Equ Directive

The EQU pseudo-op defines a symbol and equates its value to an expression. For example

```
thomas EQU 123h
```

The identifier thomas will be given the value 123h. EQU is legal only when the symbol has not previously been defined. See 6.1.9.39 Set Directive for redefinition of values.

This directive does *not* reserve memory for the symbol specified. Use the DS directive to reserve data memory (see 6.1.9.13 Ds Directive). An EQU performs a similar function to the preprocessor's #define directive (see 5.13.1 Preprocessor Directives).

6.1.9.17 Error Directive

The `error` directive produces a user-defined build-time error message that will halt the assembler. The message to be printed should this directive be executed is specified as a string. Typically this directive will be made conditional, to detect an invalid situation.

For example:

```
IF MODE
  call process
ELSE
  ERROR "no mode defined"
ENDIF
```

6.1.9.18 Expand Directive

The `EXPAND` directive shows code generated by macro expansions in the assembler listing file. The complementary `NOEXPAND` directive hides code generated by macro expansions in the assembler listing file.

6.1.9.19 Extrn Directive

The `EXTRN` *`identifier`* pseudo-op is similar to `GLOBAL` (see 6.1.9.21 Global Directive), but can only be used to link in with global symbols defined in other modules. An error will be triggered if you use `EXTRN` with a symbol that is defined in the same module.

6.1.9.20 File Directive

The `FILE` directive indicates the source file that contains the assembly code following. For example

```
FILE "X1_start.s"
```

This directive is used to build debug information associated with a project and it is typically found in intermediate assembly files, placed there by the C code generator. They shouldn't be required for hand-written assembly code.

6.1.9.21 Global Directive

The `GLOBAL` *`identifier_list`* directive declares a list of comma-separated symbols. If the symbols are defined within the current module, they are made public. If the symbols are not defined in the current module, they are made references to public symbols defined in external modules. Thus to use the same symbol in two modules the `GLOBAL` directive must be used at least twice: once in the module that defines the symbol to make that symbol public and again in the module that uses the symbol to link in with the external definition.

For example:

```
GLOBAL  lab1,lab2,lab3
```

6.1.9.22 If, Elsif, Else And Endif Directives

These directives implement conditional assembly.

The argument to `IF` and `ELSIF` should be an absolute expression. If it is non-zero, then the code following it up to the next matching `ELSE`, `ELSIF` or `ENDIF` will be assembled. If the expression is zero, then the code up to the next matching `ELSE` or `ENDIF` will not be output. At an `ELSE`, the sense of the conditional compilation will be inverted, while an `ENDIF` will terminate the conditional assembly block. Conditional assembly blocks can be nested.

These directives do not implement a runtime conditional statement in the same way that the C statement `if` does; they are only evaluated when the code is built. In addition, assembly code in both true and false cases is always scanned and interpreted, but the machine code corresponding to instructions is output only if the condition matches. This implies that assembler directives (e.g., `EQU`) will be processed regardless of the state of the condition expression and should not be used inside an `IF` construct.

For example:

```
IF ABC
    goto aardvark
ELSIF DEF
    goto denver
ELSE
    goto grapes
```

```
ENDIF
ENDIF
```

In this example, if ABC is non-zero, the first goto instruction will be assembled but not the second or third. If ABC is zero and DEF is non-zero, the second goto instruction will be assembled but the first and third will not. If both ABC and DEF are zero, the third goto instruction will be assembled.

6.1.9.23 Include Directive

The INCLUDE "*filename*" directive causes the specified file to be textually replace this directive. For example:

```
INCLUDE "options.inc"
```

The compiler driver does not pass any search paths to the assembler, so if the include file is not located in the current working directory, the file's full path must be specified with the file name.

Assembly source files with a .S extension are preprocessed, thus allowing use of preprocessor directives, such as #include, which is an alternative to the INCLUDE directive.

6.1.9.24 Irp And Irpc Directives

The IRP and IRPC directives operate in a similar way to REPT; however, instead of repeating the block a fixed number of times, it is repeated once for each member of an argument list.

In the case of IRP, the list is a conventional macro argument list. In the case or IRPC, it is each character in one argument. For each repetition, the argument is substituted for one formal parameter.

For example:

```
IRP number,4865h,6C6Ch,6F00h
    DW number
ENDM
```

would expand to:

```
DW 4865h
DW 6C6Ch
DW 6F00h
```

Note that you can use local labels and angle brackets in the same manner as with conventional macros.

The IRPC directive is similar, except it substitutes one character at a time from a string of non-space characters.

For example:

```
IRPC char,ABC
    DB ’char’
ENDM
```

will expand to:

```
DB ’A’
DB ’B’
DB ’C’
```

6.1.9.25 Line Directive

The LINE directive indicates the line number of the current source file that contains the assembly code following. For example

```
LINE 26
```

This directive is used to build debug information associated with a project and it is typically found in intermediate assembly files, placed there by the C code generator. They shouldn't be required for hand-written assembly code.

6.1.9.26 List Directive

The LIST directive controls whether listing output is produced.

If the listing was previously turned off using the `NOLIST` directive, the `LIST` directive will turn it back on.

Alternatively, the `LIST` control can include options to control the assembly and the listing. The options are listed in the Table 6-9 table.

Table 6-9. List Directive Options

List Option	Default	Description
c=*nnn*	80	Set the page (i.e., column) width.
n=*nnn*	59	Set the page length.
t=ON|OFF	OFF	Truncate listing output lines. The default wraps lines.
p=*device*	n/a	Set the device type.
x=ON|OFF	OFF	Turn macro expansion on or off.

6.1.9.27 Local Directive

The `LOCAL` *`label`* directive allows unique labels to be defined for each expansion of a given macro. Any symbols listed after the `LOCAL` directive will have a unique assembler generated symbol substituted for them when the macro is expanded. For example:

```
down MACRO count
  LOCAL more
  more: decfsz count
  goto more
ENDM
```

when expanded, will include a unique assembler generated label in place of more. For example:

```
down foobar
```

expands to:

```
??0001 decfsz foobar
goto ??0001
```

If invoked a second time, the label more would expand to `??0002` and multiply defined symbol errors will be averted.

6.1.9.28 Macro And Endm Directives

The `MACRO` … `ENDM` directives provide for the definition of assembly macros, optionally with arguments. See 6.1.9.16 Equ Directive for simple association of a value with an identifier, or 5.13.1 Preprocessor Directives for the preprocessor's `#define` macro directive, which can also work with arguments.

The `MACRO` directive should be preceded by the macro name and optionally followed by a comma-separated list of formal arguments. When the macro is used, the macro name should be used in the same manner as a machine opcode, followed by a list of arguments to be substituted for the formal parameters.

For example:

```
;macro: movlf - Move a literal value into a nominated file register
;args:  arg1 - the literal value to load
;       arg2 - the NAME of the source variable
movlf   MACRO   arg1,arg2
  movlw arg1
  movwf arg2 mod 080h
ENDM
```

When used, this macro will expand to the 2 instructions in the body of the macro, with the formal parameters substituted by the arguments. Thus:

```
movlf 2,tempvar
```

expands to:

```
movlw 2
movwf tempvar mod 080h
```

The `&` character can be used to permit the concatenation of macro arguments with other text, but is removed in the actual expansion. For example:

```
loadPort MACRO port, value
  movlw value
  movwf PORT&port
ENDM
```

will load `PORTA` if `port` is `A` when called, etc. The special meaning of the `&` token in macros implies that you can not use the bitwise `AND` operator, (also represented by `&`), in assembly macros; use the `and` form of this operator instead.

A comment can be suppressed within the expansion of a macro (thus saving space in the macro storage) by opening the comment with a double semicolon, `;;`.

When invoking a macro, the argument list must be comma-separated. If it is desired to include a comma (or other delimiter such as a space) in an argument then angle brackets `<` and `>` can be used to quote

If an argument is preceded by a percent sign, `%`, that argument will be evaluated as an expression and passed as a decimal number, rather than as a string. This is useful if evaluation of the argument inside the macro body would yield a different result.

The `nul` operator can be used within a macro to test a macro argument, for example:

```
IF nul     arg3  ;argument was not supplied.
...
ELSE             ;argument was supplied
...
ENDIF
```

See 6.1.9.27 Local Directive for use of unique local labels within macros.

By default, the assembly list file will show macro in an unexpanded format; i.e., as the macro was invoked. Expansion of the macro in the listing file can be shown by using the `EXPAND` assembler directive (see 6.1.9.18 Expand Directive).

6.1.9.29 Messg Directive

The `messg` directive produces a user-defined build-time advisory message. Execution of this directive will not prevent the assembler from building. The message to be printed should this directive be executed is specified as a string. Typically this directive will be made conditional, to detect an invalid situation.

For example:

```
IF MODE
  call process
ELSE
  MESSG "no mode defined"
ENDIF
```

6.1.9.30 Org Directive

The `ORG` directive changes the value of the location counter within the current psect. This means that the addresses set with `ORG` are relative to the base address of the psect, which is not determined until link time.

Note: The much-abused `ORG` directive does not move the location counter to the absolute address you specify. Only if the psect in which this directive is placed is absolute and overlaid will the location counter be moved to the specified address. To place objects at a particular address, place them in a psect of their own and link this at the required address using the linkers `-P` option (see 7.1.17 P: Position Psect). The `ORG` directive is not commonly required in programs.

The argument to ORG must be either an absolute value, or a value referencing the current psect. In either case, the current location counter is set to the value determined by the argument. It is not possible to move the location counter backward. For example:

```
ORG 100h
```

will move the location counter to the beginning of the current psect plus 100h. The actual location will not be known until link time.

In order to use the ORG directive to set the location counter to an absolute value, the directive must be used from within an absolute, overlaid psect. For example:

```
PSECT absdata,abs,ovrld
ORG 50h
;this is guaranteed to reside at address 50h
```

6.1.9.31 Page Directive

The PAGE directive causes a new page to be started in the listing output. A Control-L (form feed) character will also cause a new page when it is encountered in the source.

6.1.9.32 Pagelen Directive

The PAGELEN *nnn* directive sets the length of the assembly listing to be the number of specified lines.

6.1.9.33 Pagesel Directive

The PAGESEL directive can be used to generate code to select the page of the address operand. (see 6.1.1.2 Bank And Page Selection).

6.1.9.34 Pagewidth Directive

The PAGEWIDTH *nnn* directive sets the width of the assembly listing to be the number of specified characters.

6.1.9.35 Processor Directive

The PROCESSOR directive should be used in a module if the assembler source is only applicable to one device. The -mcpu option must always be used when building to specify the target device the code is being built for. If there is a mismatch between the device specified in the directive and in the option, an error will be triggered.

For example:

```
PROCESSOR 18F4520
```

6.1.9.36 Psect Directive

The PSECT directive declares or resumes a program section.

The directive takes as argument a name and, optionally, a comma-separated list of flags. The allowed flags specify attributes of the psect. They are listed in the Table 6-10 table.

The psect name is in a separate name space to ordinary assembly symbols, so a psect can use the same identifier as an ordinary assembly identifier. However, a psect name cannot be one of the assembler directives, keywords, or psect flags.

Once a psect has been declared, it can be resumed later by another PSECT directive; however, the flags need not be repeated and will be propagated from the earlier declaration. An error is generated if two PSECT directives for the same psect are encountered with contradictory flags, the exceptions being that the reloc, size and limit flags can be respecified without error.

Table 6-10. Psect Flags

Flag	Meaning
abs	psect is absolute.
bit	psect holds bit objects.
class=*name*	Specify class name for psect.

...........continued	
Flag	**Meaning**
`delta=`*`size`*	Size of an addressing unit.
`global`	psect is global (default).
`inline`	psect contents (function) can be inlined when called.
`keep`	psect will not be deleted after inlining.
`limit=`*`address`*	Upper address limit of psect (PIC18 only).
`local`	psect is unique and will not link with others having the same name.
`lowdata`	psect will be entirely located below the 0x1000 address.
`merge=`*`allow`*	Allow or prevent merging of this psect.
`noexec`	For debugging purposes, this psect contains no executable code.
`note`	psect does not contain any data that should appear in the program image.
`optim=`*`optimizations`*	specify optimizations allowable with this psect.
`ovrld`	psect will overlap same psect in other modules.
`pure`	psect is to be read-only.
`reloc=`*`boundary`*	Start psect on specified boundary.
`size=`*`max`*	Maximum size of psect.
`space=`*`area`*	Represents area in which psect will reside.
`split=`*`allow`*	Allow or prevent splitting of this psect.
`with=`*`psect`*	Place psect in the same page as specified psect.

Some examples of the use of the `PSECT` directive follow:

```
; swap output to the psect called fred
PSECT fred
; swap to the psect bill, which has a maximum size of 100 bytes and which is global
PSECT bill,size=100h,global
; swap to joh, which is an absolute and overlaid psect that is part of the CODE linker class,
; and whose content has a 2-byte word at each address
PSECT joh,abs,ovrld,class=CODE,delta=2
```

6.1.9.36.1 Abs Flag

The `abs` psect flag defines the current psect as being absolute; i.e., it is to start at location 0. This does not mean that this module's contribution to the psect will start at 0, since other modules can contribute to the same psect (See also 6.1.9.36.14 Ovrld Flag).

An `abs`-flagged psect is not relocatable and an error will result if a linker option is issued that attempts to place such a psect at any location.

6.1.9.36.2 Bit Flag

The `bit` psect flag specifies that a psect holds objects that are 1 bit wide. Such psects will have a scale value of 8, indicating that there are 8 addressable units to each byte of storage and that all addresses associated with this psect will be bit addresses, not byte addresses. Non-unity scale values for psects are indicated in the map file (see 7.3 Map Files).

6.1.9.36.3 Class Flag

The `class` psect flag specifies a corresponding linker class name for this psect. A class is a range of addresses in which psects can be placed.

Class names are used to allow local psects to be located at link time, since they cannot always be referred to by their own name in a `-P` linker option (as would be the case if there are more than one local psect with the same name).

Class names are also useful where psects need only be positioned anywhere within a range of addresses rather than at a specific address. The association of a class with a psect that you have defined typically means that you do not need to supply a custom linker option to place it in memory.

See 7.1.1 A: Define Linker Class for information on how linker classes are defined.

6.1.9.36.4 Delta Flag

The `delta` psect flag defines the size of the addressable unit. In other words, the number of data bytes that are associated with each address.

With PIC Mid-range and Baseline devices, the program memory space is word addressable; so, psects in this space must use a delta of 2. That is to say, each address in program memory requires 2 bytes of data in the HEX file to define their contents. So, addresses in the HEX file will not match addresses in the program memory.

The data memory space on these devices is byte addressable; so, psects in this space must use a delta of 1. This is the default delta value.

All memory spaces on PIC18 devices are byte addressable; so a delta of 1 (the default) should be used for all psects on these devices.

The redefinition of a psect with conflicting delta values can lead to phase errors being issued by the assembler.

6.1.9.36.5 Global Flag

The `global` psect flag indicates that the linker should concatenate this psect with global psects in other modules and which have the same name.

Psects are considered global by default, unless the `local` flag is used.

6.1.9.36.6 Inline Flag

This flag is deprecated. Consider, instead, using the `optim` psect flag.

The `inline` psect flag is used by the code generator to tell the assembler that the contents of a psect can be inlined. If this operation is performed, the contents of the `inline` psect will be copied and used to replace calls to the function defined in the psect.

6.1.9.36.7 Keep Flag

This flag is deprecated. Consider, instead, using the `optim` psect flag.

The `keep` psect flag ensures that the psect is not deleted after any inlining by the assembler optimizer. Psects that are candidates for inlining (see 6.1.9.36.6 Inline Flag) can be deleted after the inlining takes place.

6.1.9.36.8 Limit Flag

The `limit` psect flag specifies a limit on the highest address to which a psect can extend. If this limit is exceeded when it is positioned in memory, an error will be generated. This is currently only available when building for PIC18 devices.

6.1.9.36.9 Local Flag

A psect defined using the `local` psect flag will not be combined with other `local` psects from other modules at link time, even if there are others with the same name. Where there are two `local` psects in the one module, they reference the same psect. A `local` psect cannot have the same name as any `global` psect, even one in another module.

Psects which are local and which are not associated with a linker class (see 6.1.9.36.3 Class Flag) cannot be linked to an address using the `-P` linker option, since there could be more than one psect with this name. Typically these psects define a class flag and they are placed anywhere in that class range.

6.1.9.36.10 Merge Flag

This flag is deprecated. Consider, instead, using the `optim` psect flag.

The `merge` psect flag controls how the psect will be merged with others. This flag can be assigned 0, 1, or not specified. When assigned 0, the psect will never be merged by the assembly optimizer during optimizations. If assigned the value 1, the psect can be merged if other psect attributes allow it and the optimizer can see an advantage in doing so. If this flag is not specified, then merging will not take place.

Typically, merging is only performed on code-based psects (`text` psects).

6.1.9.36.11 Noexec Flag

The `noexec` psect flag is used to indicate that the psect contains no executable code. This information is only relevant for debugging purposes.

6.1.9.36.12 Note Flag

The `note` psect flag is used by special psects whose content is intended for compiler or debugger tools, and whose content will not be copied to the final program output. When the `note` flag is specified, several other psect flags are prohibited and their use with the same psect will result in a warning.

6.1.9.36.13 Optim Flag

The `optim` psect flag is used to indicate the optimizations that can be performed on the psect's content, provided such optimizations are permitted and have been enabled.

The optimizations are indicated by a colon-separated list of names, shown in the Table 6-11 table. An empty list implies that no optimizations can be performed on the psect.

Table 6-11. Optim Flag Names

Name	Optimization
`inline`	Allow the psect content to be inlined.
`jump`	Perform jump-based optimizations.
`merge`	Allow the psect's content to be merged with that of other similar psects (PIC10/12/16 devices only).
`pa`	Perform proceedural abstraction.
`peep`	Perform peephole optimizations.
`remove`	Allow the psect to be removed entirely if it is completely inlined.
`split`	Allow the psect to be split into smaller psects if it surpasses size restrictions (PIC10/12/16 devices only).
`empty`	Perform no optimization on this psect.

So, for example, the psect definition:

```
PSECT myText,class=CODE,reloc=2,optim=inline:jump:split
```

allows the assembler optimizer to perform inlining, splitting and jump-type optimizations of the `myText` psect content if those optimizations are enabled. The definition:

```
PSECT myText,class=CODE,reloc=2,optim=
```

disables all optimizations associated with this psect regardless of the optimizer setting.

The `optim` psect flag replaces the use of the separate psect flags: `merge`, `split`, `inline` and `keep`.

6.1.9.36.14 Ovrld Flag

The `ovrld` psect flag tells the linker that the content of this psect should be overlaid with that from other modules at link time. Normally psects with the same name are concatenated across modules. The contributions to an overlaid psect in the same module are always concatenated.

This flag in combination with the `abs` flag (see 6.1.9.36.1 Abs Flag) defines a truly absolute psect; i.e., a psect within which any symbols defined are absolute.

6.1.9.36.15 Pure Flag

The `pure` psect flag instructs the linker that this psect will not be modified at runtime. So, for example, be placed in ROM. This flag is of limited usefulness since it depends on the linker and target system enforcing it.

6.1.9.36.16 Reloc Flag

The `reloc` psect flag allows the specification of a requirement for alignment of the psect on a particular boundary. The boundary specification must be a power of two, for example 2, 8 or 0x40. For example, the flag `reloc=100h` would specify that this psect must start on an address that is a multiple of 0x100 (e.g., 0x100, 0x400, or 0x500).

PIC18 instructions must be word aligned, so a `reloc` value of 2 must be used for any PIC18 psect that contains executable code. All other sections, and all sections for all other devices, can typically use the default `reloc` value of 1.

6.1.9.36.17 Size Flag

The `size` psect flag allows a maximum size to be specified for the psect, e.g., `size=100h`. This will be checked by the linker after psects have been combined from all modules.

6.1.9.36.18 Space Flag

The `space` psect flag is used to differentiate areas of memory that have overlapping addresses, but are distinct. Psects that are positioned in program memory and data memory have a different space value to indicate that the program space address 0, for example, is a different location to the data memory address 0.

The memory spaces associated with the space flag numbers are shown in Table 6-12.

Table 6-12. Space Flag Numbers

Space Flag Number	Memory Space
0	Program memory, and EEPROM for PIC18 devices
1	Data memory
2	Reserved
3	EEPROM on Mid-range devices
4	Configuration bit
5	IDLOC
6	Note

Devices that have a banked data space do not use different space values to identify each bank. A full address that includes the bank number is used for objects in this space. So, each location can be uniquely identified. For example, a device with a bank size of 0x80 bytes will use address 0 to 0x7F to represent objects in bank 0, and then addresses 0x80 to 0xFF to represent objects in bank 1, etc.

6.1.9.36.19 Split Flag

This flag is deprecated. Consider, instead, using the `optim` psect flag.

The `split` psect flag can be assigned 0, 1, or not specified. When assigned 0, the psect will never be split by the assembly optimizer during optimizations. If assigned the value 1, the psect can be split if other psect attributes allow it and the psect is too large to fit in available memory. If this flag is not specified, then the splitability of this psect is based on whether the psect can be merged, see 6.1.9.36.10 Merge Flag.

6.1.9.36.20 With Flag

The `with` psect flag allows a psect to be placed in the same page with another psect. For example the flag `with=text` will specify that this psect should be placed in the same page as the `text` psect.

The term `withtotal` refers to the sum of the size of each psect that is placed “with” other psects.

6.1.9.37 Radix Directive

The `RADIX` *`radix`* directive controls the radix for numerical constants specified in the assembler source files. The allowable radices are shown in Table 6-13.

Table 6-13. Radix Operands

Radix	Meaning
dec	Decimal constants
hex	Hexadecimal constants
oct	Octal constants

6.1.9.38 Rept Directive

The REPT directive temporarily defines an unnamed macro, then expands it a number of times as determined by its argument.

For example:

```
REPT 3
    addwf fred,w
ENDM
```

will expand to:

```
addwf fred,w
addwf fred,w
addwf fred,w
```

(see 6.1.9.24 Irp And Irpc Directives).

6.1.9.39 Set Directive

The SET directive is equivalent to EQU (6.1.9.16 Equ Directive), except that it allows a symbol to be re-defined without error. For example:

```
thomas SET 0h
```

This directive does *not* reserve memory for the symbol specified. Use the DS directive to reserve data memory (see 6.1.9.13 Ds Directive). A SET performs a similar function to the preprocessor's #define directive (see 5.13.1 Preprocessor Directives).

6.1.9.40 Signat Directive

The SIGNAT directive is used to associate a 16-bit signature value with a label. At link time, the linker checks that all signatures defined for a particular label are the same. The linker will produce an error if they are not. The SIGNAT directive is used to enforce link time checking of function prototypes and calling conventions.

For example:

```
SIGNAT _fred,8192
```

associates the signature value 8192 with the symbol _fred. If a different signature value for _fred is present in any object file, the linker will report an error.

Often, this directive is used with assembly language routines that are called from C. The easiest way to determine the signature value used by the MPLAB XC8 C Compiler is to write a C routine with the same prototype as that required for the assembly routine, and check that function's signature directive argument, as determined by the code generator and as shown in the assembly list file.

6.1.9.41 Space Directive

The SPACE *nnn* directive places *nnn* blank lines in the assembly listing output.

6.1.9.42 Subtitle Directive

The SUBTITLE "*string*" directive defines a subtitle to appear at the top of every assembly listing page, but under the title. The subtitle should be enclosed in single or double quotes.

6.1.9.43 Title Directive

The `TITLE "string"` directive keyword defines a title that will appear at the top of every assembly listing page. The title should be enclosed in single or double quotes.

6.1.9.44 Warn Directive

The `WARN` directive produces a user-defined build-time warning message. Execution of this directive will not prevent the compiler from building. The warning to be printed should this directive be executed is specified as a string. Typically this directive will be made conditional, to detect an invalid situation.

For example:

```
IF MODE
  movwf modeQ
ELSE
  WARN "MODE is zero - write skipped"
ENDIF
```

6.2 Assembly-Level Optimizations

The assembler performs optimizations on assembly code, in addition to those optimizations performed by the code generator directly on the C code.

The assembler only optimizes hand-written assembly source modules if the `-fasmfile` driver optimization setting is enabled, see 4.6.6.7 Asmfile Option. Assembly added in-line (see 5.11.2 Inline Assembly) with C code is never optimized.

The optimizations that can be performed by the assembler are listed below. Unless indicated, these optimizations are only enabled in optimization levels 3 and s, thus preventing them from being used by an unlicensed compiler (see 4.6.6 Options for Controlling Optimization).

Assembly-level optimizations include:

- 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 5.7.1.2 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 that use a label, `PLx`, where *x* is a number. 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.
- Jump-to-jump type optimizations are made primarily to tidy the output related to conditional code sequences that follow a generic template. Jump-to-jump optimizations can remove jump instructions whose destinations are also jump instructions. This optimization is enabled at optimization level 2, making it accessible to unlicensed compilers.
- Unreachable code is removed. Code can become orphaned by other optimizations and cannot be reached during normal execution, e.g., instructions after a return instruction. The presence of any label is considered a possible entry point and code following a label is always considered reachable.
- 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. For example, if W is known to contain the value 0, and an instruction moves W to an address (`movwf`), this might be replaceable with a `clrf` instruction.
- Psect merging can be performed to allow other optimizations to take place. Code within the same psect is guaranteed to be located in the same program memory page. Calls and jumps within the psect do not need to have the page selection bits set before executing. Code using the `ljmp` and `fcall` instructions will benefit from this optimization (see 6.1.1 Assembly Instruction Deviations).

Assembly optimizations can often interfere with debugging in some tools, such as MPLAB X IDE. When debugging code, it might be necessary to select a lower optimization level that disables them, if possible (see 4.6.6 Options for

Controlling Optimization). The assembler optimizations can drastically reduce code size. However, they typically have little effect on RAM usage.

6.3 Assembly List Files

The assembler will produce an assembly list file if instructed. The `xc8-cc` driver option `-Wa,-a` is typically used to request generation of such a file (see 4.6.10 Mapped Assembler Options).

The assembly list file shows the assembly output produced by the compiler for both C and assembly source code. If the assembler optimizers are enabled, the assembly output can be different than assembly source code. It is still useful for assembly programming.

The list file is in a human-readable form and cannot be used by the assembly as a source file. It differs from an assembly output file in that it contains address and op-code data.

The assembler optimizer simplifies some expressions and removes some assembler directives from the listing file for clarity, although these directives are included in the true assembly output files. If you are using the assembly list file to look at the code produced by the compiler, you might wish to turn off the assembler optimizer so that all the compiler-generated directives are shown in the list file. Re-enable the optimizer when continuing development. 4.6.6 Options for Controlling Optimization gives more information on controlling the optimizers.

Provided that the link stage has successfully concluded, the listing file is updated by the linker so that it contains absolute addresses and symbol values. Thus, you can use the assembler list file to determine the position and exact op codes of instructions.

Tick marks "`'`" in the assembly listing, next to addresses or opcodes, indicate that the linker did not update the list file, most likely due to a compiler error, or a compiler option that stopped compilation before the link stage. For example, in the following listing:

```
85  000A' 027F                      subwf   127,w
86  000B' 1D03                      skipz
87  000C' 2800'                     goto    u15
```

These marks indicate that addresses are just address offsets into their enclosing psect, and that opcodes have not been fixed up. Any address field in the opcode that has not been fixed up is shown with a value of 0.

There is a single assembly list file produced by the assembler for each assembly file passed to it. So, there is a single file produced for all the C source code in a project, including p-code based library code. The file also contains some of the C initialization that forms part of the runtime startup code. There is also a single file produced for each assembly source file. Typically, there is at least one assembly file in each project. It contains some of the runtime startup file and is typically named `startup.s`.

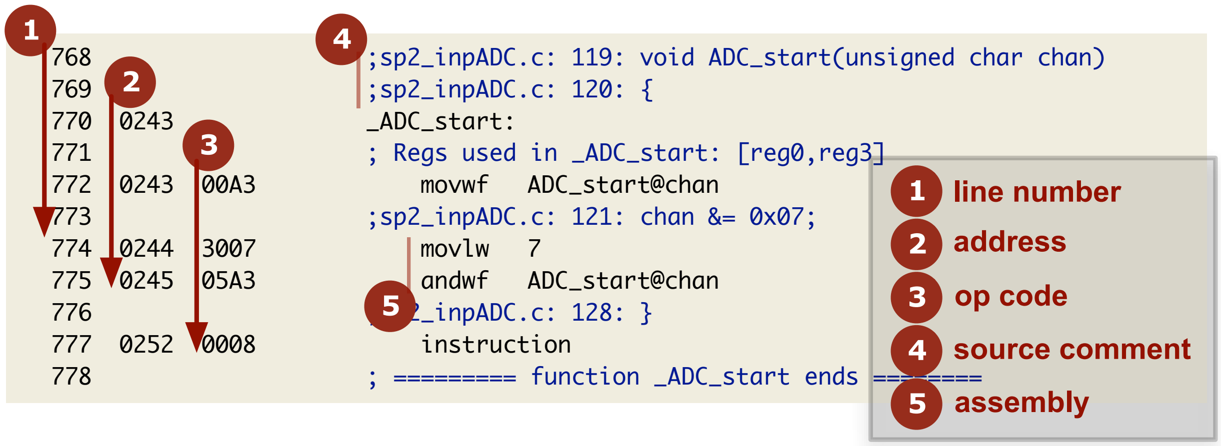
6.3.1 General Format

The format of the main listing is in the form shown in the figure below.

The line numbers purely relate to the assembly list file and are not associated with the lines numbers in the C or assembly source files. Any assembly that begins with a semicolon indicates it is a comment added by the code generator. Such comments contain either the original source code, which corresponds to the generated assembly, or is a comment inserted by the code generator to explain some action taken.

Before the output for each function, there is detailed information regarding that function summarized by the code generator. This information relates to register usage, local variable information, functions called, and the calling function.

Figure 6-1. General Form of Assembly Listing File



6.3.2 Psect Information

The assembly list file can be used to determine the name of the psect in which a data object or section of code has been placed.

For global symbols, you can check the symbol table in the map file which lists the psect name with each symbol. For symbols local to a module, find the definition of the symbol in the list file. For labels, it is the symbol's name followed by a colon, '`:`'. Look for the first `PSECT` assembler directive above this code. The name associated with this directive is the psect in which the code is placed (see 6.1.9.36 Psect Directive).

6.3.3 Function Information

For each C function, printed before the function's assembly label (search for the function's name that is immediately followed by a colon `:`), is general information relating to the resources used by that function. A typical printout is shown in the diagram below. Most of the information is self-explanatory, but special comments follow.

The locations shown use the format *`offset`*`[space]`. For example, an indicated location of `42[BANK0]` means that the variable was located in the bank 0 memory space and that it appears at an offset of 42 bytes into the compiled stack component in this space (see 5.2.4.2.1 Compiled Stack Operation). It does mean the variable was assigned address 42. Any other object with the address `42[BANK0]` is at the same address.

Whenever pointer variables are shown, they are often accompanied by the targets that the pointer can reference, these targets appear after the arrow `->` (see 6.3.5 Pointer Reference Graph). The auto and parameter section of this information is especially useful because the size of pointers is dynamic (see 5.3.6 Pointer Types). This information shows the actual number of bytes assigned to each pointer variable.

The tracked objects information is generally of no concern to programmers. It indicates the known state of the currently selected RAM bank on entry to the function and at its exit points. It also indicates the bank selection bits that did, or did not, change in the function.

The hardware stack information shows how many stack levels were taken up by this function alone, and the total levels used by this function and any functions it calls. Note that this is only valid for functions that are have not been inlined.

Functions that use a non-reentrant model are those that allocate auto and parameter variables to a compiled stack and which are, as a result, not reentrant. If a function is marked as being reentrant, it allocates stack-based variables to the software stack and can be reentrantly called.

Functions marked as using a non-reentrant model are those which allocate auto and parameter variables to a compiled stack and which are, hence, not reentrant. If a function is marked as being reentrant, then it allocates stack-based variables to the software stack and can be reentrantly called.

Figure 6-2. Function Information

```
1 *************** function _render ****************
;; Defined at:
;;                 line 29 in file "draw.c"  2
3 Parameters:     Size  Location     Type
;;                None
;; Auto vars:     Size  Location     Type
;;  lll             4   42[BANK0 ] long
;;  x          4    2   46[BANK0 ] volatile int
;;  cp              1   41[BANK0 ] PTR unsigned char
;;                 -> inputData(2),
5 Return value:   Size  Location     Type
;;                None               void
6 Registers used:
;;                wreg, fsr0l, fsr0h, status,2, status,0, pclath, cstack
7 Tracked objects:
;;                On entry : 17F/0
;;                On exit  : 0/0
;;                Unchanged: FFE00/0
;; Data sizes:      COMMON    BANK0   BANK1   BANK2
;;      Params:         0       0       0       0
8       Locals:         0       7       0       0
;;      Temps:          0       5       0       0
;;      Totals:         0      12       0       0
;;Total ram usage:       12 bytes
;; Hardware stack levels used:    1
9 Hardware stack levels required when called:    4
;; This function calls:
;;                _lrv
;;           10   ___altofl
;;                ___awdiv
;;                ___awmod
11 This function is called by:
;;                _main
12 This function uses a non-reentrant model
```

1. function's name
2. file name and line number of definition
3. size, location and type of parameters
4. size, location and type of auto variables
5. size, location and type of return value
6. registers that the function code used
7. selected GPR bank on entry and exit
8. RAM memory summary for entire function
9. hardware stack requirements
10. functions called by this function
11. which functions call this function
12. how the function was encoded

6.3.4 Switch Statement Information

Along with the generated code for each `switch` statement is information about how that statement was encoded. There are several strategies the compiler can use for `switch` statements. The compiler determines the appropriate strategy (see 5.5.3 Switch Statements) or you can indicate a preference for a particular type of strategy using a pragma (see 5.13.3.10 The #pragma Switch Directive). The information printed will look similar to the figure shown below.

Figure 6-3. Switch Statement Information

```
; Switch size 1, requested type "space"
; Number of cases is 4, Range of values is 0 to 254
; switch strategies available:
; Name         Instructions Cycles
; simple_byte           13     7 (average)
; jumptable            260     6 (fixed)
; rangetable           259     6 (fixed)
; spacedrange          516     9 (fixed)
; locatedrange         255     3 (fixed)
;      Chosen strategy is simple_byte
```

1 size of the switched value
2 number and range of the case values
3 all switch strategies and their attributes
4 the strategy choosen for this switch statement

6.3.5 Pointer Reference Graph

Other important information contained in the assembly list file is the pointer reference graph (look for pointer list with targets: in the list file). This is a list of each pointer contained in the program and each target the pointer can reference through the program. The size and type of each target is indicated, as well as the size and type of the pointer variable itself.

For example, the following shows a pointer called `task_tmr` in the C code. It is local to the function `timer_intr()`. It is also a pointer to an `unsigned int`, and it is one byte wide. There is only one target to this pointer and it is the member `timer_count` in the structure called `task`. This target variable resides in the BANK0 class and is two bytes wide.

```
timer_intr@task_tmr   PTR unsigned int  size(1); Largest target is 2
-> task.timer_count(BANK0[2]),
```

The pointer reference graph shows both pointers to data objects and pointers to functions.

6.3.6 Call Graph

The other important information in the assembly list file is the call graph. This is produced for all 8-bit devices, which can use a compiled stack to facilitate stack-based variables (function parameters, auto and temporary variables). See 5.2.4.2.1 Compiled Stack Operation, for more detailed information on compiled stack operation.

Call graph tables, showing call information on a function-by-function basis, are presented in the map file, followed by more traditional call graphs for the entire program. The call graphs are built by the code generator, and are used to allow overlapping of functions' auto-parameter blocks (APBs) in the compiled stack. The call graphs are not used when functions use the software stack. You can obtain the following information from studying the call graph.

- The functions in the program that are “root” nodes marking the top of a call tree, and that are called spontaneously.
- The functions that the linker deemed were called, or can have been called, during program execution (and those which were called indirectly via a pointer).
- The program's hierarchy of function calls.
- The size of the auto and parameter areas within each function's APB.
- The offset of each function's APB within the compiled stack.
- The estimated call tree depth.

These features are discussed in sections that follow.

6.3.6.1 Call Graph Tables

A typical call graph table can look like the extract shown in the diagram below. Look for **Call Graph Tables:** in the list file.

Figure 6-4. Call Graph Form

```
Call Graph Tables:

 ---------------------------------------------------------------------------------
 (Depth) Function              Calls       Base Space   Used Autos Params    Refs
 ---------------------------------------------------------------------------------
 (0) _main                                                12    12      0   34134
                                            43 BANK0       5     5      0
                                             0 BANK1       7     7      0
                              _aOut
                            _initSPI
 ---------------------------------------------------------------------------------
 (1) _aOut                                                 2     0      2      68
                                             2 BANK0       2     0      2
                               _SPI
                   _GetDACValue (ARG)
 ---------------------------------------------------------------------------------
 (1) _initSPI                                              0     0      0       0
 ---------------------------------------------------------------------------------
 (2) _SPI                                                  2     2      0      23
                                             0 BANK0       2     2      0
...

 Estimated maximum stack depth 6
 ---------------------------------------------------------------------------------
```

The graph table starts with the function `main()`. Note that the function name will always be shown in the assembly form, thus the function `main()` appears as the symbol `_main`. The `main()` function is always the root of one call tree. Interrupt functions will form separate trees.

All the functions that `main()` calls, or can call, are shown in the lines below, in the **Calls** column. So in this example, `main()` calls `aOut()` and `initSPI()`. These have been grouped in the orange box in the figure. If a star (`*`) appears next to the function's name, this implies the function has been called indirectly via a pointer. A function's inclusion into the call graph does not imply the function was actually called, but there is a possibility that the function was called. For example, code such as:

```
int test(int a) {
    if(a)
        foo();
    else
        bar();
}
```

will list `foo()` and `bar()` under `test()`, as either can be called. If a is always true, then the function `bar()` will never be called, even though it appears in the call graph.

In addition to the called functions, information relating to the memory allocated in the compiled stack for `main()` is shown. This memory will be used for the stack-based variables that are defined in `main()`, as well as a temporary location for the function's return value, if appropriate.

In the orange box for `main()` you can see that it defines 12 auto and temporary variable (under the **Autos** column). It defines no parameters under the **Params** column. There is a total of 34134 references in the assembly code to local objects in `main()`, shown under the **Refs** column. The **Used** column indicates the total number of bytes of local storage, i.e., the sum of the **Autos** and **Params** columns.

Rather than the compiled stack being one block of memory in one memory space, it can be broken up into multiple blocks placed in different memory spaces to utilize all of the available memory on the target device. This breakdown is shown under the memory summary line for each function. In this example, it shows that 5 bytes of auto objects for `main()` are placed in the bank 0 component of the compiled stack (**Space** column), at an offset of 43 (**Base** column) into this stack. It also shows that 7 bytes of auto objects were placed in the bank 1 data component of the compiled stack at an offset of 0. The name listed under the **Space** column, is the same name as the linker class which will hold this section of the stack.

Below the information for `main()` (outside the orange box) you will see the same information repeated for the functions that `main()` called, i.e., `aOut()` and `initSPI()`. For clarity, only the first few functions of this program are shown in the figure.

Before the name of each function (in brackets) is the call stack depth for that particular function. A function can be called from many locations in a program, and the stack depth could be different at each of those locations. The maximum call depth is always shown for a function, regardless of its position in the call table. The `main()` function will always have a depth of 0. The starting call depth for interrupt functions assumes a worst case and will start at the start depth of the previous call graph tree plus one. If a function makes recursive calls, the stack depth is marked as a question mark, **(?)**, and **(recursive)** is printed following its name.

After each tree in the call graph, there is an indication of the maximum stack depth that might be realized by that tree. For more detailed information on stack depths, consider using the stack guidance feature, described in 5.2.15 Stack Guidance. Stack depths are not printed if any functions in the graph use the software stack. In this case, a single stack depth estimate is printed for the entire program at the end of the graphs. If there are recursive function calls in a program, the maximum stack depth is indicated as being unknown due to recursion. In the example shown, the estimated maximum stack depth is 6. Check the associated data sheet for the depth of your device's hardware stack (see 5.2.4.1 Function Return Address Stack). The stack depth indicated can be used as a guide to the stack usage of the program. No definitive value can be given for the program's total stack usage for several reasons:

- Certain parts of the call tree may never be reached, reducing that tree's stack usage.
- The exact contribution of interrupt (or other) trees to the `main()` tree cannot be determined as the point in main's call tree at which the interrupt (or other function invocation) will occur cannot be known. The compiler assumes the worst case situation of interrupts occurring at the maximum `main()` depth.
- The assembler optimizer may have replaced function calls with jumps to functions, reducing that tree's stack usage.
- The assembler's procedural abstraction optimizations can have added in calls to abstracted routines, increasing the stack depth. Checks are made to ensure this does not exceed the maximum stack depth.
- Functions which are inlined are not called, reducing the stack usage.

The compiler can be configured to manage the hardware stack for PIC10/12/16 devices only (see 4.6.1.23 Stackcall Option). When this mode is selected, the compiler will convert calls to jumps if it thinks the maximum stack depth of the device is being exceeded. The stack depth estimate listed in the call table will reflect the stack savings made by this feature, and thus, the stack depth and call depth will not be the same.

Note that `main()` is jumped to by the runtime startup, not called; so, `main()` itself does not consume a level of stack.

The code generator produces a warning if the maximum stack depth appears to have been exceeded and the stack is not being managed by the compiler. For the above reasons, this warning, too, is intended to be a only a guide to potential stack problems.

6.3.6.2 Call Graph Critical Paths

Immediately prior to the call graph tables in the list file are the critical paths for memory usage identified in the call graphs. A critical path is printed for each memory space and for each call graph. Look for a line similar to Critical Paths under `_main` in BANK0, which, for this example, indicates the critical path for the `main()` function (the root of one call graph) in bank 0 memory. There will be one call graph for the function `main()` and another for each interrupt function. Each of these will appear for every memory space the device defines.

A critical path here represents the biggest range of APBs stacked together in a contiguous block. Essentially, it identifies those functions whose APBs are contributing to the program's memory usage in that particular memory space. If you can reduce the memory usage of these functions in the corresponding memory space, then you will affect the program's total memory usage in that memory space.

This information can be presented as follows.

```
;; Critical Paths under _main in BANK0
;;
;;   _main->_foobar
;;   _foobar->___flsub
;;   ___flsub->___fladd
```

In this example, it shows that of all the call graph paths starting from the function main, the path in which `main()` calls `foobar()`, which calls `__flsub()`, which calls `__fladd()`, is using the largest block of memory in bank 0 RAM. The exact memory usage of each function is shown in the call graph tables.

The memory used by functions that are not in the critical path will overlap entirely with that in the critical path. Reducing the memory usage of these will have no impact on the memory usage of the entire program.

6.3.6.3 Call Graph Graphs

Following the call tables are the call graphs, which show the full call tree for `main()` and any interrupt functions. This is a subset of the information presented in the call tables, and it is shown in a different form. The call graphs will look similar to the one shown below.

CALL GRAPH GRAPHS

```
 Call Graph Graphs:

_main (ROOT)
  _initSPI
  _aOut
    _SPI
    _GetDACValue
      ___ftadd
        ___ftpack
        ___ftmul (ARG)
...
```

Indentation is used to indicate the call depth. In the diagram, you can see that `main()` calls `aOut()`, which in turn calls `GetDACValue()`, which in turn calls the library function `__ftadd()`, etc. If a star (`*`) appears next to the function's name, this implies that the function has been called indirectly via a pointer.

6.3.6.4 Arg Nodes

In both the call trees and the call graph itself, you can see functions listed with the annotation (`ARG`) after its name. This implies that the call to that function at that point in the call graph is made to obtain an argument to another function. For example, in the following code snippet, the function `input()` is called to obtain an argument value to the function `process()`.

```
result = process(input(0x7));
```

For such code, if it were to appear inside the `main()` function, the call graph would contain the following.

```
_main (ROOT)
  _input
  _process
    _input (ARG)
```

This indicates that `main()` calls `input()` and `main()` also calls `process()`, but `input()` is also called as an argument expression to `process()`.

These argument nodes in the graph do not contribute to the overall stack depth usage of the program, but they are important for the creation of the compiled stack. The call depth stack usage of the tree indicated above would only be 1, not 2, even though the argument node function is at an indicated depth of 2. This is because there is no actual reentrancy in terms of an actual call and a return address being stored on the hardware stack.

The compiler must ensure that the parameter area for a function and any of its ‘argument functions’ must be at unique addresses in the compiled stack to avoid data corruption. Note that a function’s return value is also stored in its parameter area; so that must to be considered by the compiler even if there are no parameters. A function’s parameters become ‘active’ before the function is actually called (when the arguments are passed) and its return value location remains ‘active’ after the function has returned (while that return value is being processed).

In terms of data allocation, the compiler assumes a function has been ‘called’ the moment that any of its parameters have been loaded and is still considered ‘called’ up until its return value is no longer required. Thus, the definition for ‘reentrancy’ is much broader when considering data allocation than it is when considering stack call depth.

6.3.7 Symbol Table

At the bottom of each assembly list file is a symbol table. This differs from the symbol table presented in the map file (see 7.3.2.6 Symbol Table) in two ways:

- Only symbols associated with the assembly module, from which the list file is produced (as opposed to the entire program) are listed.
- Local as well as global symbols associated with that module are listed.

Each symbol is listed along with the address it has been assigned.

7. Linker

This chapter describes the operation and the usage of the linker.

The application name of the linker is `hlink`. In most instances it will not be necessary to invoke the linker directly, as the compiler driver, `xc8-cc`, will automatically execute the linker with all the necessary arguments. Using the linker directly is not simple, and should be attempted only by those with a sound knowledge of the requirements of the linking process. If psects are not linked correctly, code failure can result.

7.1 Operation

A command to the linker takes the following form:

```
hlink [options] files
```

The *options* are zero or more case-insensitive linker options, each of which modifies the behavior of the linker in some way. The *files* is one or more object files and zero or more library files (`.a` extension).

The options recognized by the linker are listed in the table below and are discussed in the paragraphs that follow.

Table 7-1. Linker Command-line Options

Option	Effect
-8	Use 8086 style segment: offset address form.
-A*class=low-high* ,...	Specify address ranges for a class.
-C*psect=class*	Specify a class name for a global psect.
-C*baseaddr*	Produce binary output file based at baseaddr.
-D*class=delta*	Specify a class delta value.
-D*symfile*	Produce old-style symbol file.
-E*errfile*	Write error messages to errfile.
-F	Produce `.o` file with only symbol records.
-G *spec*	Specify calculation for segment selectors.
-H *symfile*	Generate symbol file.
-H+ *symfile*	Generate enhanced symbol file.
-I	Ignore undefined symbols.
-J *num*	Set maximum number of errors before aborting.
-K	Prevent overlaying function parameter and auto areas.
-L	Preserve relocation items in .o file.
-LM	Preserve segment relocation items in `.o` file.
-N	Sort symbol table in map file by address order.
-Nc	Sort symbol table in map file by class address order.
-Ns	Sort symbol table in map file by space address order.
-M*mapfile*	Generate a link map in the named file.
-O*outfile*	Specify name of output file.
-P*spec*	Specify psect addresses and ordering.

...........continued	
Option	**Effect**
-Q*processor*	Specify the device type (for cosmetic reasons only).
-S	Inhibit listing of symbols in symbol file.
-S*class*=*limit*[,*bound*]	Specify address limit, and start boundary for a class of psects.
-U*symbol*	Pre-enter symbol in table as undefined.
-V*avmap*	Use file avmap to generate an Avocet format symbol file.
-W*warnlev*	Set warning level (-9 to 9).
-W*width*	Set map file width (>=10).
-X	Remove any local symbols from the symbol file.
-Z	Remove trivial local symbols from the symbol file.
--DISL=*list*	Specify disabled messages.
--EDF=*path*	Specify message file location.
--EMAX=*number*	Specify maximum number of errors.
--NORLF	Do not relocate list file.
--VER	Print version number and stop.

If the standard input is a file, then this file is assumed to contain the command-line argument. Lines can be broken by leaving a backslash `\` at the end of the preceding line. In this fashion, `hlink` commands of almost unlimited length can be issued. For example, a link command file called `x.lnk` and containing the following text:

```
-Z -Ox.o -Mx.map \
-Ptext=0,data=0/,bss,nvram=bss/. \
x.o y.o z.o
```

can be passed to the linker by one of the following:

```
hlink @x.lnk
hlink < x.lnk
```

Several linker options require memory addresses or sizes to be specified. The syntax for all of these is similar. By default, the number is interpreted as a decimal value. To force interpretation as a HEX number, a trailing `H`, or `h`, should be added. For example, `765FH` will be treated as a HEX number.

Typically, you will be using either the MPLAB XC8 C compiler (`xc8-cc`) or the MPLAB XC8 PIC Assembler (`pic-as`) driver, which will invoke the linker when you build, passing it a set of default linker options. As the linker is not executed directly in these situations, if you need to modify or supply additional linker options, you can do so using either the `-Wl` (see 4.6.11.8 Wl: Pass Option To The Linker, Option) or `-Xlinker` (see 4.6.11.9 Xlinker Option) driver options.

7.1.1 A: Define Linker Class

The `-Aclass=low-high` option allows one or more of the address ranges to be assigned a linker class, so that psects can be placed anywhere in this class. Ranges do not need to be contiguous. For example:

```
-ACODE=1020h-7FFEh,8000h-BFFEh
```

specifies that the class called `CODE` represents the two distinct address ranges shown.

Psects can be placed anywhere in these ranges by using the `-P` option and the class name as the address (see 7.1.17 P: Position Psect), for example:

```
-PmyText=CODE
```

Alternatively, any psect that is made part of the `CODE` class, when it is defined (see 6.1.9.36.3 Class Flag), will automatically be linked into this range, unless they are explicitly located by another option.

Where there are a number of identical, contiguous address ranges, they can be specified with a repeat count following an `x` character. For example:

```
-ACODE=0-0FFFFhx16
```

specifies that there are 16 contiguous ranges, each 64k bytes in size, starting from address zero. Even though the ranges are contiguous, no psect will straddle a 64k boundary, thus this can result in different psect placement to the case where the option

```
-ACODE=0-0FFFFFh
```

had been specified, which does not include boundaries on 64k multiples.

The `-A` option does not specify the memory space associated with the address. Once a psect is allocated to a class, the space value of the psect is then assigned to the class (see 6.1.9.36.18 Space Flag).

7.1.2 C: Associate Linker Class To Psect

The `-Cpsect=class` option allows a psect to be associated with a specific class. Normally, this is not required on the command line because psect classes are specified in object files (see 6.1.9.36.3 Class Flag).

7.1.3 D: Define Class Delta Value

The `-Dclass=delta` option allows the `delta` value for psects that are members of the specified class to be defined. The *delta* value should be a number. It represents the number of bytes per addressable unit of objects within the psects. Most psects do not need this option as they are defined with a delta value (see 6.1.9.36.4 Delta Flag).

7.1.4 D: Define Old Style Symbol File

Use the `-Dsymfile` option to produce an old-style symbol file. An old-style symbol file is an ASCII file, where each line has the link address of the symbol followed by the symbol name.

7.1.5 E: Specify Error File

The `-Eerrfile` option makes the linker write all error messages to the specified file instead of the screen, which is the default standard error destination.

7.1.6 F: Produce Symbol-only Object FIle

Normally the linker will produce an object file that contains both program code and data bytes, and symbol information. Sometimes you want to produce a symbol-only object file that can be used again in a subsequent linker run to supply symbol values. The `-F` option suppresses data and code bytes from the output file, leaving only the symbol records.

This option can be used when part of one project (i.e., a separate build) is to be shared with another, as might be the case with a bootloader and application. The files for one project are compiled using this linker option to produce a symbol-only object file. That file is then linked with the files for the other project.

7.1.7 G: Use Alternate Segment Selector

When linking programs using segmented, or bank-switched psects, there are two ways the linker can assign segment addresses, or selectors, to each segment. A segment is defined as a contiguous group of psects where each psect in sequence has both its link and load addresses concatenated with the previous psect in the group. The segment address or selector for the segment is the value derived when a segment type relocation is processed by the linker.

By default the segment selector is generated by dividing the base load address of the segment by the relocation quantum of the segment, which is based on the `reloc=` flag value given to psects at the assembler level (see 6.1.9.36.16 Reloc Flag). The `-Gspec` option allows an alternate method for calculating the segment selector. The argument to `-G` is a string similar to:

```
A/10h-4h
```

where `A` represents the load address of the segment and `/` represents division. This means “Take the load address of the psect, divide by 10 HEX, then subtract 4.” This form can be modified by substituting `N` for `A`, `*` for `/` (to represent multiplication) and adding, rather than subtracting, a constant. The token `N` is replaced by the ordinal number of the segment, which is allocated by the linker. For example:

```
N*8+4
```

means “take the segment number, multiply by 8, then add 4.” The result is the segment selector. This particular example would allocate segment selectors in the sequence 4, 12, 20, ... for the number of segments defined.

The selector of each psect is shown in the map file (see Section 6.4.2.2 “Psect Information Listed by Module”).

7.1.8 H: Generate Symbol File

The `-Hsymfile` option instructs the linker to generate a symbol file. The optional argument *symfile* specifies the name of the file to receive the data. The default file name is `l.sym`.

7.1.9 H+: Generate Enhanced Symbol FIle

The `-H+symfile` option will instruct the linker to generate an enhanced symbol file, which provides, in addition to the standard symbol file, class names associated with each symbol and a segments section which lists each class name and the range of memory it occupies. This format is recommended if the code is to be run in conjunction with a debugger. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is `l.sym`.

7.1.10 I: Ignore Undefined Symbols

Usually, failure to resolve a reference to an undefined symbol is a fatal error. Using the `-I` option causes undefined symbols to be treated as warnings, instead.

7.1.11 J: Specify Maximum Error Count

The linker will stop processing object files after a certain number of errors (other than warnings). The default number is 10, but the `-Jerrcount` option allows this to be altered.

7.1.12 L: Allow Load Relocation

When the linker produces an output file it does not usually preserve any relocation information, since the file is now absolute. In some circumstances a further “relocation” of the program is done at load time. The `-L` option generates, in the output file, one null relocation record for each relocation record in the input.

7.1.13 LM: Allow Segment Load Relocation

Similar to the `-L` option, the `-LM` option preserves relocation records in the output file, but only segment relocations.

7.1.14 M: Generate Map FIle

The `-Mmapfile` option causes the linker to generate a link map in the named file, or on the standard output, if the file name is omitted. The format of the map file is illustrated in 7.3 Map Files.

7.1.15 N: Specify Symbol Table Sorting

By default the symbol table in the map file is sorted by name. The `-N` option causes it to be sorted numerically, based on the value of the symbol. The `-Ns` and `-Nc` options work similarly except that the symbols are grouped by either their space value, or class.

7.1.16 O: Specify Output Filename

This option allows specification of an output file name for the object file.

7.1.17 P: Position Psect

Psects are linked together and assigned addresses based on information supplied to the linker via `-Pspec` options. The argument to the `-P` option consists of comma-separated sequences with the form:

```
-Ppsect=linkaddr+min/loadaddr+min,psect=linkaddr/loadaddr,...
```

All values can be omitted, in which case a default will apply, depending on previous values. The link address of a psect is the address at which it can be accessed at runtime. The load address is the address at which the psect starts within the output file (HEX or binary file etc.), but it is rarely used by 8-bit PIC devices. The addresses specified can be numerical addresses, the names of other psects, classes, or special tokens.

This argument to this option often contains a comma, so it you are passing this option to the linker from either the MPLAB XC8 C compiler (`xc8-cc`) or the MPLAB XC8 PIC Assembler (`pic-as`) drivers, you might need to use the `-Xlinker` driver option (see 4.6.11.9 Xlinker Option) rather than the `-Wl` option (see 4.6.11.8 Wl: Pass Option To The Linker, Option) to avoid the comma in the option argument causing unexpected results.

Examples of the basic and most common forms of this option are:

```
-Ptext10=02000h
```

which places (links) the starting address of psect `text10` at address 0x2000;

```
-PmyData=AUXRAM
```

which places the psect `myData` anywhere in the range of addresses specified by the linker class `AUXRAM` (which would need to be defined using the `-A` option, see 7.1.1 A: Define Linker Class), and

```
-PstartCode=0200h,endCode
```

which places `endCode` immediately after the end of `startCode`, which will start at address 0x200.

The additional variants of this option are rarely needed; but, are described below.

If a link or load address cannot be allowed to fall below a minimum value, the `+min`, suffix indicates the minimum address.

If the link address is a negative number, the psect is linked in reverse order with the top of the psect appearing at the specified address minus one. Psects following a negative address will be placed before the first psect in memory.

If the load address is omitted entirely, it defaults to the link address. If the slash `/` character is supplied with no address following, the load address will concatenate with the load address of the previous psect. For example, after processing the option:

```
-Ptext=0,data=0/,bss
```

the `text` psect will have a link and load address of 0; data will have a link address of 0 and a load address following that of text. The `bss` psect will concatenate with data in terms of both link and load addresses.

A load address specified as a dot character, "`.`" tells the linker to set the load address to be the same as the link address.

The final link and load address of psects are shown in the map file (see 7.3.2.2 Psect Information Listed By Module).

7.1.18 Q: Specify Device

The `-Qprocessor` option allows a device type to be specified. This is purely for information placed in the map file. The argument to this option is a string describing the device. There are no behavioral changes attributable to the device type.

7.1.19 S: Omit Symbol Information Form Symbol File

The `-S` option prevents symbol information from being included in the symbol file produced by the linker. Segment information is still included.

7.1.20 S: Place Upper Address Limit On Class

A class of psects can have an upper address limit associated with it. The following example of the `-Sclasslimit[,bound]=` option places a limit on the maximum address of the `CODE` class of psects to one less than 400h.

```
-SCODE=400h
```

Note that to set an upper limit to a psect, this must be set in assembler code using the psect `limit` flag, (see 6.1.9.36.8 Limit Flag).

If the *bound* (boundary) argument is used, the class of psects will start on a multiple of the bound address. This example below places the `FARCODE` class of psects at a multiple of 1000h, but with an upper address limit of 6000h.

```
-SFARCODE=6000h,1000h
```

7.1.21 U: Add Undefined Symbol

The `-Usymbol` option will enter the specified symbol into the linker's symbol table as an undefined symbol. This is useful for linking entirely from libraries, or for linking a module from a library where the ordering has been arranged so that by default a later module will be linked.

7.1.22 V: Produce Avocet Symbol File

To produce an Avocet format symbol file, the linker needs to be given a map file using the `-Vavmap` option to allow it to map psect names to Avocet memory identifiers. The *avmap* file will normally be supplied with the compiler, or created automatically by the compiler driver as required.

7.1.23 W: Specify Warning Level/Map Width

The `-Wnum` option can be used to set the warning level, in the range -9 to 9, or the width of the map file, for values of *num* >= 10.

`-W9` will suppress all warning messages. `-W0` is the default. Setting the warning level to -9 (`-W-9`) will give the most comprehensive warning messages.

7.1.24 X: Omit Local Symbols From Symbol File

Local symbols can be suppressed from a symbol file with the `-X` option. Global symbols will always appear in the symbol file.

7.1.25 Z: Omit Trivial Symbols From Symbol File

Some local symbols are compiler generated and not of interest in debugging. The `-Z` option will suppress from the symbol file all local symbols that have the form of a single alphabetic character, followed by a digit string. The set of letters that can start a trivial symbol is currently "`klfLSu`". The `-Z` option will strip any local symbols starting with one of these letters, and followed by a digit string.

7.1.26 Disl

The `--disl=messages` option is mainly used by the command-line driver, `xc8-cc`, to disable particular message numbers. It takes a comma-separate list of message numbers that will be disabled during compilation.

See the Compiler Messages section for full information about the compiler's messaging system.

7.1.27 Edf

The `--edf=file` option is mainly used by the command-line driver, `xc8-cc`, to specify the path of the message description file. The default file is located in the `dat` directory in the compiler's installation directory.

See the Compiler Messages section for full information about the compiler's messaging system.

7.1.28 Emax

The `--emax=number` option is mainly used by the command-line driver, `xc8-cc`, to specify the maximum number of errors that can be encountered before the assembler terminates. The default number is 10 errors.

This option is applied if compiling using `xc8-cc`, the command-line driver, and the `-fmax-errors` driver option.

See the Compiler Messages section for full information about the compiler's messaging system.

7.1.29 Norlf

Use of the `--norlf` option prevents the linker applying fixups to the assembly list file produced by the assembler. This option is normally using by the command line driver, `xc8-cc`, when performing pre-link stages, but is omitted when performing the final link step so that the list file shows the final absolute addresses.

If you are attempting to resolve fixup errors, this option should be disabled so as to fix up the assembly list file and allow absolute addresses to be calculated for this file. If the compiler driver detects the presence of a preprocessor macro `__DEBUG`, which is equated to 1, then this option will be disabled when building. This macro is set when choosing a Debug build in MPLAB X IDE. So, always have this option selected if you encounter such errors.

7.1.30 Ver

The `--ver` option prints information stating the version and build of the linker. The linker will terminate after processing this option, even if other options and files are present on the command line.

7.2 Psects and Relocation

The linker can read both relocatable object files (`.o` extension) and object-file libraries (`.a` extension). Library files are a collection of object files packaged into a single unit and once unpacked, are processed in the same way as individual object files.

Each object file consists of a number of records. Each record has a type that indicates what sort of information it holds. Some record types hold general information about the target device and its configuration, other records types can hold data; and others, program debugging information.

A lot of the information in object files relates to psects (program sections). Psects are an assembly domain construct and are essentially a block of something, either instructions or data. Everything that contributes to the program is located in a psect. See 6.1.8 Program Sections for an introductory guide. There is a particular record type that is used to hold the data in psects. The bulk of each object file consists of psect records containing the executable code and some objects.

The linker performs the following tasks.

- Combining the content of all referenced relocatable object files into one.
- Relocation of psects contained in the object files into the available device memory.
- Fixup of symbolic references in content of the psects.

Relocation consists of allocating the psects into the memory of the target device.

The target device memory specification is passed to the linker by the way of linker options. These options are generated by the command-line driver, `xc8-cc`. There are no linker scripts or means of specifying options in any source file. The default linker options rarely need adjusting. But they can be changed, if required, with caution, using the driver option `-Wl`, (see 4.6.11.8 Wl: Pass Option To The Linker, Option).

Once the psects have been placed at their final memory locations, symbolic references made within the psect can be replaced with absolute values. This is a process called fixup.

The output of the linker is a single object file. This object file is absolute, since relocation is complete and all code and objects have been assigned an address.

7.3 Map Files

The map file contains information relating to the memory allocation of psects and the addresses assigned to symbols within those psects.

7.3.1 Map File Generation

If compilation is being performed via MPLAB X IDE, a map file is generated by default. If you are using the driver from the command line, use the `-Wl,-Map` option to request that the map file be produced (see 4.6.11.8 Wl: Pass Option To The Linker, Option). Map files are typically assigned the extension `.map`.

Map files are produced by the linker application. If the build is stopped before the linker is executed, then no map file is produced. A map file is produced, even if the linker generates errors and this partially-complete file can help you track down the cause of these errors. However, if the linker did not run to completion, due to too many errors or a fatal error, the map file will not be created. You can use the `-fmax-errors` driver option to increase the number of errors allowed before the linker exits.

7.3.2 Contents

The sections in the map file, in order of appearance, are as follows.

- The compiler name and version number.
- A copy of the command line used to invoke the linker.
- The version number of the object code in the first file linked.
- The machine type.
- A psect summary sorted by the psect's parent object file.
- A psect summary sorted by the psect's CLASS.
- A segment summary.
- Unused address ranges summary.
- The symbol table.
- Information summary for each function.
- Information summary for each module.

Portions of an example map file, along with explanatory text, are shown in the following sections.

7.3.2.1 General Information

At the top of the map file is general information relating to the execution of the linker.

When analyzing a program, always confirm the compiler version number shown at the very top of the map file to ensure you are using the compiler you intended to use.

The device selected with the `-mcpu` option (see 4.6.1.5 Cpu Option), or the one selected in your IDE, should appear after the **Machine type** entry.

The object code version relates to the file format used by relocatable object files produced by the assembler. Unless either the assembler or linker have been updated independently, this should not be of concern.

A typical map file might begin something like the following cut down example.

```
Linker command line:
--edf=/Applications/Microchip/XC8/2.20/dat/en_msgs.txt -cs -h+main.sym -z \
-Q16F946 -ol.o -Mmain.map -ver=XC8 -ACONST=00h-0FFhx32 \
-ACODE=00h-07FFhx4 -ASTRCODE=00h-01FFFh -AENTRY=00h-0FFhx32 \
-ASTRING=00h-0FFhx32 -ACOMMON=070h-07Fh -ABANK0=020h-06Fh \
-ABANK1=0A0h-0EFh -ABANK2=0120h-016Fh -ABANK3=01A0h-01EFh \
-ARAM=020h-06Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh \
-AABS1=020h-07Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh -ASFR0=00h-01Fh \
-ASFR1=080h-09Fh -ASFR2=0100h-011Fh -ASFR3=0180h-019Fh \
-preset_vec=00h,intentry,init,end_init -ppowerup=CODE -pfunctab=CODE \
-ACONFIG=02007h-02007h -pconfig=CONFIG -DCONFIG=2 -AIDLOC=02000h-02003h \
-pidloc=IDLOC -DIDLOC=2 -AEEDATA=00h-0FFh/02100h -peeprom_data=EEDATA \
-DEEDATA=2 -DCODE=2 -DSTRCODE=2 -DSTRING=2 -DCONST=2 -DENTRY=2 -k \
startup.o main.o

Object code version is 3.10

Machine type is 16F946
```

The information following **Linker command line:** shows all the command-line options and files that were passed to the linker for the last build. Remember, these are linker options, not command-line driver options.

The linker options are necessarily complex. Fortunately, they rarely need adjusting from their default settings. They are formed by the command-line driver, `xc8-cc`, based on the selected target device and the specified driver options. You can often confirm that driver options were valid by looking at the linker options in the map file. For example, if you ask the driver to reserve an area of memory, you should see a change in the linker options used.

If the default linker options must be changed, this can be done indirectly through the driver using the driver `-Wl` option (see 4.6.11.8 Wl: Pass Option To The Linker, Option). If you use this option, always confirm the change appears correctly in the map file.

7.3.2.2 Psect Information Listed By Module

The next section in the map file lists those modules that have made a contribution to the output and information regarding the psects that these modules have defined.

This section is heralded by the line that contains the headings:

```
Name   Link  Load  Length  Selector  Space  Scale
```

Under this on the far left is a list of object files (`.o` extension). Both object files that were generated from source modules and those extracted from object library files (`.a` extension) are shown. In the latter case, the name of the library file is printed before the object file list.

Next to the object file are the psects (under the **Name** column) that were linked into the program from that object file. Useful information about that psect is shown in the columns, as follows.

The linker deals with two kinds of addresses: link and load. Generally speaking, the **Link** address of a psect is the address by which it is accessed at runtime.

The **Load** address, which is often the same as the link address, is the address at which the psect starts within the output file (HEX or binary file etc.). If a psect is used to hold bits, the load address is irrelevant and is used to hold the link address (in bit units) converted into a byte address instead.

The **Length** of the psect is shown in the units that are used by that psect.

The **Selector** is less commonly used and is of no concern when compiling for PIC devices.

The **Space** field is important as it indicates the memory space in which the psect was placed. For Harvard architecture machines, with separate memory spaces (such as the PIC10/12/16 devices), this field must be used in conjunction with the address to specify an exact storage location. A space of 0 indicates the program memory and a space of 1 indicates the data memory (see 6.1.9.36.18 Space Flag).

The **Scale** of a psect indicates the number of address units per byte. This remains blank if the scale is 1 and shows 8 for psects that hold bit objects. The load address of psects that hold bits is used to display the link address converted into units of bytes, rather than the load address (see 6.1.9.36.2 Bit Flag).

For example, the following appears in a map file.

```
Name      Link  Load  Length Selector  Space  Scale
ext.o     text    3A    3A       22      30      0
          bss     4B    4B       10      4B      1
          rbit    50     A        2       0      1      8
```

This indicates that one of the files that the linker processed was called `ext.o`.

This object file contained a `text` psect, as well as psects called `bss` and `rbit`.

The psect `text` was linked at address 3A and `bss` at address 4B. At first glance, this seems to be a problem, given that text is 22 words long. However, they are in different memory areas, as indicated by the space flag (0 for `text` and 1 for `bss`), and so they do not even occupy the same memory space.

The psect `rbit` contains bit objects, and this can be confirmed by looking at the scale value, which is 8. Again, at first glance it seems that there could be an issue with `rbit` linked over the top of `bss`. Their space flags are the same, but since `rbit` contains bit objects, its link address is in units of bits. The load address field of `rbit` psect displays the link address converted to byte units, i.e., 50h/8 => Ah.

Underneath the object file list there can be a label **COMMON**. This shows the contribution to the program from program-wide psects. For C-based projects, this is where you will find information on the compiled stack.

7.3.2.3 Psect Information Listed By Class

The next section in the map file shows the same psect information but grouped by the psects' class.

This section is heralded by the line that contains the headings:

```
TOTAL   Name  Link  Load  Length
```

Under this are the class names followed by those psects which belong to this class (see 6.1.9.36.3 Class Flag). These psects are the same as those listed by module in the above section; there is no new information contained in this section, just a different presentation.

7.3.2.4 Segment Listing

The class listing in the map file is followed by a listing of segments. Typically this section of the map file can be ignored by the user.

A segment is a conceptual grouping of contiguous psects in the same memory space, and is used by the linker as an aid in psect placement. There is no segment assembler directive and segments cannot be controlled in any way.

This section is heralded by the line that contains the headings:

```
SEGMENTS   Name  Load  Length  Top  Selector  Space  Class
```

The name of a segment is derived from the psect in the contiguous group with the lowest link address. This can lead to confusion with the psect with the same name. Do not read psect information from this section of the map file.

Again, this section of the map file can be ignored.

7.3.2.5 Unused Address Ranges

The last of the memory summaries show the memory that has *not* been allocated and is still available for use.

This section follows the heading:

```
UNUSED ADDRESS RANGES
```

and is followed by a list of classes and the memory that is still available in each class. If there is more than one memory range available in a class, each range is printed on a separate line. Any paging boundaries located within a class are not displayed. But the column Largest block shows the largest contiguous free space (which takes into account any paging in the memory range). If you are looking to see why psects cannot be placed into memory (e.g., cant-find-space type errors) then this is important information to study.

Note that memory can be part of more than one class, thus the total free space is not simply the addition of all the unused ranges.

7.3.2.6 Symbol Table

The next section in the map file alphabetically lists the global symbols that the program defines. This section has the heading:

```
Symbol Table
```

The symbols listed in this table are:

- Global assembly labels
- Global `EQU`/`SET` assembler directive labels
- Linker-defined symbols

Assembly symbols are made global via the `GLOBAL` assembler directive, see 6.1.9.21 Global Directive for more information.

Linker-defined symbols act like `EQU` directives. However, they are defined by the linker during the link process, and no definition for them appears in any source or intermediate file (see 5.14.6 Linker-Defined Symbols).

Each symbol is shown with the psect in which it is defined and the value (usually an address) it has been assigned. There is not any information encoded into a symbol to indicate whether it represents code or data – nor in which memory space it resides.

If the psect of a symbol is shown as (`abs`), this implies that the symbol is not directly associated with a psect. Such is the case for absolute C variables, or any symbols that are defined using an `EQU` directive in assembly.

Note that a symbol table is also shown in each assembler list file. These differ to that shown in the map file as they also list local symbols and they only show symbols defined in the corresponding module.

7.3.2.7 Function Information

Following the symbol table is information relating to each function in the program. This information is identical to the function information displayed in the assembly list file. However, the information from all functions is collated in the one location.

See the Function Information section for detailed descriptions of this information.

7.3.2.8 Module Information

The final section in the map file shows code usage summaries for each module. Each module in the program will show information similar to the following.

```
Module      Function          Class         Link    Load    Size
main.c
            init              CODE          07D8    0000    1
            main              CODE          07E5    0000    13
            getInput          CODE          07D9    0000    4
main.c estimated size: 18
```

The module name is listed (`main.c` in the above example). The special module name shared is used for data objects allocated to program memory and to code that is not specific to any particular module.

Next, the user-defined and library functions defined by each module are listed along with the class in which that psect is located, the psect’s link and load address, and its size (shown as bytes for PIC18 devices and words for other 8-bit devices).

After the function list is an estimated size of the program memory used by that module.

8. 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.

8.1 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.

8.1.1 Using the Archiver/Librarian

The archiver program is called `xc8-ar` and is used to create and edit library archive files. It has the following basic command format:

```
xc8-ar [options] file.a [file.p1 file.o ...]
```

where *`file`*`.a` represents the library archive being created or edited.

The files following the archive file, if required, are the modules that are required by the command specified and may be either p-code (`.p1`) or object (`.o`) modules. P-code modules must be used for content built from C source; object modules must be used for content built from assembly source.

The *`options`* is zero or more options, tabulated below, that control the program.

Table 8-1. 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.

Object (`.o` extension) and p-code (`.p1` extension) modules can be added to the same archive.

The archiver creates library archives with the modules in the order in which they were given on the command line. When updating an archive, the order of the modules is preserved. Any modules added to an archive will be appended to the end.

The ordering of the modules in an archive is significant to the linker. If an archive contains a module that references a symbol defined in another module in the same archive, the module defining the symbol should come after the module referencing the symbol.

When using the `-d` option, the specified modules will be deleted from the archive. In this instance, it is an error not to supply any module names.

The `-p` option will list the modules within the archive file.

The `-m` option takes a list of module names and re-orders the matching modules in the archive file so that they have the same order as the one listed on the command line. Modules that are not listed are left in their existing order, and will appear after the re-ordered modules.

8.1.1.1 Examples

Here are some examples of usage of the librarian. The following command:

```
xc8-ar -r myLib.a ctime.p1 init.p1
```

creates a library called `myLib.a` that contains the modules `ctime.p1` and `init.p1`

The following command deletes the object module `a.p1` from the library `lcd.a`:

```
xc8-ar -d lcd.a a.p1
```

8.2 Hexmate

The Hexmate application is a post-link-stage utility designed to manipulate Intel HEX files.

Hexmate is automatically invoked by the compiler driver, but it can also be executed as a stand-alone application, if required.

8.2.1 Hexmate Uses

Hexmate can be used for a variety of tasks relating to Intel HEX files. These include the following.

- Merging multiple Intel HEX files into one Intel HEX file.
- Calculating and storing variable-length hash values, such as CRC or SHA.
- Filling unused memory locations with known data sequences.
- Converting INHX32 files to other INHX formats (e.g., INHX8M).
- Detecting specific or partial opcode sequences within a HEX file.
- Finding/replacing specific or partial opcode sequences.
- Providing a map of addresses used in a HEX file.
- Changing or fixing the length of data records in a HEX file.
- Validating checksums within Intel HEX files.

Typical applications for Hexmate might include:

- Merging a bootloader or debug module into a main application at build time.
- Calculating a hash value over a range of program memory and storing its value in program memory or EEPROM.
- Filling unused memory locations with an instruction to send the program counter to a known location if it gets lost.
- Storing a serial number at a fixed address.
- Storing a string (e.g., time stamp) at a fixed address.
- Storing initial values at a particular memory address (e.g., initialize EEPROM).
- Detecting the occurrence of a buggy/restricted instructions.
- Adjusting HEX file to meet the requirements of particular bootloaders.

8.2.2 Hexmate Command-line Options

Hexmate is automatically called by the command line driver, `xc8-cc` to merge any HEX files specified on the command line with the output generated from the program's source files. Some other Hexmate functions can be requested using compiler driver options and without running Hexmate explicitly, but for full control you may run Hexmate on the command line and use the options detailed here.

Run Hexmate directly with the following command format:

```
hexmate [specs,]file1.hex [... [specs,]fileN.hex] [options]
```

where *file1*.hex through to *fileN*.hex forms a list of input Intel HEX files to merge using Hexmate. The *options* can appear anywhere on the command line and are tabulated below.

If only one HEX file is specified, no merging takes place, but other actions can be performed on the HEX file, as specified by the options.

Hexmate can read and write common 8-, 16-, and 32-bit named formats, which contain only specific subsets of record types. The formats are discussed in 8.2.2.16 Format.

Table 8-2. Hexmate Command-line Options

Option	Effect
`--edf=`*file*	Specify the message description file.
`--emax=`*n*	Set the maximum number of permitted errors before terminating.
`--msgdisable=`*number*	Disable messages with the numbers specified.
`--sla=`*address*	Set the start linear address for a type 5 record.
`--ssa=`*address*	Set the start segment address for a type 3 record.
`--ver`	Display version and build information then quit.
`-addressing=`*units*	Set address fields in all Hexmate options to use word addressing or other.
`-break`	Break continuous data so that a new record begins at a set address.
`-ck=spec`	Calculate and store a hash value.
`-fill=`*spec*	Program unused locations with a known value.
`-find=`*spec*	Search and notify if a particular code sequence is detected.
`-find=`*spec*`,delete`	Remove the code sequence if it is detected (use with caution).
`-find=`*spec*`,replace=`*spec*	Replace the code sequence with a new code sequence.
`-format=`*type*	Specify maximum data record length or select INHX variant.
`-help`	Show all options or display help message for specific option.
`-logfile=`*file*	Save Hexmate analysis of output and various results to a file.
`-mask=`*spec*	Logically AND a memory range with a bitmask.
`-o`*file*	Specify the name of the output file.
`-serial=`*spec*	Store a serial number or code sequence at a fixed address.
`-size`	Report the number of bytes of data contained in the resultant HEX image.
`-string=`*spec*	Store an ASCII string at a fixed address.
`-strpack=`*spec*	Store an ASCII string at a fixed address using string packing.
`-w`*level*	Adjust warning sensitivity.
`+`	Prefix to any option to overwrite other data in its address range, if necessary.

If you are using the driver, `xc8-cc`, to compile your project (or the IDE), a log file is produced by default. It will have the project's name and the extension `.hxl`.

The format or assumed radix of values associated with options are detailed with each option description. Note that any address fields specified in these options are to be entered as HEX file addresses, unless you use the `-addressing` option to change this.

8.2.2.1 Specifications And Filename

Hexmate can process Intel HEX files that use either INHX32 or INHX8M format. Additional specifications can be applied to each HEX file listed on the command line to place restrictions or conditions on how this file should be processed.

If any specifications are used, they must precede the filename. The list of specifications will then be separated from the filename by a comma.

A range restriction can be applied with the specification `r`*`Start`*`-`*`End`*, where *`Start`* and *`End`* are both assumed to be hexadecimal values. Hexmate will only process data within the address range restriction. For example:

```
r100-1FF,myfile.hex
```

will use `myfile.hex` as input, but only process data which is addressed within the range 0x100-1FF (inclusive) from that file.

An address shift can be applied with the specification `s`*`Offset`*, where *`Offset`* is assumed to be an unqualified hexadecimal value. If an address shift is used, data read from this HEX file will be shifted (by the offset specified) to a new address in the output file. The offset can be either positive or negative. For example:

```
r100-1FFs2000,myfile.hex
```

will shift the block of data from 0x100-1FF in `myfile.hex` to the new address range 0x2100-21FF in the output.

Be careful when shifting sections of executable code. Program code should only be shifted if it is position independent.

8.2.2.2 Override Prefix

When the `+` operator precedes an input file or option, the data obtained from that file or the data generated by that option will take priority and be forced into the output file, overwriting any another other data existing at the same addresses.

For example, if `input.hex` contains data at address 0x1000, the option:

```
input.hex +-string@1000="My string"
```

will have the data specified by the `-string` option placed at address 0x1000 in the output file; however:

```
+input.hex -string@1000="My string"
```

will copy the data contained in the input HEX file at address 0x1000 into the final output.

Without this option, Hexmate will issue an error if two sources try to store differing data at the same location.

8.2.2.3 Edf

The `--edf=`*`file`* specifies the message description file to use when displaying warning or error messages. The argument to this option should be the full path to the message file. Hexmate contains an internal copy of the message file, so this option is not normally required. Use this option if you want to specify an alternate file with updated contents.

Message files are shipped with the MPLAB XC8 C Compiler and are located in the compiler's `pic/dat` directory. The English language file is called `en_msgs.txt`.

8.2.2.4 Emax

The `--emax=`*`num`* option sets the maximum number of errors Hexmate will display before execution is terminated, e.g., `--emax=25`. By default, up to 20 error messages will be displayed.

8.2.2.5 Msgdisable

The `--msgdisable=`*`number`* option allows error, warning or advisory messages to be disabled during execution of Hexmate.

The option is passed a comma-separated list of message numbers that are to be disabled. Any error message numbers in this list are ignored unless they are followed by an `:off` argument. For example:

```
--msgdisable=2031,963
```

If the message list is specified as `0`, then all warnings are disabled.

8.2.2.6 Sla Hexmate Option

The `--sla=address` option allows you to specify the linear start address (SLA) in a type 5 record in an INHX32 or INHX032 output file. For example `--sla=0x10000` will ensure the output HEX file will contain a type 5 record with payload 0x10000, e.g.:

```
:040000050001000F6
```

When this option is used, any input SLA records present in the input files are checked for correct syntax and checksum. An error will be issued if they are not valid. These SLA records will then be discarded with *no* warning message. If the output file format is not INHX32 or INHX032, a warning will be issued and no SLA record will be written; otherwise, one SLA record only will appear in the output, containing the value specified by the option.

If this option is not used, any input SLA records present in the input files are checked for correct syntax and checksum. If there is no discrepancy between the addresses specified by these records, one and only one SLA record with that address is written to the output. If there is a conflict between SLA records present in the input files, a warning message will be emitted and *no* SLA record will appear in the output. If there are no SLA records present in the input files, no SLA record will be written to the output.

8.2.2.7 Ssa Hexmate option

The `--ssa=address` option allows you to specify the segment start address (SSA) in a type 3 record in an INHX16 output file. For example `--ssa=0x10000` will ensure that the output HEX file will contain a type 3 record with payload 0x10000, e.g.:

```
:0400000300010000F8
```

When this option is used, any SSA records present in the input files are checked for correct syntax and checksum. An error will be issued if they are not valid. These SSA records will then be discarded with *no* warning message. If the output file format is not INHX16, a warning will be issued and no SSA record will be written; otherwise, one SSA record only will appear in the output, containing the value specified by the option.

If this option is not used, any SSA records present in the input files are checked for correct syntax and checksum. If there are no discrepancy between the addresses specified by these records, one and only one SSA record with that address is written to the output. If there is a conflict between SSA records present in the input files, a warning message will be emitted and *no* SSA record will appear in the output. If there are no SSA records present in the input files, no SSA record will be written to the output.

8.2.2.8 Ver

The `--ver` option will ask Hexmate to print version and build information and then quit.

8.2.2.9 Addressing

The `-addressing=units` option allows the addressing units of any addresses in Hexmate's command line options to be changed from the default value of 1 to a maximum value of 4.

By default, all address arguments specified in Hexmate options are assumed to be byte addresses, as used by Intel HEX files. For example, in the option `-mask=0F@0-FF`, the mask will be performed on any HEX file data from address 0x0 to address 0xFF. In some device architectures, the native addressing format can be something other than byte addressing. For example, a HEX file might contain the bytes 0x0F and 0x55 at addresses 0x200 and 0x201, respectively, but when this HEX file is loaded into a Mid-range PIC device, these bytes will form one word at address 0x100 in the device. In this case, the word value at each device address expands into two byte values at separate addresses in the HEX file. If you prefer to use device addresses with Hexmate options, use this option to specify the mapping between HEX file addresses and device addresses.

This option takes one parameter that indicates the number of HEX file bytes that will be stored in each device address location. The parameter may range from the values 1 thru 4. For 8-bit AVR devices, Baseline, Mid-range, and 24-bit PIC devices, an addressing unit of 2 can be used, if desired, for example, `-addressing=2`. You may then specify device addresses in all Hexmate options. For all other Microchip devices, you would typically use the default addressing unit of 1 byte, for example use `-addressing=1` or omit this option entirely. For these devices, the HEX file and device addresses for any location are the same and no mapping is required.

8.2.2.10 Break

The `-break` option takes a comma-separated list of unqualified hexadecimal addresses. If any of these addresses are encountered in the HEX file, the current data record will conclude and a new data record will recommence from the nominated address.

For example, if the output of Hexmate normally contains:

```
:100000000EEF0AF03412ADDEADDEADDEADDEADDEFC
:10001000ADDEADDEADDEADDEADDEADDEADDEADDE88
:10002000ADDEADDEADDEADDEADDEADDEADDEADDE78
...
```

then if the `-break=16` option is used, the output will become:

```
:100000000EEF0AF03412ADDEADDEADDEADDEADDEFC
:06001000ADDEADDEADDE49
:10001600ADDEADDEADDEADDEADDEADDEADDEADDE82
:10002600ADDEADDEADDEADDEADDEADDEADDEADDE72
...
```

Breaking data records can create a distinction between functionally different areas of the program space. Some HEX file readers depend on records being arranged this way.

8.2.2.11 Ck Hexmate Option

The `-ck` option is for calculating a hash value. The usage of this option is:

```
-ck=start-end@dest[+offset][wWidth][tCode[.Base]][gAlgorithm][pPolynomial][rRevWidth]
[sSkipWidth[.SkipBytes]][oXORvalue]
```

where:

- *`start`* and *`end`* specify the hexadecimal address range over which the hash will be calculated. If these addresses are not a multiple of the data width for checksum and Fletcher algorithms, the value zero will be padded into the relevant input word locations that are missing.
- *`dest`* is the hexadecimal address where the hash result will be stored. This address cannot be within the range of addresses over which the hash is calculated.
- *`offset`* is an optional initial hexadecimal value to be used in the hash calculations. It is not used with SHA algorithms.
- *`Width`* is optional and specifies the decimal width of the result. Results can be calculated for byte-widths of 1 to 4 bytes for most algorithms, but it represents the bit width for SHA algorithms. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order. This width argument is not required with any Fletcher algorithm, as they have fixed widths, but it may be used to alter the default endianism of the result.
- *`Code`* is an optional hexadecimal code sequence that will trail each byte in the result. Use this feature if you need each byte of the hash result to be embedded within an instruction or if the hash value has to be padded to allow the device to read it at runtime. For example, `t34` will embed each byte of the result in a `retlw` instruction (bit sequence 0x34xx) on Mid-range PIC devices. If the code sequence specifies multiple bytes, these are stored in big-endian order after the hash bytes, for example `tAABB` will append 0xAA immediately after the hash byte and 0xBB at the following address. The trailing code specification `t0000` will store two 0x00 bytes after each byte of the hash. The code sequence argument can be optionally followed by `.`*`Base`*, where *`Base`* is the number of bytes of hash to be output before the trailing code sequence is appended. A specification of `t11.2`, for example, will output the byte 0x11 after each *two* bytes of the hash result.
- *`Algorithm`* is a decimal integer to select which Hexmate hash algorithm to use to calculate the result. A list of selectable algorithms is provided in the table below. If unspecified, the default algorithm used is 8-bit checksum addition (algorithm 1).
- *`Polynomial`* is a hexadecimal value which is the polynomial to be used if you have selected a CRC algorithm.

- *RevWidth* is an optional reverse word width. If this is non-zero, then bytes within each word are read in reverse order when calculating a hash value. Words are aligned to the addresses in the HEX file. At present, the width must be 0 or 2. A zero width disables the reverse-byte feature, as if the `r` suboption was not present. This suboption is intended for situations when Hexmate is being used to match a CRC produced by a PIC hardware CRC module that uses the Scanner module to stream data to it. This feature will work with all hash types, but has no effect when using any checksum algorithm (algorithms -4 thru 4).
- *SkipWidth* is an optional skip word width. If this is non-zero, then the byte at the highest address within each word is skipped for the purposes of calculating a hash value. Words are aligned to the addresses in the HEX file. At present, the width must be 0 (which disables the skip feature, as if the `s` suboption was not present) or greater than 1. This skip width argument can be optionally followed by `.`*SkipBytes*, where *SkipBytes* is a number representing the number of bytes to skip in each word, for example `s4.2` will skip the two bytes at the highest addresses in each 4-byte word. To avoid processing the 'phantom' 0x00 bytes added to HEX files by the MPLAB XC16 C Compiler in hash calculations, use `s4`.
- *XORvalue* is a hexadecimal value that will be XORed with the hash result before it stored.

The single letter argument tokens are case insensitive, so for example `w2` and `W2` are both valid width arguments to this option.

A typical example of the use of this option to calculate a checksum is:

```
-ck=0-1FFF@2FFE+2100w-2g2
```

This will calculate a checksum (16-bit addition) over the range 0 to 0x1FFF and program the checksum result at address 0x2FFE. The checksum value will be offset by 0x2100. The result will be two bytes wide and stored in little-endian format.

Note that the reverse and skip features act on words that are aligned to the HEX file addresses, not to the starting byte of data in the sequence being processed. In other words, the positions of the words are not affected by the start and end addresses specified in the `-ck` option. Consider this option:

```
-ck=0-5@100w2g5p1021s2
```

which specifies that when calculating the hash value, every second byte be skipped (`s2`) over HEX addresses 0 thru 5. If it is acting on the HEX record (data underlined):

```
:10000000640025000030A7500760077007800640001C
```

the hash will be calculated from the (hexadecimal) bytes 64, 25, and 03. Processing the same HEX record with an option that uses a different start and end address range (1 thru 6):

```
-ck=1-6@100w2g5p1021s2
```

the hash will be calculated from the (hexadecimal) bytes 25, 03, and 75. These features attempt to mimic data read limitations of code running on the device, and thus the words they use are aligned with device addresses, which are in turn aligned to HEX file addresses.

Table 8-3. Hexmate Hash Algorithm Selection

Selector	Algorithm Description
-5	Reflected cyclic redundancy check (CRC).
-4	Subtraction of 32 bit values from initial value.
-3	Subtraction of 24 bit values from initial value.
-2	Subtraction of 16 bit values from initial value.
-1	Subtraction of 8 bit values from initial value.
1	Addition of 8 bit values from initial value.
2	Addition of 16 bit values from initial value.

...........continued	
Selector	**Algorithm Description**
3	Addition of 24 bit values from initial value.
4	Addition of 32 bit values from initial value.
5	Cyclic redundancy check (CRC).
7	Fletcher's checksum (8 bit calculation, 2-byte result width).
8	Fletcher's checksum (16 bit calculation, 4-byte result width).
10	SHA-2 (currently only SHA256 is supported).

See 8.2.3 Hash Value Calculations for more details about the algorithms that are used to calculate hashes.

8.2.2.12 Fill

The `-fill` option is used for filling unused (unspecified) memory locations in a HEX file with a known value. The usage of this option is:

```
-fill=[wwidth:]fill_expr@address[:end_address]
```

where the arguments have the following meaning:

wwidth signifies the decimal width of each constant in the *fill_expr* and can range from 1 thru 9. If this width is not specified, the default value is two bytes. For example, `-fill=w1:0x55@0:0xF` with fill every unused byte between address 0 and 0xF with the byte value 0x55, for example:

```
:10000000FBEF3FF05555555555555555555555555DB
```

whereas `-fill=w2:0x55@0:0xF` with fill every unused byte between the same addresses with the value 0x0055, for example:

```
:10000000FBEF3FF05500550055005500550055005500D9
```

fill_expr defines the values to fill and consists of *const*, which is a base value to place in the first memory location and optionally with *increment*, which indicates how this base value should change after each use. If the base value specifies more than one byte, these are stored in little-endian byte order. These following show the possible fill expressions:

- *const* fill memory with a repeating constant; i.e., `-fill=0xBEEF@0-0x1FF` fills unused locations starting at address 0 with the values 0xBEEF, 0xBEEF, 0xBEEF, 0xBEEF, etc., for example:

  ```
  :10000000FBEF3FF0EFBEEFBEEFBEEFBEEFBEEFBEC9
  ```

- *const+=increment* fill memory with an incrementing constant; i.e., `-fill=0xBEEF+=1@0:0x1FF` attempts to fills with the values 0xBEEF, 0xBEF0, 0xBEF1, 0xBEF2, etc., for example:

  ```
  :10000000FBEF3FF0F1BEF2BEF3BEF4BEF5BEF6BEAE
  ```

  Note that *const* increments with each location scanned, regardless of whether that location is populated or unused.

- *const-=increment* fill memory with a decrementing constant; i.e.,`-fill=0xBEEF-=0x10@0:0x1FF` attempts to fills with the values 0xBEEF, 0xBEDF, 0xBECF, 0xBEBF, etc., for example:

  ```
  :10000000FBEF3FF0CFBEBFBEAFBE9FBE8FBE7FBE79
  ```

  Note that *const* decrements with each location scanned, regardless of whether that location is populated or unused.

- *const*,*const*,...,*const* fill memory with a list of repeating constants; i.e., -fill=0xDEAD,0xBEEF@0:0x1FF fills with 0xDEAD, 0xBEEF, 0xDEAD, 0xBEEF, etc., for example:

```
:10000000FBEF3FF0ADDEEFBEADDEEFBEADDEEFBE2F
```

@*address* fills a specific address with *fill_expr*; for example, -fill=0xBEEF@0x1000 puts the byte value 0xEF at addresses 0x1000 when the addressing value is set to 1.

```
:01100000EF00
```

If the -addressing=2 option had been additionally used in the above example, the fill option would place 2-bytes at address 0x2000 and 0x2001.

```
:02200000EFBE31
```

:*end_address* optionally specifies an end address of memory to be filled with *fill_expr*; for example, -fill=0xBEEF@0xF0:0xFF puts 0xBEEF in unused addresses between 0 and 0xFF, inclusive.

```
:1000F000EFBEEFBEEFBEEFBEEFBEEFBEEFBEEFBE98
```

If the address range (multiplied by the -addressing value) is not a multiple of the fill value width, the final location will only use part of the fill value, and a warning will be issued.

The fill values are word-aligned so they start on an address that is a multiple of the fill width. Should the fill value be an instruction opcode, this alignment ensures that the instruction can be executed correctly. Similarly, if the total length of the fill sequence is larger than 1 (and even if the specified width is 1), the fill sequence is aligned to that total length. For example the following fill option, which specifies 2 bytes of fill sequence and a starting address that is not a multiple of 2:

```
-fill=w1:0x11,0x22@0x11001:0x1100c
```

will result in the following HEX record, where the starting address was filled with the second byte of the fill sequence due to this alignment.

```
:0C100100221122112211221122112211B1
```

Compare that to when the option is -fill=w1:0x11,0x22@0x1100**0**:0x1100c, which does specify a starting address that is a multiple of 2.

```
:0D100000112211221122112211221122112211A0
```

All fill constants (excluding the width specification) can be expressed in (unsigned) binary, octal, decimal or hexadecimal, as per normal C syntax, for example, 1234 is a decimal value, 0xFF00 is hexadecimal and FF00 is illegal.

8.2.2.13 Find

The -find=*opcode* option is used to detect and log occurrences of an opcode or code sequence. The usage of this option is:

```
-find=Findcode[mMask]@Start-End[/Align][w][t”Title”]
```

where:

- *Findcode* is the hexadecimal code sequence to search for. For example, to find a clrf instruction with the opcode 0x01F1, use 01F1 as the sequence. In the HEX file, this will appear as the byte sequence F1 01, that is 0xF1 at HEX address 0 and 0x01 at HEX address 1.
- *Mask* is optional. It specifies a bit mask applied over the *Findcode* value to allow a less restrictive search. It is entered in little endian byte order.
- Start and *End* limit the address range to search.
- *Align* is optional. It specifies that a code sequence can only match if it begins on an address that is a multiple of this value.

- `w`, if present, will cause Hexmate to issue a warning whenever the code sequence is detected.
- *`Title`* is optional. It allows a title to be given to this code sequence. Defining a title will make log-reports and messages more descriptive and more readable. A title will not affect the actual search results.

All numerical arguments are assumed to be hexadecimal values.

Here are some examples.

The option `-find=1234@0-7FFF/2w` will detect the code sequence 1234h (stored in the HEX file as 34 12) when aligned on a 2 (two) byte address boundary, between 0h and 7FFFh. w indicates that a warning will be issued each time this sequence is found.

In this next example, `-find=1234M0F00@0-7FFF/2wt"ADDXY"`, the option is the same as in last example but the code sequence being matched is masked with 000Fh, so Hexmate will search for any of the opcodes `123`*`x`*h, where *`x`* is any digit. If a byte-mask is used, it must be of equal byte-width to the opcode it is applied to. Any messaging or reports generated by Hexmate will refer to this opcode by the name, `ADDXY`, as this was the title defined for this search.

When requested, a log file will contain the results of all searches. The `-find` option accepts whole bytes of HEX data from 1 to 8 bytes in length. Optionally, `-find` can be used in conjunction with `replace` or `delete` (as described separately).

8.2.2.14 Find And Delete

If the `delete` form of the `-find` option is used, any matching sequences will be deleted from the output file. This implies removal of the data entirely, not replacing it with zero bytes.

To have the `-find` option perform deletion, append `,delete` to the option, for example:

```
-find=ff@7fe0-7fff,delete
```

This function should be used with extreme caution and is not normally recommended for removal of executable code.

8.2.2.15 Find and Replace

If the `replace` form of the `-find` option is used, any matching sequences will be replaced, or partially replaced, with new codes.

To have the `-find` option perform replacement, append `,replace=`*`spec`* to the option in the following manner.

```
-find=spec,replace=Code[mMask]
```

where:

- *`Code`* is a hexadecimal code sequence to replace the sequences that match the `-find` criteria.
- *`Mask`* is an optional bit mask to specify which bits within *`Code`* will replace the code sequence that has been matched. This can be useful if, for example, it is only necessary to modify 4 bits within a 16-bit instruction. The remaining 12 bits can masked and left unchanged.

For example:

```
-find=ff@7fe0-7fff,replace=55
```

This function should be used with extreme caution.

8.2.2.16 Format

The `-format=`*`type`*`[,`*`length`*`]` option can be used to specify a particular named format of INHX file to output and/or to adjust the maximum record length.

The *`type`*argument specifies a particular named INHX format to generate, as tabulated below. This option might be used to change the format of an input file, for example from an INHX16 to an INHX32 file, but the presence of record types 2, 3, 4, or 5 in the input will prevent conversion to an INHX8 file from any other format. Note that the record types present in a HEX file solely determine the file's format, for example, if only record types 0 and 1 are present in a HEX file, then that file is considered to have an INHX8M format; if type 4 or 5 records were also present, then it is considered to have INHX32 format; if type 2 or 3 and type 4 or 5 records were present, then the file does not conform to any named format and those records which are not permitted in the specified output format will be removed.

The *length*argument is optional and sets the maximum number of bytes per data record. A valid length is between 1 and 16 decimal, with 16 being the default.

The possible types that are supported by this option are listed below. Note that `INHX032` is not an actual named INHX format. Selection of this type generates an INHX32 file, but will also initialize the upper address information to zero. This is a requirement of some device programmers.

Table 8-4. HEX file Formats

Type	Valid record types	Comments
`INHX8M`	0, 1	16-bit wide address field.
`INHX16`	0, 1, 2, 3	20-bit wide address field.
`INHX32`	0,1, 4, 5	32-bit wide address field.
`INHX032`	0,1, 4, 5	INHX32 with initialization of upper address to zero.

8.2.2.17 Help

Using `-help` will list all Hexmate options. Entering another Hexmate option as a parameter of `-help` will show a detailed help message for the given option. For example:

```
-help=string
```

will show additional help for the `-string` Hexmate option.

8.2.2.18 Logfile

The `-logfile` option saves HEX file statistics to the named file. For example:

```
-logfile=output.hxl
```

will analyze the HEX file that Hexmate is generating and save a report to a file named `output.hxl`.

8.2.2.19 Mask

Use the `-mask=`*spec* option to logically AND a memory range with a particular bitmask. This is used to ensure that the unimplemented bits in program words (if any) are left blank. The usage of this option is as follows:

```
-mask=hexcode@start-end
```

where *hexcode* is a value that will be ANDed with data within the *start* to *end* address range. All values are assumed to be hexadecimal. Multibyte mask values can be entered in little endian byte order.

8.2.2.20 O: Specify Output File

When using the `-o`*file* option, the generated Intel HEX output will be created in the specified file. For example:

```
-oprogram.hex
```

will save the resultant output to `program.hex`. The output file can take the same name as one of its input files but, by doing so, it will replace the input file entirely.

If this option is used without a filename, no output is produced, which may be useful if you want to use Hexmate to only show the size of a HEX file, for example. If this option is not used at all, the content of the output HEX file is printed to the standard output stream.

8.2.2.21 Serial

The `-serial=`*specs* option will store a particular HEX value sequence at a fixed address. The usage of this option is:

```
-serial=Code[+/-Increment]@Address[+/-Interval][rRepetitions]
```

where:

- *Code* is a hexadecimal sequence to store. The first byte specified is stored at the lowest address.
- *Increment* is optional and allows the value of *Code* to change by this value with each repetition (if requested).
- *Address* is the location to store this code, or the first repetition thereof.
- *Interval* is optional and specifies the address shift per repetition of this code.
- *Repetitions* is optional and specifies the number of times to repeat this code.

All numerical arguments are assumed to be hexadecimal values, except for the *Repetitions* argument, which is decimal value by default.

For example:

```
-serial=000001@EFFE
```

will store HEX code 0x00001 to address 0xEFFE.

Another example:

```
-serial=0000+2@1000+10r5
```

will store 5 codes, beginning with value 0000 at address 0x1000. Subsequent codes will appear at address intervals of +0x10 and the code value will change in increments of +0x2.

8.2.2.22 Size

Using the `-size` option will report the number of bytes of data within the resultant HEX image to standard output. The size will also be recorded in the log file if one has been requested.

8.2.2.23 String

The `-string` option will embed an ASCII string at a fixed address. The usage of this option is:

```
-string@Address[tCode]=”Text”
```

where:

- *Address* is assumed to be a hexadecimal value representing the address at which the string will be stored.
- *Code* is optional and allows a byte sequence to trail each byte in the string. This can allow the bytes of the string to be encoded within an instruction.
- *Text* is the string to convert to ASCII and embed.

For example:

```
-string@1000=”My favorite string”
```

will store the ASCII data for the string, My favorite string (including the null character terminator), at address 0x1000.

And again:

```
-string@1000t34=”My favorite string”
```

will store the same string, trailing every byte in the string with the HEX code 0x34.

8.2.2.24 Strpack

The `-strpack=`*spec* option performs the same function as `-string`, but with two important differences.

Whereas `-string` stores the full byte corresponding to each character, `-strpack` stores only the lower seven bits from each character. Pairs of 7-bit characters are then concatenated and stored as a 14-bit word rather than in separate bytes. This is known as string packing. This is often useful for Mid-range PIC devices, where the program memory is addressed as 14-bit words. If you intend to use this option, you must ensure that the encoded characters are fully readable and correctly interpreted at runtime.

The second difference between these two options is that the `t` specifier usable with `-string` is not applicable with the `-strpack` option.

8.2.2.25 W: Specify warning level

The `-wlevel` option sets a warning level threshold. The `level` value can be a digit from -9 thru 9, for example `-w5`.

The warning level determines how pedantic Hexmate is about dubious requests or file content. Each warning has a designated warning level; the higher the warning level, the more important the warning message. If the warning message's warning level exceeds the threshold set with this option, the warning is printed. The default warning level threshold is 0 and will allow all normal warning messages.

8.2.3 Hash Value Calculations

A hash value is a small fixed-size value that is calculated from, and used to represent, all the values in an arbitrary-sized block of data. If that data block is copied, a hash recalculated from the new block can be compared to the original hash. Agreement between the two hashes provides a high level of certainty that the copy is valid. There are many hash algorithms. More complex algorithms provide a more robust verification, but can sometimes be too computationally demanding when used in an embedded environment, particularly for smaller devices.

Hexmate can be used to calculate the hash of a program image that is contained in a HEX file. This hash can be embedded into that same HEX file and burned into the target device along with the program image. At runtime, the target device can run a similar hash algorithm over the program image, now stored in its memory. If the stored and calculated hashes are the same, the embedded program can assume that it has a valid program image to execute.

Hexmate implements several hash algorithms, such as checksums and cyclic redundancy checks, which can be selected to calculate the hash value.

If you are using `xc8-cc` or the MPLAB X IDE to perform project builds, the compiler driver's `-mchecksum` option and suitable arguments will invoke Hexmate and pass it the appropriate options to calculate a hash. When executing Hexmate explicitly, the `-ck` option requests that a hash be calculated, as described in 8.2.2.11 Ck Hexmate Option.

Some consideration is required when a hash value is being calculated over memory that contains unused memory locations. If you are using `xc8-cc` or the MPLAB X IDE to perform project builds, requesting a hash value automatically requests that Hexmate fill unused memory locations to match unprogrammed device memory. When executing Hexmate explicitly, consider using the `-fill` option (see 8.2.2.12 Fill) to have these locations programmed with a known value.

Hexmate can produce a hash value from any Intel HEX file, regardless of which compiler produced the file and which device that file is intended to program. However, the architecture of the target device may restrict which memory locations can be read at runtime, thus requiring modification to the way in which Hexmate should perform hash calculations, so that the two hashes are calculated similarly and agree. In addition, some compilers might insert padding or phantom bytes into the HEX file that are not present in the device memory. These bytes might need to be ignored by Hexmate when it calculates a hash value and the following discussion indicates possible solutions.

Not all devices can read the entire width of their program memory. For example, Baseline and Mid-range PIC devices can only read the lower byte of each program memory location. The HEX file, however, will contain two bytes for each program memory word and both these bytes will normally be processed by Hexmate when calculating a hash value. If you are using `xc8-cc` or the MPLAB X IDE to perform project builds, use the `skip=2` suboption to the `-mchecksum` option to have Hexmate skip the MSB of each program word. When executing Hexmate explicitly, use the `s2` suboption to the `-ck` option to have the MSB of each 2-byte word skipped. Note, however, that this sort of verification process is not considering corruption in the MSB of each program word.

Some devices have hardware CRC modules which can calculate a CRC hash value. If desired, program memory data can be streamed to this module using the Scanner module to automate the calculation. As the Scanner module reads the MSB of each program memory word first, you need to have Hexmate also process HEX file bytes within an instruction word in the reverse order. If you are using `xc8-cc` or the MPLAB X IDE to perform project builds when using these modules, use the `revword=2` suboption to the `-mchecksum` option to have Hexmate process bytes in a 2-byte word in reverse order. When executing Hexmate explicitly, use the `r2` suboption to the `-ck` option to have Hexmate process bytes in a 2-byte word in reverse order.

Some consideration must also be given to how the Hexmate hash value encoded in the HEX file can be read at runtime.

Baseline and Mid-range PIC devices must store data in program memory using `retlw` instructions. Thus they need one instruction to store each byte of the hash value calculated by Hexmate. If you are using `xc8-cc` or the MPLAB X IDE to perform project builds when using these devices, use the `code=34` suboption to the `-mchecksum` option to have Hexmate store each byte of the hash value in a `retlw` instruction (the `retlw` instruction is encoded as `0x34`*nn*, where *nn* is the 8-bit data value to be loaded to WREG when executed.) When executing Hexmate explicitly, use the `t34` suboption to the `-ck` option to have Hexmate process store each bytes as part of a `retlw` instruction.

The dsPIC and PIC24 devices cannot read the full width of their program memory and data to be loaded to this memory is typically stored in 2-byte chunks per every 4 bytes of HEX file data.If you are using `xc8-cc` or the MPLAB X IDE to perform project builds when using these devices, use the `code=0000.2` suboption to the `-mchecksum` option to have Hexmate append the bytes sequence 0x00 0x00 to each two bytes of hash data.When executing Hexmate explicitly, use the `t0000.2` suboption to the `-ck` option to have Hexmate store two bytes of the hash value in the lower half of each 4-bytes of the HEX file, with the upper bytes set to zero.

8.2.3.1 Hash Algorithms

The following sections provide examples of the algorithms that Hexmate uses and that can be used to calculate the corresponding hash value at runtime. Note that these examples may require modification for the intended device and situation.

8.2.3.1.1 Addition Algorithms

Hexmate has several simple checksum algorithms that sum data values over a range in the program image. These algorithms correspond to the selector values 1, 2, 3 and 4 in the algorithm suboption and read the data in the program image as 1, 2, 3 or 4 byte quantities, respectively. This summation is added to an initial value (offset) that is supplied to the algorithm via the same option. The width to which the final checksum is truncated is also specified by this option and can be 1, 2, 3 or 4 bytes. Hexmate will automatically store the checksum in the HEX file at the address specified in the checksum option.

The function shown below can be customized to work with any combination of data size (`read_t`) and checksum width (`result_t`).

```
#include <stdint.h>
typedef uint8_t read_t;        // size of data values read and summed
typedef uint16_t result_t;     // size of checksum result

// add to offset, n additions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to sum
// n:    the number of sums to perform
// offset: the intial value to which the sum is added
result_t ck_add(const read_t *data, unsigned n, result_t offset)
{
    result_t chksum;
    chksum = offset;
    while(n--) {
        chksum += *data;
        data++;
    }
    return chksum;
}
```

The `read_t` and `result_t` type definitions should be adjusted to suit the data read/sum width and checksum result width, respectively. When using MPLAB XC8 and for a size of 1, use a `char` type; for a size of 4, use a `long` type, etc. If you never use an offset, that parameter can be removed and `chksum` assigned 0 before the loop.

Here is how this function might be used when, for example, a 2-byte-wide checksum is to be calculated from the addition of 1-byte-wide values over the address range 0x100 to 0x7fd, starting with an offset of 0x20. The checksum is to be stored at 0x7fe and 0x7ff in little endian format.

The following option is specified when building the project. In MPLAB X IDE, only enter the information to the right of the first = in the **Checksum** field in the **Additional options** Option category in the **XC8 Linker** category.

```
-mchecksum=100-7fd@7fe,offset=20,algorithm=1,width=-2
```

When executing Hexmate explicitly, use the option:

```
-ck=100-7fd@7fe+20g1w-2
```

Adapt the following MPLAB XC8 code snippet for PIC devices, which calls `ck_add()` and compares the runtime checksum with that stored by Hexmate at compile time.

```
extern const read_t ck_range[0x6fe/sizeof(read_t)] __at(0x100);
extern const result_t hexmate __at(0x7fe);
result_t result;

result = ck_add(ck_range, sizeof(ck_range)/sizeof(read_t), 0x20);
if(result != hexmate)
    ck_failure();  // take appropriate action
```

This code uses the placeholder array, `ck_range`, to represent the memory over which the checksum is calculated and the variable `hexmate` is mapped over the locations where Hexmate will have stored its checksum result. Being `extern` and absolute, neither of these objects consume additional device memory. Adjust the addresses and sizes of these objects to match the option you pass to Hexmate.

Hexmate can calculate a checksum over any address range; however, the test function, `ck_add()`, assumes that the start and end address of the range being summed are a multiple of the `read_t` width. This is a non-issue if the size of `read_t` is 1. It is recommended that your checksum specification adheres to this assumption, otherwise you will need to modify the test code to perform partial reads of the starting and/or ending data values. This will significantly increase the code complexity.

8.2.3.1.2 Subtraction Algorithms

Hexmate has several checksum algorithms that subtract data values over a range in the program image. These algorithms correspond to the selector values -1, -2, -3, and -4 in the algorithm suboption and read the data in the program image as 1-, 2-, 3- or 4-byte quantities, respectively. In other respects, these algorithms are identical to the addition algorithms. See 8.2.3.1.1 Addition Algorithms for further information regarding the subtraction algorithms.

The function shown below can be customized to work with any combination of data size (`read_t`) and checksum width (`result_t`).

```
#include <stdint.h>
typedef uint8_t read_t;        // size of data values read and subtracted
typedef uint16_t result_t;     // size of checksum result

// add to offset n subtractions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to subtract
// n:    the number of subtractions to perform
// offset: the intial value to which the subtraction is added
result_t ck_sub(const read_t *data, unsigned n, result_t offset)
{
    result_t chksum;
    chksum = offset;
    while(n--) {
        chksum -= *data;
        data++;
    }
    return chksum;
}
```

Here is how this function might be used when, for example, a 4-byte-wide checksum is to be calculated from the addition of 2-byte-wide values over the address range 0x0 to 0x7fd, starting with an offset of 0x0. The checksum is to be stored at 0x7fe and 0x7ff in little endian format.

The following option is specified when building the project using `xc8-cc`. In MPLAB X IDE, only enter the information to the right of the first = in the **Checksum** field in the **Additional options** Option category in the **XC8 Linker** category.

```
-mchecksum=0-7fd@7fe,offset=0,algorithm=-2,width=-4
```

When executing Hexmate explicitly, use the option:

```
-ck=0-7fd@7fe+0g-2w-4
```

Adapt the following MPLAB XC8 code snippet for PIC devices, which calls `ck_sub()` and compares the runtime checksum with that stored by Hexmate at compile time.

```
extern const read_t ck_range[0x7fe/sizeof(read_t)] __at(0x0);
extern const result_t hexmate __at(0x7fe);
result_t result;

result = ck_sub(ck_range, sizeof(ck_range)/sizeof(read_t), 0x0);
if(result != hexmate)
    ck_failure();  // take appropriate action
```

8.2.3.1.3 Fletcher Algorithms

Hexmate has several algorithms that implement Fletcher's checksum. These algorithms are more complex, providing a robustness approaching that of a cyclic redundancy check, but with less computational effort. There are two forms of this algorithm which correspond to the selector values 7 and 8 in the algorithm suboption and which implement a 1-byte calculation and 2-byte result, with a 2-byte calculation and 4-byte result, respectively. Hexmate will automatically store the checksum in the HEX file at the address specified in the checksum option.

The function shown below performs a 1-byte-wide addition and produces a 2-byte result.

```
#include <stdint.h>
typedef uint16_t result_t;     // size of fletcher result

result_t
fletcher8(const unsigned char * data, unsigned int n )
{
    result_t sum = 0xff, sumB = 0xff;
    unsigned char tlen;
    while (n) {
        tlen = n > 20 ? 20 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xff) + (sum >> 8);
        sumB = (sumB & 0xff) + (sumB >> 8);
    }
    sum = (sum & 0xff) + (sum >> 8);
    sumB = (sumB & 0xff) + (sumB >> 8);
    return sumB << 8 | sum;
}
```

Here is how this function might be used when, for example, a 2-byte-wide Fletcher hash is to be calculated over the address range 0x100 to 0x7fb, starting with an offset of 0x20. The checksum is to be stored at 0x7fc thru 0x7ff in little endian format.

The following option is specified when building the project. In MPLAB X IDE, only enter the information to the right of the first = in the **Checksum** field in the **Additional options** Option category in the **XC8 Linker** category. Note that the width cannot be controlled with the `width` suboption, but the sign of this suboption's argument is used to indicate the required endianism of the result.

```
-mchecksum=100-7fb@7fc,offset=20,algorithm=8,width=-4
```

When executing Hexmate explicitly, use the following option. Note that the width cannot be controlled with the `w` suboption, but the sign of this suboption's argument is used to indicate the required endianism of the result.

```
-ck=100-7bd@7fc+20g8w-4
```

This code can be called in a manner similar to that shown for the addition algorithms (see 8.2.3.1.1 Addition Algorithms).

The code for the 2-byte-addition Fletcher algorithm, producing a 4-byte result is shown below.

```
#include <stdint.h>
typedef uint32_t result_t;     // size of fletcher result

result_t
fletcher16(const unsigned int * data, unsigned n)
{
```

```
    result_t sum = 0xffff, sumB = 0xffff;
    unsigned tlen;
    while (n) {
        tlen = n > 359 ? 359 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xffff) + (sum >> 16);
        sumB = (sumB & 0xffff) + (sumB >> 16);
    }
    sum = (sum & 0xffff) + (sum >> 16);
    sumB = (sumB & 0xffff) + (sumB >> 16);
    return sumB << 16 | sum;
}
```

8.2.3.1.4 CRC Algorithms

Hexmate has several algorithms that implement the robust cyclic redundancy checks (CRC). There is a choice of two algorithms that correspond to the selector values 5 and -5 in the algorithm suboption and that implement a CRC calculation and reflected CRC calculation, respectively. The reflected algorithm works on the least significant bit of the data first.

The polynomial to be used and the initial value can be specified in the option. Hexmate will automatically store the CRC result in the HEX file at the address specified in the checksum option.

Some devices implement a CRC module in hardware that can be used to calculate a CRC at runtime. These modules can stream data read from program memory using a Scanner module. To ensure that the order of the bytes processed by Hexmate and the CRC/Scanner module are identical, you must specify a reverse word width of 2, which will read each 2-byte word in the HEX file in order, but process the bytes within those words in reverse order. When using the MPLAB XC8 compiler, use the suboption `revword=2` When running Hexmate explicitly, use the `r2` suboption to `-ck`.

The function shown below can be customized to work with any result width (`result_t`). It calculates a CRC hash value using the polynomial specified by the `POLYNOMIAL` macro.

```
#include <stdint.h>
typedef uint16_t result_t;     // size of CRC result
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(result_t))
#define MSb     ((result_t)1 << (WIDTH - 1))

result_t
crc(const unsigned char * data, unsigned n, result_t remainder) {
    unsigned pos;
    unsigned char bitp;
    for (pos = 0; pos != n; pos++) {
        remainder ^= ((result_t)data[pos] << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    return remainder;
}
```

The `result_t` type definition should be adjusted to suit the result width. When using MPLAB XC8 and for a size of 1, use a `char` type; for a size of 4, use a `long` type, etc.

Here is how this function might be used when, for example, a 2-byte-wide CRC hash value is to be calculated values over the address range 0x0 to 0xFF, starting with an initial value of 0xFFFF. The result is to be stored at 0x100 and 0x101 in little endian format.

The following option is specified when building the project. In MPLAB X IDE, only enter the information to the right of the first `=` in the **Checksum** field in the **Additional options** Option category in the **XC8 Linker** category.

```
-mchecksum=0-FF@100,offset=0xFFFF,algorithm=5,width=-2,polynomial=0x1021
```

When executing Hexmate explicitly, use the option:

```
-ck=0-ff@100+ffffg5w-2p1021
```

Adapt the following MPLAB XC8 code snippet for PIC devices, which calls `crc()` and compares the runtime hash result with that stored by Hexmate at compile time.

```
extern const unsigned char ck_range[0x100] __at(0x0);
extern const result_t hexmate __at(0x100);
result_t   result;

result = crc(ck_range, sizeof(ck_range), 0xFFFF);
if(result != hexmate){
    // something's not right, take appropriate action
    ck_failure();
}
// data verifies okay, continue with the program
```

The reflected CRC result can be calculated by reflecting the input data and final result, or by reflecting the polynomial. The functions shown below can be customized to work with any result width (`result_t`). The `crc_reflected_IO()` function calculates a reflected CRC hash value by reflecting the data stream bit positions. Alternatively, the `crc_reflected_poly()` function does not adjust the data stream but reflects instead the polynomial, which in both functions is specified by the `POLYNOMIAL` macro. Both functions use the `reflect()` function to perform bit reflection.

```
#include <stdint.h>
typedef uint16_t result_t;     // size of CRC result
typedef unsigned char read_t;
typedef unsigned int reflectWidth;
// This is the polynomial used by the CRC-16 algorithm we are using.
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(result_t))
#define MSb     ((result_t)1 << (WIDTH - 1))
#define LSb     (1)
#define REFLECT_DATA(X)       ((read_t) reflect((X), 8))
#define REFLECT_REMAINDER(X)  (reflect((X), WIDTH))

reflectWidth
reflect(reflectWidth data, unsigned char nBits)
{
    reflectWidth reflection = 0;
    reflectWidth reflectMask = (reflectWidth)1 << nBits - 1;
    unsigned char bitp;
    for (bitp = 0; bitp != nBits; bitp++) {
        if (data & 0x01) {
            reflection |= reflectMask;
        }
        data >>= 1;
        reflectMask >>= 1;
    }
    return reflection;
}

result_t
crc_reflected_IO(const unsigned char * data, unsigned n, result_t remainder) {
    unsigned pos;
    unsigned char reflected;
    unsigned char bitp;
    for (pos = 0; pos != n; pos++) {
        reflected = REFLECT_DATA(data[pos]);
        remainder ^= ((result_t)reflected << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    remainder = REFLECT_REMAINDER(remainder);
    return remainder;
}
```

```
result_t
crc_reflected_poly(const unsigned char * data, unsigned n, result_t remainder) {
    unsigned pos;
    unsigned char bitp;
    result_t rpoly;
    rpoly = reflect(POLYNOMIAL, WIDTH);
    for (pos = 0; pos != n; pos++) {
        remainder ^= data[pos];
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & LSb) {
                remainder = (remainder >> 1) ^ rpoly;
            } else {
                remainder >>= 1;
            }
        }
    }
    return remainder;
}
```

Here is how this function might be used when, for example, a 2-byte-wide reflected CRC result is to be calculated over the address range 0x0 to 0xFF, starting with an initial value of 0xFFFF. The result is to be stored at 0x100 and 0x101 in little endian format.

The following option is specified when building the project. In MPLAB X IDE, only enter the information to the right of the first = in the **Checksum** field in the **Additional options** Option category in the **XC8 Linker** category. Note the algorithm selected is negative 5 in this case.

```
-mchecksum=0-FF@100,offset=0xFFFF,algorithm=-5,width=-2,polynomial=0x1021
```

When executing Hexmate explicitly, instead use the following option, noting that the algorithm selected is negative 5 in this case.

```
-ck=0-ff@100+ffffg-5w-2p1021
```

In your project, call either the `crc_reflected_IO()` or `crc_reflected_poly()` functions, as shown previously.

8.2.3.1.5 SHA Algorithms

Hexmate implements a secure hash algorithm (SHA). The selector value 10 selects the SHA256 algorithm, a 256-bit variant of the SHA-2 algorithm.

The code to implement a SHA256 is more complex than other algorithms supported by Hexmate, and as its name suggests, the result of such a hash is 256 bits (32 bytes) wide. Public-domain implementations of this algorithm are available for download from third-party websites, such as github.com/B-Con/crypto-algorithms.

Here is how Hexmate might be used when, for example, a SHA256 hash value is to be calculated values over the address range 0x0 to 0x1FF. The result is to be stored at a starting address of 0x1000 in little endian format.

```
-ck=0-1ff@1000g10w-256
```

9. 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.

9.1 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 example code might be encoded differently on different target devices and might operate differently at runtime.

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

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

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

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

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

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

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

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

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

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

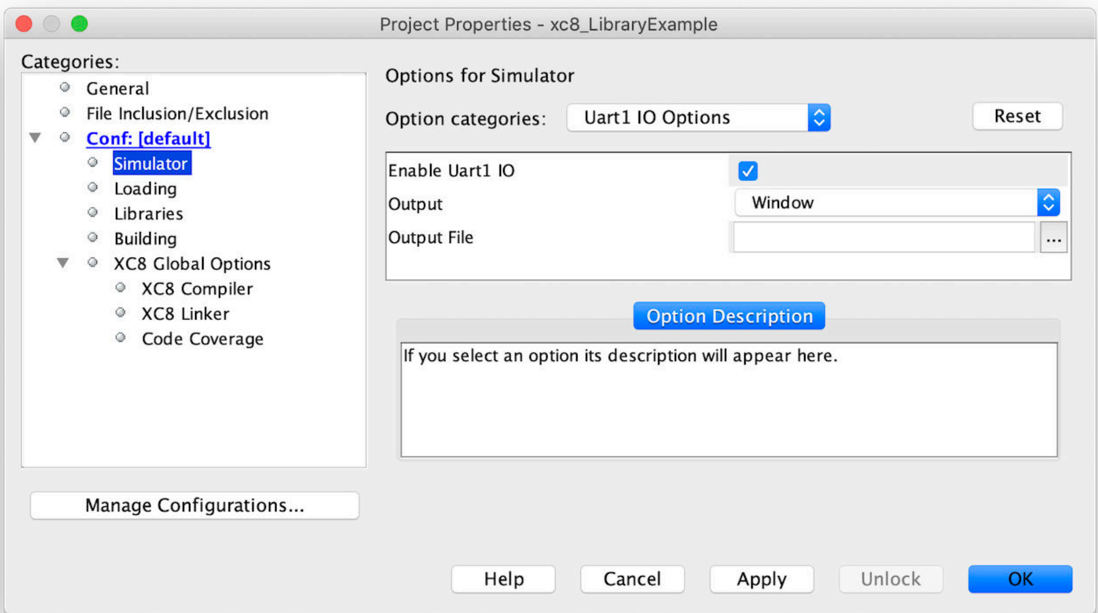
Compiler-specific implementations of the above are discussed in more detail in the following sections.

9.1.1 Example code for 8-bit PIC MCUs

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

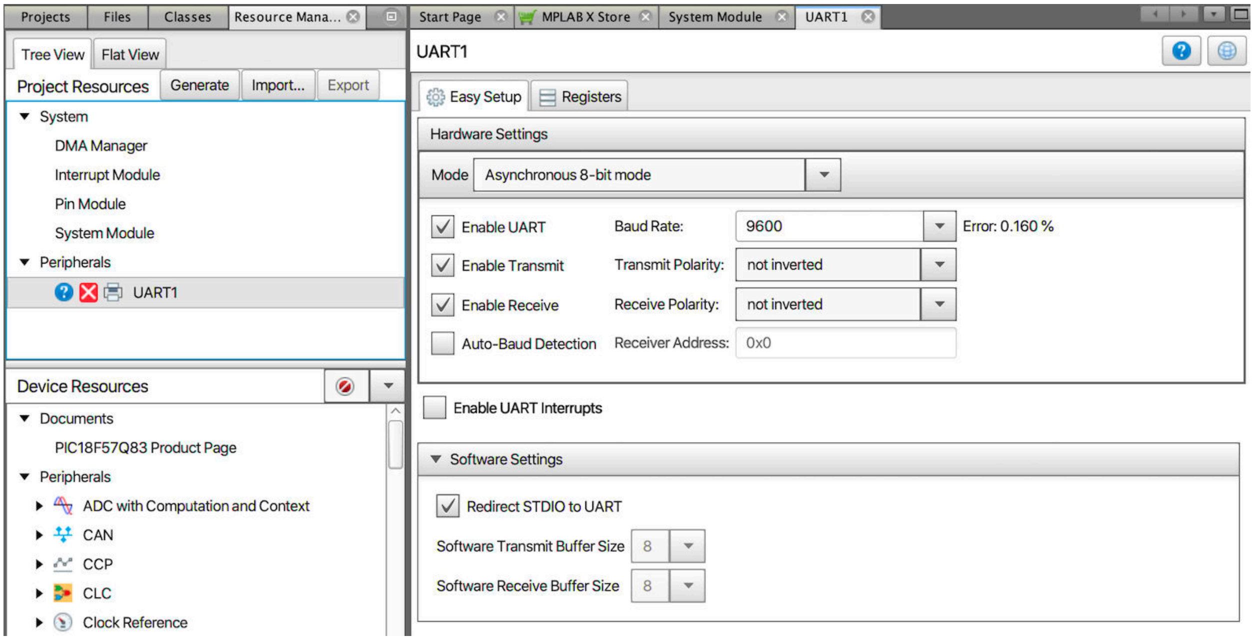
When available, this feature is enabled using the **Enable Uartx IO** checkbox in the **Project properties > Simulator > Uartx IO Options** dialog (shown below). You might have a choice of UARTs. Choose the UART that your code will write to. Output can be displayed in a window in the IDE or sent to a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

Figure 9-1. Enabling the UART IO feature in the MPLAB X IDE



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

Figure 9-2. Initializing the UART using MCC



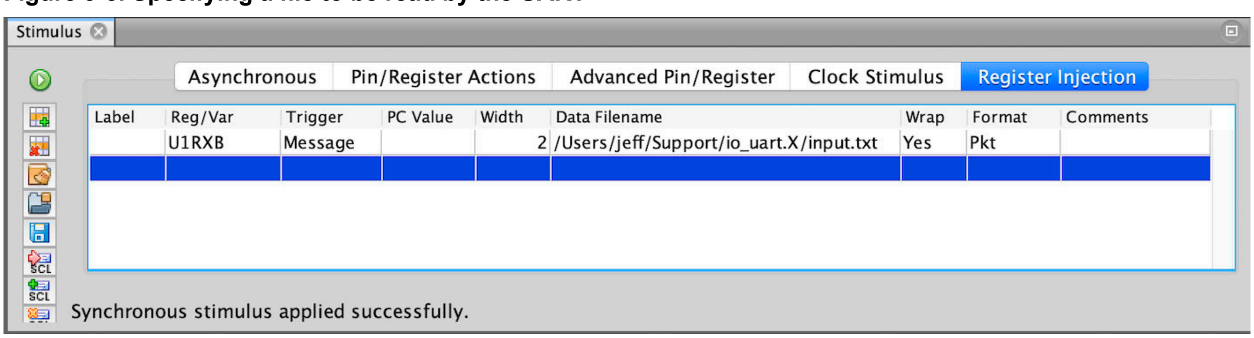
The same UART IO feature allows you to read input from the `stdin` stream. Once properly configured, `scanf()` and other functions reading from `stdin` can then take input from a file when the program is run in the simulator.

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

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

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

Figure 9-3. Specifying a file to be read by the UART



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

9.2 <xc.h> Device-specific Functions

The header file, `<xc.h>`, consists of types, macros and functions specific to your target device. It includes device-specific header files that also provide access to special function registers.

9.2.1 CLRWDT Macro

A macro that clears the watchdog timer.

Include

`<xc.h>`

Prototype

`void CLRWDT(void);`

Remarks

This macro executes a `clrwdt` instruction.

Example

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
void main(void) {
  unsigned char c;

  SYSTEM_Initialize();
  WDTCON0bits.SWDTEN = 1;

  while (1) {
    c = PORTA;
    LATB = c;
    CLRWDT();
  }
}
```

9.2.2 di Macro

A macro that disables interrupts.

Include

`<xc.h>`

Prototype

`void di(void);`

Remarks

This macro clears the GIE bit in the INTCON register. The MCC-generated `INTERRUPT_GlobalInterruptDisable()` function performs the same task as the `di()` macro .

Example

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

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

unsigned int count;

void __interrupt() myIsr (void) {
    if(INTCONbits.TMR0IE == 1 && INTCONbits.TMR0IF == 1) {
        INTCONbits.TMR0IF = 0;
        TMR0 = timer0ReloadVal;
        count++;
    }
```

```
}

unsigned int getTicks(void) {
  unsigned int val;

  di();         /* disable interrupts */
  val = count;  /* protected non-atomic operation */
  ei();         /* re-enable interrupts */
  return val;
}

int main(void) {
  SYSTEM_Initialize();                 /* MCC generated code */
  INTERRUPT_GlobalInterruptEnable();   /* MCC alternative to ei() */
  INTERRUPT_PeripheralInterruptEnable();
  while(getTicks() < 0x400)            /* wait for 0x400 overflows */
    continue;
  INTERRUPT_GlobalInterruptDisable();  /* MCC alternative to di() */
  printf("done");
  while(1)
    continue;
}
```

9.2.3 ei Macro

A macro that enables interrupts.

Include

`<xc.h>`

Prototype

`void ei(void);`

Remarks

This macro sets the GIE bit in the INTCON register. The MCC-generated `INTERRUPT_GlobalInterruptEnable()` function performs the same task as the `ei()` macro .

Example

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

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

unsigned int count;

void __interrupt() myIsr (void) {
    if(INTCONbits.TMR0IE == 1 && INTCONbits.TMR0IF == 1) {
        INTCONbits.TMR0IF = 0;
        TMR0 = timer0ReloadVal;
        count++;
    }
}

unsigned int getTicks(void) {
  unsigned int val;

  di();         /* disable interrupts */
  val = count;  /* protected non-atomic operation */
  ei();         /* re-enable interrupts */
  return val;
}

int main(void) {
  SYSTEM_Initialize();                 /* MCC generated code */
  INTERRUPT_GlobalInterruptEnable();   /* MCC alternative to ei() */
  INTERRUPT_PeripheralInterruptEnable();
  while(getTicks() < 0x400)            /* wait for 0x400 overflows */
    continue;
  INTERRUPT_GlobalInterruptDisable();  /* MCC alternative to di() */
```

```
  printf("done");
  while(1)
    continue;
}
```

9.2.4 eeprom_read Function

A function that reads data from the EEPROM memory space.

Include

`<xc.h>`

Prototype

`unsigned char eeprom_read(unsigned char address);`

Argument

address	The EEPROM address to read

Remarks

This function is available for all Mid-range devices that implement EEPROM. For PIC18 devices, calls to this routine will instead attempt to call the equivalent functions in the legacy PIC18 peripheral library, which you must download and install separately. It is recommended that for PIC18 devices you use MPLAB MCC to generate EEPROM access code, if possible.

This function tests and waits for any concurrent writes to EEPROM to conclude before performing the required operation.

Example

Several routines which can read EEPROM have been shown in this example, alone with a macro that can store data into the device.

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

__EEPROM_DATA(0x00, 0x20, 0x40, 0x60, 0x80, 0x00, 0x00, 0x00);

void main(void) {
  unsigned char byte;

  SYSTEM_Initialize();

  byte = DATAEE_ReadByte(2);  /* MCC-generated function */
  printf("Byte at address 2 in EEPROM is %x\n", byte);
  byte = eeprom_read(3);
  printf("Byte at address 3 in EEPROM is %x\n", byte);
  byte = EEPROM_READ(4);
  printf("Byte at address 4 in EEPROM is %x\n", byte);

  while (1) {
  }
}
```

Example Output

```
Byte at address 2 in EEPROM is 40
Byte at address 3 in EEPROM is 60
Byte at address 4 in EEPROM is 80
```

9.2.5 eeprom_write Function

A function that writes data to the EEPROM memory space.

Include

`<xc.h>`

Prototype

`void eeprom_write(unsigned char address, unsigned char value);`

Arguments

address	The EEPROM address to program
value	The value to program

Remarks

This function is available for all Mid-range devices that implement EEPROM. For PIC18 devices, calls to this routine will instead attempt to call the equivalent functions in the legacy PIC18 peripheral library, which you must download and install separately. It is recommended that for PIC18 devices you use MPLAB MCC to generate EEPROM access code, if possible.

This function tests and waits for any concurrent writes to EEPROM to conclude before performing the required operation. The function will initiate the write of `value` to EEPROM at `address` and this process will still be taking place when the function returns. The new data written to EEPROM will become valid at a later time. See your device data sheet for exact information about EEPROM on your target device.

Example

```
#include <xc.h>

int main(void)
{
  unsigned char data = 0x55;
  unsigned char addr = 0x20;

  eeprom_write(addr, data);
}
```

9.2.6 EEPROM_READ Macro

A macro that reads data from the EEPROM memory space.

Include

`<xc.h>`

Prototype

`unsigned char EEPROM_READ(scalar address);`

Argument

address	The EEPROM address to read

Remarks

This macro is available for all Mid-range devices that implement EEPROM. It is recommended that for PIC18 devices you use MPLAB MCC to generate EEPROM access code, if possible.

Unlike the `eeprom_read` function, this macro does not wait for any concurrent writes to EEPROM to conclude before performing the required operation. It read the EEPROM and returns the value at the nominated `address`.

Example

Several routines which can read EEPROM have been shown in this example, alone with a macro that can store data into the device.

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

__EEPROM_DATA(0x00, 0x20, 0x40, 0x60, 0x80, 0x00, 0x00, 0x00);

void main(void) {
  unsigned char byte;

  SYSTEM_Initialize();
```

```
  byte = DATAEE_ReadByte(2);  /* MCC-generated function */
  printf("Byte at address 2 in EEPROM is %x\n", byte);
  byte = eeprom_read(3);
  printf("Byte at address 3 in EEPROM is %x\n", byte);
  byte = EEPROM_READ(4);
  printf("Byte at address 4 in EEPROM is %x\n", byte);

  while (1) {
  }
}
```

Example Output

```
Byte at address 2 in EEPROM is 40
Byte at address 3 in EEPROM is 60
Byte at address 4 in EEPROM is 80
```

9.2.7 EEPROM_WRITE Macro

A macro that writes data to the EEPROM memory space.

Include

`<xc.h>`

Prototype

`void EEPROM_WRITE(`*`scalar`*` address, `*`scalar`*` value);`

Arguments

`address`	The EEPROM address to program
`value`	The value to program

Remarks

This macro is available for all Mid-range devices that implement EEPROM. It is recommended that for PIC18 devices you use MPLAB MCC to generate EEPROM access code, if possible.

This function tests and waits for any concurrent writes to EEPROM to conclude before performing the required operation. It writes `value` to `address` in the EEPROM.

Example

```
#include <xc.h>

int main (void)
{
  unsigned char data = 0x55;
  unsigned char addr = 0x20;

  EEPROM_WRITE(addr, data);
}
```

9.2.8 __EEPROM_DATA Macro

A macro that stores values in the device's EEPROM registers at the time of programming.

Include

`<xc.h>`

Prototype

`__EEPROM_DATA(value, value, value, value, value, value, value, value);`

Arguments

`value`	One of eight values that are to be programmed to EEPROM

Remarks

This macro is used to store initial values in the device's EEPROM registers at the time of programming.

The macro must be given blocks of 8 bytes to write each time it is called and can be called repeatedly to store multiple blocks.

The macro will begin writing to EEPROM address zero and auto-increment the address by 8 each time it is used.

Example

Several routines which can read EEPROM have been shown in this example, alone with a macro that can store data into the device.

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/
#include <stdio.h>

__EEPROM_DATA(0x00, 0x20, 0x40, 0x60, 0x80, 0x00, 0x00, 0x00);

void main(void) {
  unsigned char byte;

  SYSTEM_Initialize();

  byte = DATAEE_ReadByte(2);  /* MCC-generated function */
  printf("Byte at address 2 in EEPROM is %x\n", byte);
  byte = eeprom_read(3);
  printf("Byte at address 3 in EEPROM is %x\n", byte);
  byte = EEPROM_READ(4);
  printf("Byte at address 4 in EEPROM is %x\n", byte);

  while (1) {
  }
}
```

Example Output

```
Byte at address 2 in EEPROM is 40
Byte at address 3 in EEPROM is 60
Byte at address 4 in EEPROM is 80
```

9.2.9 _delay Builtin

A builtin function that delays execution.

Include

`<xc.h>`

Prototype

`void _delay(unsigned long cycles);`

Argument

`cycles`	The number of cycles to delay

Remarks

This is an inbuilt function that is expanded by the code generator. When called, this routine expands to an in-line assembly delay sequence. The sequence will consist of code that delays for the number of instruction cycles that is specified as the argument. The argument must be a constant expression that does not contain variables or function calls and that can be fully evaluated at compile time.

The `_delay()` builtin function can use loops and the `nop` instruction to implement the delay.

An error will result if the requested delay is not a constant expression or is greater than 50,463,240 instructions. For even larger delays, call this function multiple times.

Example

All of the inbuilt delay routines are shown in the following example.

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC. */

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected. */

unsigned char count;

void main(void) {
  SYSTEM_Initialize();

  while (1) {
    LATA = 0xFF;
    _delay(100);          /* wait 100 cycles*/
    LATA = 0xAA;
    _delaywdt(100);       /* wait 100 cycles, clearing the watchdog */
    LATA = 0x55;
    _delay3(10);          /* wait 30 cycles */
    LATA = 0xFF;
    __delay_us(800);      /* wait 800 micro seconds */
    LATA = 0x00;
    __delaywdt_us(800);   /* wait 800 micro seconds, clearing the watchdog */
    LATA = count;
    __delay_ms(500);      /* wait half a second */
    LATA = ~count;
    __delaywdt_ms(500);   /* wait half a second, clearing the watchdog */
    count++;
  }
}
```

9.2.10 _delaywdt Builtin

A builtin function that delays execution.

Include

`<xc.h>`

Prototype

`void _delaywdt(unsigned long cycles);`

Argument

`cycles`	The number of cycles to delay

Remarks

This is an inbuilt function that is expanded by the code generator. When called, this routine expands to an in-line assembly delay sequence. The sequence will consist of code that delays for the number of instruction cycles that is specified as the argument. The argument must be a constant expression that does not contain variables or function calls and that can be fully evaluated at compile time.

The `_delay()` builtin function can use loops and the `clrwdt` instruction to implement the delay.

An error will result if the requested delay is not a constant expression or is greater than 50,463,240 instructions. For even larger delays, call this function multiple times.

Example

All of the inbuilt delay routines are shown in the following example.

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC. */
```

```
/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected. */

unsigned char count;

void main(void) {
  SYSTEM_Initialize();

  while (1) {
    LATA = 0xFF;
    _delay(100);          /* wait 100 cycles*/
    LATA = 0xAA;
    _delaywdt(100);       /* wait 100 cycles, clearing the watchdog */
    LATA = 0x55;
    _delay3(10);          /* wait 30 cycles */
    LATA = 0xFF;
    __delay_us(800);      /* wait 800 micro seconds */
    LATA = 0x00;
    __delaywdt_us(800);  /* wait 800 micro seconds, clearing the watchdog */
    LATA = count;
    __delay_ms(500);     /* wait half a second */
    LATA = ~count;
    __delaywdt_ms(500);  /* wait half a second, clearing the watchdog */
    count++;
  }
}
```

9.2.11 _delay3 Builtin

A builtin function that delays execution.

Include

`<xc.h>`

Prototype

`void _delay3(unsigned char loops);`

Argument

loops	The number of loops, each of 3 cycles, to delay

Remarks

This is an inbuilt function that is expanded by the code generator. When called, this routine expands to an in-line assembly delay sequence. The sequence will consist of code that delays for 3 times the number of instruction cycles that is specified as the argument. The argument must be a constant expression less than 257 that does not contain variables or function calls and that can be fully evaluated at compile time.

The `_delay3()` builtin function will use a loop to implement the delay.

Example

All of the inbuilt delay routines are shown in the following example.

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC. */

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected. */

unsigned char count;

void main(void) {
  SYSTEM_Initialize();

  while (1) {
```

```
    LATA = 0xFF;
    _delay(100);         /* wait 100 cycles*/
    LATA = 0xAA;
    _delaywdt(100);      /* wait 100 cycles, clearing the watchdog */
    LATA = 0x55;
    _delay3(10);         /* wait 30 cycles */
    LATA = 0xFF;
    __delay_us(800);     /* wait 800 micro seconds */
    LATA = 0x00;
    __delaywdt_us(800);  /* wait 800 micro seconds, clearing the watchdog */
    LATA = count;
    __delay_ms(500);     /* wait half a second */
    LATA = ~count;
    __delaywdt_ms(500);  /* wait half a second, clearing the watchdog */
    count++;
  }
}
```

9.2.12 __builtin_software_breakpoint Builtin

A builtin function that triggers a software breakpoint.

- Usable with all devices but has no effect when used with 8-bit Baseline PIC targets.

Include

`<xc.h>`

Prototype

`void __builtin_software_breakpoint(void);`

Remarks

This is an inbuilt function that is expanded by the code generator. When called, this routine unconditionally triggers a software breakpoint when the code is executed using a debugger.

The software breakpoint code is only generated for mid-range and PIC18 devices. Baseline devices do not support software breakpoints in this way, and the builtin will be ignored if used with these devices.

Example

The following example shows different types of breakpoints added to an MCC-generated function.

```
static i2c1_fsm_states_t I2C1_DO_BUS_COLLISION(void)
{
  // Clear bus collision status flag
  I2C1_MasterClearIrq();

  I2C1_Status.error = I2C1_FAIL;
  switch (I2C1_Status.callbackTable[I2C1_WRITE_COLLISION]
(I2C1_Status.callbackPayload[I2C1_WRITE_COLLISION])) {
  case I2C1_RESTART_READ:
    return I2C1_DO_SEND_RESTART_READ();
  case I2C1_RESTART_WRITE:
    __debug_break();                /* break when debugging only */
    return I2C1_DO_SEND_RESTART_WRITE();
  default:
    __builtin_software_breakpoint(); /* unconditional break */
    return I2C1_DO_RESET();
  }
}
```

9.2.13 __debug_break Builtin

A builtin function that triggers a software breakpoint for debug builds.

Include

`<xc.h>`

Prototype

`void __debug_break(void);`

Remarks

This is an inbuilt function that is expanded by the code generator for debug builds, but is ignore for production builds. When called, this routine unconditionally triggers a software breakpoint when the code is executed using a debugger.

The software breakpoint code is only generated for mid-range and PIC18 devices. Baseline devices do not support software breakpoints in this way, and the builtin will be ignored if used with these devices.

Example

The following example shows different types of breakpoints added to an MCC-generated function.

```
static i2c1_fsm_states_t I2C1_DO_BUS_COLLISION(void)
{
  // Clear bus collision status flag
  I2C1_MasterClearIrq();

  I2C1_Status.error = I2C1_FAIL;
  switch (I2C1_Status.callbackTable[I2C1_WRITE_COLLISION]
(I2C1_Status.callbackPayload[I2C1_WRITE_COLLISION])) {
  case I2C1_RESTART_READ:
    return I2C1_DO_SEND_RESTART_READ();
  case I2C1_RESTART_WRITE:
    __debug_break();                 /* break when debugging only */
    return I2C1_DO_SEND_RESTART_WRITE();
  default:
    __builtin_software_breakpoint(); /* unconditional break */
    return I2C1_DO_RESET();
  }
}
```

9.2.14 __delay_ms Builtin

A builtin function that delays execution for the specified time.

Include

`<xc.h>`

Prototype

`void __delay_ms(unsigned long time);`

Argument

`time`	The number of milli seconds to delay

Remarks

This is an inbuilt function that is expanded by the code generator. When called, this routine expands to an in-line assembly delay sequence. The sequence will consist of code that delays for the number of milli seconds specified as the argument. The argument must be a constant expression that does not contain variables or function calls and that can be fully evaluated at compile time.

This macro require the prior definition of the preprocessor macro `_XTAL_FREQ`, which indicates the system frequency. This macro should equate to the oscillator frequency (in hertz) used by the system. Note that this macro only controls the behavior of these delays and does not affect the device execution speed.

The `__delay_ms()` builtin function can use loops and the `nop` instruction to implement the delay.

An error will result if the requested delay is not a constant expression or is too large. For larger delays, call this function multiple times.

Example

All of the inbuilt delay routines are shown in the following example.

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC. */
```

```
/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected. */

unsigned char count;

void main(void) {
  SYSTEM_Initialize();

  while (1) {
    LATA = 0xFF;
    _delay(100);         /* wait 100 cycles*/
    LATA = 0xAA;
    _delaywdt(100);      /* wait 100 cycles, clearing the watchdog */
    LATA = 0x55;
    _delay3(10);         /* wait 30 cycles */
    LATA = 0xFF;
    __delay_us(800);     /* wait 800 micro seconds */
    LATA = 0x00;
    __delaywdt_us(800);  /* wait 800 micro seconds, clearing the watchdog */
    LATA = count;
    __delay_ms(500);     /* wait half a second */
    LATA = ~count;
    __delaywdt_ms(500);  /* wait half a second, clearing the watchdog */
    count++;
  }
}
```

9.2.15 __delaywdt_ms Builtin

A builtin function that delays execution for the specified time.

Include

`<xc.h>`

Prototype

`void __delaywdt_ms(unsigned long time);`

Argument

time	The number of milli seconds to delay

Remarks

This is an inbuilt function that is expanded by the code generator. When called, this routine expands to an in-line assembly delay sequence. The sequence will consist of code that delays for the number of milli seconds specified as the argument. The argument must be a constant expression that does not contain variables or function calls and that can be fully evaluated at compile time.

This macro require the prior definition of the preprocessor macro `_XTAL_FREQ`, which indicates the system frequency. This macro should equate to the oscillator frequency (in hertz) used by the system. Note that this macro only controls the behavior of these delays and does not affect the device execution speed.

The `__delay_ms()` builtin function can use loops and the `clrwdt` instruction to implement the delay.

An error will result if the requested delay is not a constant expression or is too large. For larger delays, call this function multiple times.

Example

All of the inbuilt delay routines are shown in the following example.

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC. */

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected. */
```

```
unsigned char count;

void main(void) {
  SYSTEM_Initialize();

  while (1) {
    LATA = 0xFF;
    _delay(100);          /* wait 100 cycles*/
    LATA = 0xAA;
    _delaywdt(100);       /* wait 100 cycles, clearing the watchdog */
    LATA = 0x55;
    _delay3(10);          /* wait 30 cycles */
    LATA = 0xFF;
    __delay_us(800);      /* wait 800 micro seconds */
    LATA = 0x00;
    __delaywdt_us(800);   /* wait 800 micro seconds, clearing the watchdog */
    LATA = count;
    __delay_ms(500);      /* wait half a second */
    LATA = ~count;
    __delaywdt_ms(500);   /* wait half a second, clearing the watchdog */
    count++;
  }
}
```

9.2.16 __delay_us Builtin

A builtin function that delays execution for the specified time.

Include

`<xc.h>`

Prototype

`void __delay_us(unsigned long time);`

Argument

`time`	The number of micro seconds to delay

Remarks

This is an inbuilt function that is expanded by the code generator. When called, this routine expands to an in-line assembly delay sequence. The sequence will consist of code that delays for the number of micro seconds specified as the argument. The argument must be a constant expression that does not contain variables or function calls and that can be fully evaluated at compile time.

This macro require the prior definition of the preprocessor macro `_XTAL_FREQ`, which indicates the system frequency. This macro should equate to the oscillator frequency (in hertz) used by the system. Note that this macro only controls the behavior of these delays and does not affect the device execution speed.

The `__delay_us()` builtin function can use loops and the `nop` instruction to implement the delay.

An error will result if the requested delay is not a constant expression or is too large. For larger delays, call this function multiple times.

Example

All of the inbuilt delay routines are shown in the following example.

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC. */

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected. */

unsigned char count;
```

```
void main(void) {
  SYSTEM_Initialize();

  while (1) {
    LATA = 0xFF;
    _delay(100);          /* wait 100 cycles*/
    LATA = 0xAA;
    _delaywdt(100);       /* wait 100 cycles, clearing the watchdog */
    LATA = 0x55;
    _delay3(10);          /* wait 30 cycles */
    LATA = 0xFF;
    __delay_us(800);      /* wait 800 micro seconds */
    LATA = 0x00;
    __delaywdt_us(800);   /* wait 800 micro seconds, clearing the watchdog */
    LATA = count;
    __delay_ms(500);      /* wait half a second */
    LATA = ~count;
    __delaywdt_ms(500);   /* wait half a second, clearing the watchdog */
    count++;
  }
}
```

9.2.17 __delaywdt_us Builtin

A builtin function that delays execution for the specified time.

Include

`<xc.h>`

Prototype

`void __delaywdt_us(unsigned long time);`

Argument

time	The number of micro seconds to delay

Remarks

This is an inbuilt function that is expanded by the code generator. When called, this routine expands to an in-line assembly delay sequence. The sequence will consist of code that delays for the number of micro seconds specified as the argument. The argument must be a constant expression that does not contain variables or function calls and that can be fully evaluated at compile time.

This macro require the prior definition of the preprocessor macro `_XTAL_FREQ`, which indicates the system frequency. This macro should equate to the oscillator frequency (in hertz) used by the system. Note that this macro only controls the behavior of these delays and does not affect the device execution speed.

The `__delay_us()` builtin function can use loops and the `clrwdt` instruction to implement the delay.

An error will result if the requested delay is not a constant expression or is too large. For larger delays, call this function multiple times.

Example

All of the inbuilt delay routines are shown in the following example.

```
/* Using MPLAB Code Configurator */

#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this header explicitly if not using
MCC. */

/* _XTAL_FREQ is defined by MCC headers; when not using MCC, place a definition similar to:
  #define _XTAL_FREQ 4000000
... in your code to ensure that the 'us' and 'ms' forms of delay routines work as expected. */

unsigned char count;

void main(void) {
  SYSTEM_Initialize();
```

```
  while (1) {
    LATA = 0xFF;
    _delay(100);          /* wait 100 cycles*/
    LATA = 0xAA;
    _delaywdt(100);       /* wait 100 cycles, clearing the watchdog */
    LATA = 0x55;
    _delay3(10);          /* wait 30 cycles */
    LATA = 0xFF;
    __delay_us(800);      /* wait 800 micro seconds */
    LATA = 0x00;
    __delaywdt_us(800);   /* wait 800 micro seconds, clearing the watchdog */
    LATA = count;
    __delay_ms(500);      /* wait half a second */
    LATA = ~count;
    __delaywdt_ms(500);   /* wait half a second, clearing the watchdog */
    count++;
  }
}
```

9.2.18 __fpnormalize Function

A function that normalizes a floating-point value.

Include

`<xc.h>`

Prototype

`__fpnormalize(double value);`

`value`	The floating-point value to normalize

Remarks

This function can be used to ensure that an arbitrary 32-bit floating-point value (which is not the result of a calculation performed by the compiler) conforms to the "relaxed" floating-point rules (as described in Section 4.4.4 “Floating-Point Data Types”).

This function returns the value passed to it, but ensures that any subnormal argument is flushed to zero, and converts any negative zero argument to a positive zero result.

Example

```
#include <xc.h>

int main(void)
{
  double input_fp;
  // read in a floating-point value from an external source
  input_fp = getFP();
  // ensure it is formatted using the relaxed rules
  input_fp = __fpnormalize(input_fp);
}
```

9.2.19 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 "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/

void main(void) {
  SYSTEM_Initialize();

  INTERRUPT_GlobalInterruptEnable();
  WeatherStation_initialize();

  while (1) {
    if (tmr1_tick == 1) {
      WeatherStation_Print();
      tmr1_tick = 0;
    }
    SLEEP(); // wait for interrupt
    NOP();   // ensure instruction pre-fetched while sleeping is safe to execute on awakening
  }
}
```

9.2.20 __osccal_val Inbuilt

A builtin function that returns the internal oscillator calibration constant.

Include

`<xc.h>`

Prototype

`unsigned char __osccal_val(void);`

Remarks

This is a builtin function that is expanded to a label associated with address of the `retlw` instruction that encapsulates the oscillator configuration value. Calls to the function will return the device's oscillator configuration value, which can then be used in any expression, if required.

This function is only available for those devices that are shipped with such a value stored in program memory. It is automatically called by the runtime start-up code when required (unless you have explicitly disabled this option, see Section 3.7.1.14 "osccal") and you do not need to explicitly call it to calibrate the internal oscillator.

Example

```
#include <xc.h>

int main(void)
{
  OSCCAL = __osccal_val();
}
```

9.2.21 READTIMERx Macros

A macro that reads the timer peripheral.

Include

`<xc.h>`

Prototype

`unsigned short READTIMERx(void);`

Return Value

The value of the relevant TMR register.

Remarks

The `READTIMERx()` macro is available for PIC18 projects and returns the value held by the TMR*x* register, where *x* is one of the digits 0, 1 or 3.

Example

```
#include <xc.h>

int main(void)
{
  while(READTIMER0() != 0xFF)
    continue;
}
```

9.2.22 RESET Macro

A macro that resets the device.

Include

`<xc.h>`

Prototype

`void RESET(void);`

Remarks

Where possible, this macro executes a `reset` instruction. For devices that do not implement this instruction, this macro puts down some form of jump instruction whose destination is itself. For a Reset to then occur, ensure the watchdog timer is enabled and wait for that to expire. Check your device data sheet to see if the `reset` instruction is implemented by your device.

Example

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/

void main(void) {
  SYSTEM_Initialize();
  DISP_Initialize();

  INTERRUPT_GlobalInterruptEnable();

  while (1) {
    if(DISP_needsUpdate) {
      if(DISP_update() == 0)
        RESET();  // something's gone horribly wrong
    }
  }
}
```

9.2.23 SLEEP Macro

A macro that puts the device into sleep mode.

Include

`<xc.h>`

Prototype

`void SLEEP(void);`

Remarks

This macro executes a `sleep` instruction.

Example

```
#include "mcc_generated_files/mcc.h"
/* <xc.h> is automatically included by "mcc.h"; include this explicitly if not using MCC.*/

void main(void) {
  SYSTEM_Initialize();

  INTERRUPT_GlobalInterruptEnable();
```

```
  WeatherStation_initialize();

  while (1) {
    if (tmr1_tick == 1) {
      WeatherStation_Print();
      tmr1_tick = 0;
    }
    SLEEP(); // wait for interrupt
    NOP();   // ensure instruction pre-fetched while sleeping is safe to execute on awakening
  }
}
```

9.2.24 WRITETIMER Macros

A macro that writes the timer peripheral.

Include

`<xc.h>`

Prototype

`void WRITETIMERx(int n);`

Remarks

The `WRITETIMERx()` macro is available for PIC18 projects and writes the 2-byte argument to both bytes of the TMR*x* register, where *x* is one of the digits 0, 1 or 3. It ensures that the registers are written in the correct order, which cannot be guaranteed when accessing the entire TMR SFR directly.

Example

```
#include <xc.h>

int main(void)
{
  WRITETIMER0(0x4A);
}
```

10. Error and Warning Messages

Listed here are the MPLAB XC8 C Compiler error, warning, and advisory messages, with an explanation of each message. This is the complete and historical message set covering all former HI-TECH C compilers and all compiler versions. Not all messages shown here will be relevant for the compiler version you are using.

Messages have been assigned a unique number that appears in brackets before each message description. It is also printed by the compiler when the message is issued. The messages shown here are sorted by their number.

The name of the application(s) that could have produced the messages are listed in brackets opposite the error message. In some cases examples of code or options that could trigger the error are given. The use of * in the error message is used to represent a string that the compiler will substitute that is specific to that particular error.

Note that one problem in your C or assembler source code can trigger more than one error message. You should attempt to resolve errors or warnings in the order in which they are displayed.

10.1 Messages 0 Thru 499

(1) too many errors (*) (all applications)
The executing compiler application has encountered too many errors and will exit immediately. Other uncompiled source files will be processed, but the compiler applications that would normally be executed in due course will not be run. The number of errors that can be accepted is controlled using the `-fmax-errors` option (see 4.6.4.1 Max Errors Option.

(2) error/warning (*) generated but no description available (all applications)
The executing compiler application has emitted a message (advisory/warning/error), but there is no description available in the message description file (MDF) to print. This could be because the MDF is out-of-date, or the message issue has not been translated into the selected language.

(3) malformed error information on line * in file * (all applications)
The compiler has attempted to load the messages for the selected language, but the message description file (MDF) was corrupted and could not be read correctly.

(100) unterminated #if[n][def] block from line * (Preprocessor)
A `#if` or similar block was not terminated with a matching `#endif`, for example:

```
#if INPUT        /* error flagged here */
int main(void)
{
    run();
}                /* no #endif was found in this module */
```

(101) #* cannot follow #else (Preprocessor)
A `#else` or `#elif` has been used in the same conditional block as a `#else`. These can only follow a `#if`, for example:

```
#ifdef FOO
    result = foo;
#else
    result = bar;
#elif defined(NEXT)    /* the #else above terminated the #if */
    result = next(0);
#endif
```

(102) #* must be in an #if (Preprocessor)
The `#elif`, `#else` or `#endif` directive must be preceded by a matching `#if` line. If there is an apparently corresponding `#if` line, check for things like extra `#endif`s, or improperly terminated comments, for example:

```
#ifdef FOO
    result = foo;
#endif
    result = bar;
#elif defined(NEXT)   /* the #endif above terminated the #if */
    result = next(0);
#endif
```

(103) #error: * (Preprocessor)
This is a programmer generated error; there is a directive causing a deliberate error. This is normally used to check compile time defines, etc. Remove the directive to remove the error, but first determine why the directive is there.

(104) preprocessor #assert failure (Preprocessor)
The argument to a preprocessor `#assert` directive has evaluated to zero. This is a programmer induced error.

```
#assert SIZE == 4  /* size should never be 4 */
```

(105) no #asm before #endasm (Preprocessor)
A `#endasm` operator has been encountered, but there was no previous matching `#asm`, for example:

```
void cleardog(void)
{
    clrwdt
    #endasm  /* in-line assembler ends here, only where did it begin? */
}
```

(106) nested #asm directives (Preprocessor)
It is not legal to nest `#asm` directives. Check for a missing or misspelled `#endasm` directive, for example:

```
#asm
    MOVE  r0, #0aah
#asm      ; previous #asm must be closed before opening another
    SLEEP
#endasm
```

(107) illegal # directive "*" (Preprocessor, Parser)
The compiler does not understand the `#` directive. It is probably a misspelling of a directive token, for example:

```
#indef DEBUG  /* oops -- that should be #undef DEBUG */
```

(108) #if[n][def] without an argument (Preprocessor)
The preprocessor directives `#if`, `#ifdef`, and `#ifndef` must have an argument. The argument to `#if` should be an expression, while the argument to `#ifdef` or `#ifndef` should be a single name, for example:

```
#if            /* oops -- no argument to check */
output = 10;
#else
output = 20;
#endif
```

(109) #include syntax error (Preprocessor)
The syntax of the filename argument to `#include` is invalid. The argument to `#include` must be a valid file name, either enclosed in double quotes `" "` or angle brackets `< >`. Spaces should not be included and the closing quote or bracket must be present. There should be nothing else on the line other than comments, for example:

```
#include stdio.h  /* oops -- should be: #include <stdio.h> */
```

(110) too many file arguments; usage: cpp [input [output]] (Preprocessor)
CPP should be invoked with at most two file arguments. Contact Microchip Technical Support if the preprocessor is being executed by a compiler driver.

(111) redefining preprocessor macro "*" (Preprocessor)
The macro specified is being redefined to something different than the original definition. If you want to deliberately redefine a macro, use #undef first to remove the original definition, for example:

```
#define ONE 1
/* elsewhere: */
/* Is this correct? It will overwrite the first definition. */
#define ONE one
```

(112) #define syntax error (Preprocessor)
A macro definition has a syntax error. This could be due to a macro or formal parameter name that does not start with a letter or a missing closing parenthesis,), for example:

```
#define FOO(a, 2b)  bar(a, 2b)  /* 2b is not to be! */
```

(113) unterminated string in preprocessor macro body (Preprocessor, Assembler)
A macro definition contains a string that lacks a closing quote.

(114) illegal #undef argument (Preprocessor)
The argument to `#undef` must be a valid name. It must start with a letter, for example:

```
#undef 6YYY  /* this isn't a valid symbol name */
```

(115) recursive preprocessor macro definition of "*" defined by "*" (Preprocessor)
The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself.

(116) end of file within preprocessor macro argument from line * (Preprocessor)
A macro argument has not been terminated. This probably means the closing parenthesis has been omitted from a macro invocation. The line number given is the line where the macro argument started, for example:

```
#define FUNC(a, b) func(a+b)
FUNC(5, 6;    /* oops -- where is the closing bracket? */
```

(117) misplaced constant in #if (Preprocessor)
A constant in a `#if` expression should only occur in syntactically correct places. This error is probably caused by omission of an operator, for example:

```
#if FOO BAR  /* oops -- did you mean: #if FOO == BAR ? */
```

(118) stack overflow processing #if expression (Preprocessor)
The preprocessor filled up its expression evaluation stack in a `#if` expression. Simplify the expression – it probably contains too many parenthesized subexpressions.

(119) invalid expression in #if line (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(120) operator "*" in incorrect context (Preprocessor)
An operator has been encountered in a `#if` expression that is incorrectly placed (two binary operators are not separated by a value), for example:

```
#if FOO * % BAR == 4  /* what is "* %" ? */
#define BIG
#endif
```

(121) expression stack overflow at operator "*" (Preprocessor)
Expressions in `#if` lines are evaluated using a stack with a size of 128. It is possible for very complex expressions to overflow this. Simplify the expression.

(122) unbalanced parenthesis at operator "*" (Preprocessor)
The evaluation of a `#if` expression found mismatched parentheses. Check the expression for correct parenthesizing, for example:

```
#if ((A) + (B)  /* oops -- a missing ), I think */
#define ADDED
#endif
```

(123) misplaced "?" or ":"; previous operator is "*" (Preprocessor)
A colon operator has been encountered in a `#if` expression that does not match up with a corresponding `?` operator, for example:

```
#if XXX : YYY  /* did you mean:  #if COND ? XXX : YYY */
```

(124) illegal character "*" in #if (Preprocessor)
There is a character in a `#if` expression that should not be there. Valid characters are the letters, digits and those comprising the acceptable operators, for example:

```
#if YYY  /* what are these characters doing here? */
int m;
#endif
```

(125) illegal character (* decimal) in #if (Preprocessor)
There is a non-printable character in a `#if` expression that should not be there. Valid characters are the letters, digits and those comprising the acceptable operators, for example:

```
#if ^S YYY  /* what is this control characters doing here? */
int m;
#endif
```

(126) strings can't be used in #if (Preprocessor)
The preprocessor does not allow the use of strings in `#if` expressions, for example:

```
/* no string operations allowed by the preprocessor */
#if MESSAGE > "hello"
#define DEBUG
#endif
```

(127) bad syntax for defined() in #[el]if (Preprocessor)
The `defined()` pseudo-function in a preprocessor expression requires its argument to be a single name. The name must start with a letter and should be enclosed in parentheses, for example:

```
/* oops -- defined expects a name, not an expression */
#if defined(a&b)
input = read();
#endif
```

(128) illegal operator in #if (Preprocessor)
A `#if` expression has an illegal operator. Check for correct syntax, for example:

```
#if FOO = 6  /* oops -- should that be: #if FOO == 5 ? */
```

(129) unexpected “\” in #if (Preprocessor)
The backslash is incorrect in the `#if` statement, for example:

```
#if FOO == \34
#define BIG
#endif
```

(130) unknown type “*” in #[el]if sizeof() (Preprocessor)
An unknown type was used in a preprocessor `sizeof()`. The preprocessor can only evaluate `sizeof()` with basic types, or pointers to basic types, for example:

```
#if sizeof(unt) == 2  /* should be: #if sizeof(int) == 2 */
i = 0xFFFF;
#endif
```

(131) illegal type combination in #[el]if sizeof() (Preprocessor)
The preprocessor found an illegal type combination in the argument to `sizeof()` in a `#if` expression, for example:

```
/* To sign, or not to sign, that is the error. */
#if sizeof(signed unsigned int) == 2
i = 0xFFFF;
#endif
```

(132) no type specified in #[el]if sizeof() (Preprocessor)
`sizeof()` was used in a preprocessor `#if` expression, but no type was specified. The argument to `sizeof()` in a preprocessor expression must be a valid simple type, or pointer to a simple type, for example:

```
#if sizeof()  /* oops -- size of what? */
i = 0;
#endif
```

(133) unknown type code (0x*) in #[el]if sizeof() (Preprocessor)
The preprocessor has made an internal error in evaluating a `sizeof()` expression. Check for a malformed type specifier. This is an internal error. Contact Microchip Technical Support with details.

(134) syntax error in #[el]if sizeof() (Preprocessor)
The preprocessor found a syntax error in the argument to `sizeof()` in a `#if` expression. Probable causes are mismatched parentheses and similar things, for example:

```
#if sizeof(int == 2)  // oops - should be: #if sizeof(int) == 2
i = 0xFFFF;
#endif
```

(135) unknown operator (*) in #if (Preprocessor)
The preprocessor has tried to evaluate an expression with an operator it does not understand. This is an internal error. Contact Microchip Technical Support with details.

(137) strange character “*” after ## (Preprocessor)
A character has been seen after the token catenation operator `##` that is neither a letter nor a digit. Because the result of this operator must be a legal token, the operands must be tokens containing only letters and digits, for example:

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(138) strange character (*) after ## (Preprocessor)
An unprintable character has been seen after the token catenation operator `##` that is neither a letter nor a digit. Because the result of this operator must be a legal token, the operands must be tokens containing only letters and digits, for example:

```
/* the ' character will not lead to a valid token */
#define cc(a, b) a ## 'b
```

(139) end of file in comment (Preprocessor)
End of file was encountered inside a comment. Check for a missing closing comment flag, for example:

```
/* Here the comment begins. I'm not sure where I end, though
}
```

(140) can't open * file "*": * (Driver, Preprocessor, Code Generator, Assembler)
The command file specified could not be opened for reading. Confirm the spelling and path of the file specified on the command line, for example:

```
xc8 @communds
```

should that be:

```
xc8 @commands
```

(141) can't open * file "*": * (Any)
An output file could not be created. Confirm the spelling and path of the file specified on the command line.

(144) too many nested #if blocks (Preprocessor)
`#if`, `#ifdef`, etc., blocks can only be nested to a maximum of 32.

(146) #include filename too long (Preprocessor)
A filename constructed while looking for an include file has exceeded the length of an internal buffer. Because this buffer is 4096 bytes long, this is unlikely to happen.

(147) too many #include directories specified (Preprocessor)
A maximum of 7 directories can be specified for the preprocessor to search for include files. The number of directories specified with the driver is too many.

(148) too many arguments for preprocessor macro (Preprocessor)
A macro can only have up to 31 parameters, per the C Standard.

(149) preprocessor macro work area overflow (Preprocessor)
The total length of a macro expansion has exceeded the size of an internal table. This table is normally 32768 bytes long. Thus any macro expansion must not expand to a total of more than 32K bytes.

(150) illegal "__" preprocessor macro "*" (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(151) too many arguments in preprocessor macro expansion (Preprocessor)
There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

(152) bad dp/nargs in openpar(): c = * (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(153) out of space in preprocessor macro * argument expansion (Preprocessor)
A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

(155) work buffer overflow concatenating "*" (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(156) work buffer "*" overflow (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(157) can't allocate * bytes of memory (Code Generator, Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(158) invalid disable in preprocessor macro "*" (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(159) too many calls to unget() (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(161) control line "*" within preprocessor macro expansion (Preprocessor)
A preprocessor control line (one starting with a `#`) has been encountered while expanding a macro. This should not happen.

(162) #warning: * (Preprocessor, Driver)
This warning is either the result of user-defined `#warning` preprocessor directive, or the driver encountered a problem reading the map file. If the latter, contact Microchip Technical Support with details

(163) unexpected text in control line ignored (Preprocessor)
This warning occurs when extra characters appear on the end of a control line. The extra text will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the text as a comment, for example:

```
#if defined(END)
#define NEXT
#endif END     /* END would be better in a comment here */
```

(164) #include filename "*" was converted to lower case (Preprocessor)
The `#include` file name had to be converted to lowercase before it could be opened, for example:

```
#include <STDIO.H>  /* oops -- should be: #include <stdio.h> */
```

(165) #include filename "*" does not match actual name (check upper/lower case) (Preprocessor)
In Windows versions this means the file to be included actually exists and is spelled the same way as the `#include` filename; however, the case of each does not exactly match. For example, specifying `#include "code.c"` will include `Code.c`, if it is found. In Linux versions this warning could occur if the file wasn't found.

(166) too few values specified with option "*" (Preprocessor)
The list of values to the preprocessor (CPP) `-S` option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver. The values passed to this option represent the sizes of `char`, `short`, `int`, `long`, `float` and `double` types.

(167) too many values specified with -S option; "*" unused Preprocessor)
There were too many values supplied to the `-S` preprocessor option. See message 166.

(168) unknown option “*” (Any)
The option given to the component which caused the error is not recognized.

(169) strange character (*) after ## (Preprocessor)
There is an unexpected character after `#`.

(170) symbol “*” in undef was never defined (Preprocessor)
The symbol supplied as argument to `#undef` was not already defined. This warning can be disabled with some compilers. This warning can be avoided with code like:

```
#ifdef SYM
#undef SYM  /* only undefine if defined */
#endif
```

(171) wrong number of preprocessor macro arguments for “*” (* instead of *) (Preprocessor)
A macro has been invoked with the wrong number of arguments, for example:

```
#define ADD(a, b) (a+b)
ADD(1, 2, 3)        /* oops -- only two arguments required */
```

(172) formal parameter expected after # (Preprocessor)
The stringization operator `#` (not to be confused with the leading `#` used for preprocessor control lines) must be followed by a formal macro parameter, for example:

```
#define str(x) #y  /* oops -- did you mean x instead of y? */
```

If you need to stringize a token, you will need to define a special macro to do it, for example:

```
#define __mkstr__(x) #x
```

then use `__mkstr__(`*`token`*`)` wherever you need to convert *`token`* into a string.

(173) undefined symbol “*” in #if; 0 used (Preprocessor)
A symbol on a `#if` expression was not a defined preprocessor macro. For the purposes of this expression, its value has been taken as zero. This warning can be disabled with some compilers. Example:

```
#if FOO+BAR    /* e.g. FOO was never #defined */
#define GOOD
#endif
```

(174) multi-byte constant “*” isn’t portable (Preprocessor)
Multi-byte constants are not portable; and will be rejected by later passes of the compiler, for example:

```
#if CHAR == ’ab’
#define MULTI
#endif
```

(175) division by zero in #if; zero result assumed (Preprocessor)
Inside a `#if` expression, there is a division by zero which has been treated as yielding zero, for example:

```
#if foo/0  /* divide by 0: was this what you were intending? */
int a;
#endif
```

(176) missing newline (Preprocessor)
A new line is missing at the end of the line. Each line, including the last line, must have a new line at the end. This problem is normally introduced by editors.

(177) symbol “*” in -U option was never defined (Preprocessor)
A macro name specified in a `-U` option to the preprocessor was not initially defined, and thus cannot be undefined.

(179) nested comments (Preprocessor)
This warning is issued when nested comments are found. A nested comment can indicate that a previous closing comment marker is missing or malformed, for example:

```
output = 0;  /* a comment that was left unterminated
flag = TRUE; /* next comment:
hey, where did this line go? */
```

(180) unterminated comment in included file (Preprocessor)
Comments begun inside an included file must end inside the included file.

(181) non-scalar types can’t be converted to other types (Parser)
You cannot convert a structure, union, or array to another type, for example:

```
struct TEST test;
struct TEST * sp;
sp = test;       /* oops -- did you mean: sp = &test; ? */
```

(182) illegal conversion between types (Parser)
This expression implies a conversion between incompatible types, i.e., a conversion of a structure type into an integer, for example:

```
struct LAYOUT layout;
int i;
layout = i;          /* int cannot be converted to struct */
```

Note that even if a structure only contains an `int` , for example, it cannot be assigned to an `int` variable and vice versa.

(183) function or function pointer required (Parser)
Only a function or function pointer can be the subject of a function call, for example:

```
int a, b, c, d;
a = b(c+d);   /* b is not a function -- did you mean a = b*(c+d) ? */
```

(184) calling an interrupt function is illegal (Parser)
A function-qualified interrupt cannot be called from other functions. It can only be called by a hardware (or software) interrupt. This is because an interrupt function has special function entry and exit code that is appropriate only for calling from an interrupt. An interrupt function can call other non-interrupt functions.

(185) function does not take arguments (Parser, Code Generator)
This function has no parameters, but it is called here with one or more arguments, for example:

```
int get_value(void);
int main(void)
{
int input;
input = get_value(6);  /* oops --
parameter should not be here */
}
```

(186) too many function arguments (Parser)
This function does not accept as many arguments as there are here.

```
void add(int a, int b);
add(5, 7, input);      /* call has too many arguments */
```

(187) too few function arguments (Parser)
This function requires more arguments than are provided in this call, for example:

```
void add(int a, int b);
add(5);                  /* this call needs more arguments */
```

(188) constant expression required (Parser)
In this context an expression is required that can be evaluated to a constant at compile time, for example:

```
int a;
switch(input) {
case a:  /* oops!
cannot use variable as part of a case label */
input++;
}
```

(189) illegal type for array dimension (Parser)
An array dimension must be either an integral type or an enumerated value.

```
int array[12.5];  /* oops -- twelve and a half elements, eh? */
```

(190) illegal type for index expression (Parser)
An index expression must be either integral or an enumerated value, for example:

```
int i, array[10];
i = array[3.5];   /* oops --
exactly which element do you mean? */
```

(191) cast type must be scalar or void (Parser)
A typecast (an abstract type declarator enclosed in parentheses) must denote a type which is either scalar (i.e., not an array or a structure) or the type `void`, for example:

```
lip = (long [])input;  /* oops -- possibly: lip = (long *)input */
```

(192) undefined identifier "*" (Parser)
This symbol has been used in the program, but has not been defined or declared. Check for spelling errors if you think it has been defined.

(193) not a variable identifier "*" (Parser)
This identifier is not a variable; it can be some other kind of object, i.e., a label.

(194) ")" expected (Parser)
A closing parenthesis,), was expected here. This can indicate you have left out this character in an expression, or you have some other syntax error. The error is flagged on the line at which the code first starts to make no sense. This can be a statement following the incomplete expression, for example:

```
if(a == b  /* the closing parenthesis is missing here */
b = 0;   /* the error is flagged here */
```

(195) expression syntax (Parser)
This expression is badly formed and cannot be parsed by the compiler, for example:

```
a /=% b;  /* oops -- possibly that should be: a /= b; */
```

(196) struct/union required (Parser)
A structure or union identifier is required before a dot "`.`" , for example:

```
int a;
a.b = 9;  /* oops -- a is not a structure */
```

(197) struct/union member expected (Parser)
A structure or union member name must follow a dot "`.`" or an arrow ("`->`").

(198) undefined struct/union "*" (Parser)
The specified structure or union tag is undefined, for example:

```
struct WHAT what;  /* a definition for WHAT was never seen */
```

(199) logical type required (Parser)
The expression used as an operand to `if`, `while` statements or to boolean operators like `!` and `&&` must be a scalar integral type, for example:

```
struct FORMAT format;
if(format)            /* this operand must be a scaler type */
    format.a = 0;
```

(200) taking the address of a register variable is illegal (Parser)
A variable declared register cannot have storage allocated for it in memory, and thus it is illegal to attempt to take the address of it by applying the `&` operator, for example:

```
int * proc(register int in)
{
    int * ip = &in;
    /* oops -- in cannot have an address to take */
    return ip;
}
```

(201) taking the address of this object is illegal (Parser)
The expression which was the operand of the `&` operator is not one that denotes memory storage ("an lvalue") and therefore its address cannot be defined, for example:

```
ip = &8;  /* oops -- you cannot take the address of a literal */
```

(202) only lvalues can be assigned to or modified (Parser)
Only an lvalue (i.e., an identifier or expression directly denoting addressable storage) can be assigned to or otherwise modified, for example:

```
int array[10];
int * ip;
char c;
array = ip;   /* array is not a variable, it cannot be written to */
```

A typecast does not yield an lvalue, for example:

```
/* the contents of c cast to int is only a intermediate value */
(int)c = 1;
```

However, you can write this using pointers:

```
*(int *)&c = 1
```

(203) illegal operation on bit variable (Parser)
Not all operations on bit variables are supported. This operation is one of those, for example:

```
bit   b;
int * ip;
ip = &b;  /* oops -- cannot take the address of a bit object */
```

(204) void function can't return a value (Parser)
A void function cannot return a value. Any return statement should not be followed by an expression, for example:

```
void run(void)
{
    step();
    return 1;  /* either run should not be void, or remove the 1 */
}
```

(205) integral type required (Parser)
This operator requires operands that are of integral type only.

(206) illegal use of void expression (Parser)
A `void` expression has no value and therefore you cannot use it anywhere an expression with a value is required, i.e., as an operand to an arithmetic operator.

(207) simple type required for "*" (Parser)
A simple type (i.e., not an array or structure) is required as an operand to this operator.

(208) operands of "*" not same type (Parser)
The operands of this operator are of different pointers, for example:

```
int * ip;
char * cp, * cp2;
cp = flag ? ip : cp2;  /* result of ? : will be int * or char * */
```

Possibly, you meant something like:

```
cp = flag ? (char *)ip : cp2;
```

(209) type conflict (Parser)
The operands of this operator are of incompatible types.

(210) bad size list (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(211) taking sizeof bit is illegal (Parser)
It is illegal to use the `sizeof()` operator with the C `__bit` type. When used against a type, the `sizeof()` operator gives the number of bytes required to store an object that type. Therefore its usage with the `__bit` type make no sense and it is an illegal operation.

(212) missing number after pragma "pack" (Parser)
The `pragma pack` requires a decimal number as argument. This specifies the alignment of each member within the structure. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries, for example:

```
#pragma pack  /* what is the alignment value */
```

Possibly, you meant something like:

```
#pragma pack 2
```

(214) missing number after pragma "interrupt_level" (Parser)

The `pragma interrupt_level` requires an argument to indicate the interrupt level. It will be the value 1 for mid-range devices, or 1 or 2 or PIC18 devices.

(215) missing argument to pragma "switch" (Parser)

The `pragma switch` requires an argument of auto, direct or simple, for example:

```
#pragma switch  /* oops -- this requires a switch mode */
```

Possibly, you meant something like:

```
#pragma switch simple
```

(216) missing argument to pragma "psect" (Parser)

The `pragma psect` requires an argument of the form *`oldname = newname`* where *`oldname`* is an existing psect name known to the compiler and *`newname`* is the desired new name, for example:

```
#pragma psect  /* oops -- this requires an psect to redirect */
```

Possibly, you meant something like:

```
#pragma psect text=specialtext
```

(218) missing name after pragma "inline" (Parser)

The `inline pragma` expects the name of a function to follow. The function name must be recognized by the code generator for it to be expanded; other functions are not altered, for example:

```
#pragma inline  /* what is the function name? */
```

Possibly, you meant something like:

```
#pragma inline memcpy
```

(219) missing name after pragma "printf_check" (Parser)

The `printf_check pragma` expects the name of a function to follow. This specifies printf-style format string checking for the function, for example:

```
#pragma printf_check  /* what function is to be checked? */
```

Possibly, you meant something like:

```
#pragma printf_check sprintf
```

Pragmas for all the standard printf-like function are already contained in `<stdio.h>`.

(220) exponent expected (Parser)

A floating-point constant must have at least one digit after the `e` or `E`, for example:

```
float f;
f = 1.234e;  /* oops -- what is the exponent? */
```

(221) hexadecimal digit expected (Parser)

After `0x` should follow at least one of the HEX digits 0-9 and A-F or a-f, for example:

```
a = 0xg6;  /* oops -- was that meant to be a = 0xf6 ? */
```

(222) binary digit expected (Parser)
A binary digit was expected following the 0b format specifier, for example:

```
i = 0bf000;  /* oops -- f000 is not a base two value */
```

(223) digit out of range (Parser, Assembler)
A digit in this number is out of range of the radix for the number, i.e., using the digit 8 in an octal number, or HEX digits A-F in a decimal number. An octal number is denoted by the digit string commencing with a zero, while a HEX number starts with 0X or 0x. For example:

```
int a = 058; /* leading 0 implies octal which has digits 0 - 7 */
```

(224) illegal "#" directive (Parser)
An illegal # preprocessor has been detected. Likely, a directive has been misspelled in your code somewhere.

(225) missing character in character constant (Parser)
The character inside the single quotes is missing, for example:

```
char c = "";  /* the character value of what? */
```

(226) char const too long (Parser)
A character constant enclosed in single quotes cannot contain more than one character, for example:

```
c = '12';  /* oops -- only one character can be specified */
```

(227) "." expected after ".." (Parser)
The only context in which two successive dots can appear is as part of the ellipsis symbol, which must have 3 dots (an ellipsis is used in function prototypes to indicate a variable number of parameters).

Either .. was meant to be an ellipsis symbol which would require you to add an extra dot, or it was meant to be a structure member operator which would require you to remove one dot.

(228) illegal character (*) (Parser)
This character is illegal in the C code. Valid characters are the letters, digits and those comprising the acceptable operators, for example:

```
c = a;  /* oops -- did you mean c = 'a'; ? */
```

(229) unknown qualifier "*" given to -A (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(230) missing argument to -A (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(231) unknown qualifier "*" given to -I (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(232) missing argument to -I (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(233) bad -Q option "*" (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(234) close error (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(236) simple integer expression required (Parser)
A simple integral expression is required after the `__at()` operator, used to associate an absolute address with a variable, for example:

```
int address;
char LOCK __at(address);
```

(237) function "*" redefined (Parser)
More than one definition for a function has been encountered in this module. Function overloading is illegal, for example:

```
int twice(int a)
{
    return a*2;
}
/* only one prototype & definition of rv can exist */
long twice(long a)
{
    return a*2;
}
```

(238) illegal initialization (Parser)
You cannot initialize a `typedef` declaration, because it does not reserve any storage that can be initialized, for example:

```
/* oops -- uint is a type, not a variable */
typedef unsigned int uint = 99;
```

(239) identifier "*" redefined (from line *) (Parser)
This identifier has already been defined in the same scope. It cannot be defined again, for example:

```
int a;  /* a filescope variable called "a" */
int a;  /* attempting to define another of the same name */
```

Note that variables with the same name, but defined with different scopes, are legal; but; not recommended.

(240) too many initializers (Parser)
There are too many initializers for this object. Check the number of initializers against the object definition (array or structure), for example:

```
/* three elements, but four initializers */
int ivals[3] = { 2, 4, 6, 8};
```

(241) initialization syntax (Parser)
The initialization of this object is syntactically incorrect. Check for the correct placement and number of braces and commas, for example:

```
int iarray[10] = {{'a', 'b', 'c'};
/* oops -- one two many {s */
```

(242) illegal type for switch expression (Parser)
A `switch()` operator must have an expression that is either an integral type or an enumerated value, e.g:

```
double d;
switch(d) {  /* oops -- this must be integral */
case '1.0':
    d = 0;
}
```

(243) inappropriate break/continue (Parser)
A `break` or `continue` statement has been found that is not enclosed in an appropriate control structure. A continue can only be used inside a `while`, `for`, or `do while` loop, while `break` can only be used inside those loops or a `switch` statement, for example:

```
switch(input) {
case 0:
    if(output == 0)
        input = 0xff;
    } /* oops! this should not be here; it closed the switch */
    break;     /* this should be inside the switch */
```

(244) "default" case redefined (Parser)
Only one `default` label is allowed to be in a `switch()` statement. You have more than one, for example:

```
switch(a) {
default:     /* if this is the default case... */
    b = 9;
    break;
default:     /* then what is this? */
    b = 10;
    break;
```

(245) "default" case not in switch (Parser)
A label has been encountered called `default`, but it is not enclosed by a `switch` statement. A default label is only legal inside the body of a `switch` statement.

If there is a `switch` statement before this default label, there could be one too many closing braces in the `switch` code. That would prematurely terminate the `switch` statement. See message 246.

(246) case label not in switch (Parser)
A `case` label has been encountered, but there is no enclosing `switch` statement. A `case` label can only appear inside the body of a `switch` statement.

If there is a `switch` statement before this `case` label, there might be one too many closing braces in the `switch` code. That would prematurely terminate the switch statement, for example:

```
switch(input) {
case '0':
    count++;
    break;
case '1':
    if(count>MAX)
        count= 0;
    }          /* oops -- this shouldn't be here */
    break;
case '2':    /* error flagged here */
```

(247) duplicate label "*" (Parser)
The same name is used for a label more than once in this function. Note that the scope of labels is the entire function, not just the block that encloses a label, for example:

```
start:
if(a > 256)
    goto end;
start:           /* error flagged here */
if(a == 0)
    goto start;  /* which start label do I jump to? */
```

(248) inappropriate “else” (Parser)
An `else` keyword has been encountered that cannot be associated with an `if` statement. This can mean there is a missing brace or other syntactic error, for example:

```
/* here is a comment which I have forgotten to close...
if(a > b) {
    c = 0;
/* ... that will be closed here, thus removing the "if" */
else        /* my "if" has been lost */
    c = 0xff;
```

(249) probable missing “}” in previous block (Parser)
The compiler has encountered what looks like a function or other declaration, but the preceding function was ended with a closing brace. This probably means that a closing brace has been omitted from somewhere in the previous function, although it might not be the last one, for example:

```
void set(char a)
{
    PORTA = a;
    /* the closing brace was left out here */
void clear(void)  /* error flagged here */
{
    PORTA = 0;
}
```

(251) array dimension redeclared (Parser)
An array dimension has been declared as a different non-zero value from its previous declaration. It is acceptable to redeclare the size of an array that was previously declared with a zero dimension; but, not otherwise, for example:

```
extern int array[5];
int array[10];        /* oops -- has it 5 or 10 elements? */
```

(252) argument * conflicts with prototype (Parser)
The argument specified (argument 0 is the left most argument) of this function definition does not agree with a previous prototype for this function, for example:

```
/* this is supposedly calc’s prototype */
extern int calc(int, int);
int calc(int a, long int b)  /* hmmm -- which is right? */
{                            /* error flagged here */
    return sin(b/a);
}
```

(253) argument list conflicts with prototype (Parser)
The argument list in a function definition is not the same as a previous prototype for that function. Check that the number and types of the arguments are all the same.

```
extern int calc(int);   /* this is supposedly calc’s prototype */
int calc(int a, int b)  /* hmmm -- which is right? */
{                       /* error flagged here */
    return a + b;
}
```

(254) undefined *: “*” (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(255) not a member of the struct/union “*” (Parser)
This identifier is not a member of the structure or union type with which it used here, for example:

```
struct {
    int a, b, c;
} data;
```

```
if(data.d)    /* oops -- there is no member d in this structure */
    return;
```

(256) too much indirection (Parser)
A pointer declaration can only have 16 levels of indirection.

(257) only “register” storage class allowed (Parser)
The only storage class allowed for a function parameter is `register`, for example:

```
void process(static int input)
```

(258) duplicate qualifier (Parser)
There are two occurrences of the same qualifier in this type specification. This can occur either directly or through the use of a `typedef`. Remove the redundant qualifier. For example:

```
typedef volatile int vint;
/* oops -- this results in two volatile qualifiers */
volatile vint very_vol;
```

(259) object can’t be qualified both far and near (Parser)
It is illegal to qualify a type as both `far` and `near`, for example:

```
far near int spooky;  /* oops -- choose far or near, not both */
```

(260) undefined enum tag “*” (Parser)
This `enum` tag has not been defined, for example:

```
enum WHAT what;  /* a definition for WHAT was never seen */
```

(261) struct/union member “*” redefined (Parser)
This name of this member of the `struct` or `union` has already been used in this `struct` or `union`, for example:

```
struct {
    int a;
    int b;
    int a;  /* oops -- a different name is required here */
} input;
```

(262) struct/union “*” redefined (Parser)
A structure or union has been defined more than once, for example:

```
struct {
    int a;
} ms;
struct {
    int a;
} ms;    /* was this meant to be the same name as above? */
```

(263) members can’t be functions (Parser)
A member of a structure or a union cannot be a function. It could be a pointer to a function, for example:

```
struct {
    int a;
    int get(int);  /* should be a pointer: int (*get)(int); */
} object;
```

(264) bad bitfield type (Parser)
A bit-field can only have a type of int (or unsigned), for example:

```
struct FREG {
    char b0:1;   /* these must be part of an int, not char */
    char   :6;
    char b7:1;
} freg;
```

(265) integer constant expected (Parser)
A colon appearing after a member name in a structure declaration indicates that the member is a bit-field. An integral constant must appear after the colon to define the number of bits in the bit-field, for example:

```
struct {
    unsigned first:  /* oops -- should be: unsigned first; */
    unsigned second;
} my_struct;
```

If this was meant to be a structure with bit-fields, then the following illustrates an example:

```
struct {
    unsigned first : 4;  /* 4 bits wide */
    unsigned second: 4;  /* another 4 bits */
} my_struct;
```

(266) storage class illegal (Parser)
A structure or union member cannot be given a storage class. Its storage class is determined by the storage class of the structure, for example:

```
struct {
    /* no additional qualifiers can be present with members */
    static int first;
} ;
```

(267) bad storage class (Code Generator)
The code generator has encountered a variable definition whose storage class is invalid, for example:

```
auto int foo; /* auto not permitted with global variables */
int power(static int a)  /* parameters cannot be static */
{
    return foo * a;
}
```

(268) inconsistent storage class (Parser)
A declaration has conflicting storage classes. Only one storage class should appear in a declaration, for example:

```
extern static int where;  /* so is it static or extern? */
```

(269) inconsistent type (Parser)
Only one basic type can appear in a declaration, for example:

```
int float input;  /* is it int or float? */
```

(270) variable can't have storage class "register" (Parser)
Only function parameters or auto variables can be declared using the `register` qualifier, for example:

```
register int gi;      /* this cannot be qualified register */
int process(register int input)  /* this is okay */
{
    return input + gi;
}
```

(271) type can't be long (Parser)
Only `int` and `double` can be qualified with long.

```
long char lc;  /* what? */
```

(272) type can't be short (Parser)
Only `int` can be modified with `short`, for example:

```
short float sf;  /* what? */
```

(273) type can't be both signed and unsigned (Parser)
The type modifiers `signed` and `unsigned` cannot be used together in the same declaration, as they have opposite meaning, for example:

```
signed unsigned int confused;  /* which is it? */
```

(274) type can't be unsigned (Parser)
A floating-point type cannot be made `unsigned`, for example:

```
unsigned float uf;  /* what? */
```

(275) "..." illegal in non-prototype argument list (Parser)
The ellipsis symbol can only appear as the last item in a prototyped argument list. It cannot appear on its own, nor can it appear after argument names that do not have types; i.e., K&R-style non-prototype function definitions. For example:

```
/* K&R-style non-prototyped function definition */
int kandr(a, b, ...)
    int a, b;
{
```

(276) type specifier required for prototyped argument (Parser)
A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

(277) can't mix prototyped and non-prototyped arguments (Parser)
A function declaration can only have all prototyped arguments (i.e., with types inside the parentheses) or all K&R style arguments (i.e., only names inside the parentheses and the argument types in a declaration list before the start of the function body), for example:

```
int plus(int a, b)  /* oops -- a is prototyped, b is not */
int b;
{
    return a + b;
}
```

(278) argument "*" redeclared (Parser)
The specified argument is declared more than once in the same argument list, for example:

```
/* cannot have two parameters called "a" */
int calc(int a, int a)
```

(279) initialization of function arguments is illegal (Parser)
A function argument cannot have an initializer in a declaration. The initialization of the argument happens when the function is called and a value is provided for the argument by the calling function, for example:

```
/* oops -- a is initialized when proc is called */
extern int proc(int a = 9);
```

(280) arrays of functions are illegal (Parser)
You cannot define an array of functions. You can, however, define an array of pointers to functions, for example:

```
int * farray[]();  /* oops -- should be: int (* farray[])(); */
```

(281) functions can't return functions (Parser)
A function cannot return a function. It can return a function pointer. A function returning a pointer to a function could be declared like this: `int (* (name()))()`. Note the many parentheses that are necessary to make the parts of the declaration bind correctly.

(282) functions can't return arrays (Parser)
A function can return only a scalar (simple) type or a structure. It cannot return an array.

(283) dimension required (Parser)
Only the most significant (i.e., the first) dimension in a multi-dimension array cannot be assigned a value. All succeeding dimensions must be present as a constant expression, for example:

```
/* This should be, for example: int arr[][7] */
int get_element(int arr[2][])
{
    return array[1][6];
}
```

(284) invalid dimension (Parser)
The array dimension specified is not valid. It must be larger than 0.

```
int array[0];  // oops -- you cannot have an array of size 0
```

(285) no identifier in declaration (Parser)
The identifier is missing in this declaration. This error can also occur when the compiler has been confused by such things as missing closing braces, for example:

```
void interrupt(void)  /* what is the name of this function? */
{
}
```

(286) declarator too complex (Parser)
This declarator is too complex for the compiler to handle. Examine the declaration and find a way to simplify it. If the compiler finds it too complex, so will anybody maintaining the code.

(287) arrays of bits or pointers to bit are illegal (Parser)
It is not legal to have an array of `__bit` objects, or a pointer to `bit` variable, for example:

```
bit barray[10];  /* wrong -- no bit arrays */
bit * bp;        /* wrong -- no pointers to bit variables */
```

(288) the type 'void' is applicable only to functions (Parser)
A variable cannot be `void`. Only a function can be void, for example:

```
int a;
void b;  /* this makes no sense */
```

(289) the specifier 'interrupt' is applicable only to functions (Parser)
The qualifier interrupt cannot be applied to anything except a function, for example:

```
/* variables cannot be qualified interrupt */
interrupt int input;
```

(290) illegal function qualifier(s) (Parser)
A qualifier has been applied to a function which makes no sense in this context. Some qualifier only make sense when used with an lvalue, i.e., `const` or `volatile`. This can indicate that you have forgotten a star, `*`, that is indicating that the function should return a pointer to a qualified object, for example:

```
const char ccrv(void) /* const * char ccrv(void) perhaps? */
{                     /* error flagged here */
    return ccip;
}
```

(291) K&R identifier "*" not an argument (Parser)
This identifier, that has appeared in a K&R style argument declarator, is not listed inside the parentheses after the function name, for example:

```
int process(input)
int unput;        /* oops -- that should be int input; */
{
}
```

(292) a function is not a valid parameter type (Parser)
A function parameter cannot be a function. It can be a pointer to a function, so perhaps a `*` has been omitted from the declaration.

(293) bad size in index_type() (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(294) can't allocate * bytes of memory (Code Generator, Hexmate)
This is an internal compiler error. Contact Microchip Technical Support with details.

(295) expression too complex (Parser)
This expression has caused overflow of the compiler's internal stack and should be rearranged or split into two expressions.

(296) out of memory (Objtohex)
This could be an internal compiler error. Contact Microchip Technical Support with details.

(297) bad argument (*) to tysize() (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(298) end of file in #asm (Preprocessor)
An end of file has been encountered inside a `#asm` block. This probably means the `#endasm` is missing or misspelled, for example:

```
#asm
MOV  r0, #55
MOV  [r1], r0
}                /* oops -- where is the #endasm */
```

(300) unexpected end of file (Parser)
An end-of-file in a C module was encountered unexpectedly, for example:

```
int main(void)
{
    init();
    run();    /* is that it? What about the close brace */
```

(301) end of file on string file (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(302) can't reopen "*": * (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(303) can't allocate * bytes of memory (line *) (Parser)
The parser was unable to allocate memory for the longest string encountered, as it attempts to sort and merge strings. Try reducing the number or length of strings in this module.

(306) can't allocate * bytes of memory for * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(307) too many qualifier names (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(308) too many case labels in switch (Code Generator)
There are too many case labels in this `switch` statement. The maximum allowable number of case labels in any one `switch` statement is 511.

(309) too many symbols (Assembler, Parser)
There are too many symbols for the assembler's symbol table. Reduce the number of symbols in your program.

(310) "]" expected (Parser)
A closing square bracket was expected in an array declaration or an expression using an array index, for example:

```
process(carray[idx);  /* oops --
should be: process(carray[idx]); */
```

(311) closing quote expected (Parser)
A closing quote was expected for the indicated string.

(312) "*" expected (Parser)
The indicated token was expected by the parser.

(313) function body expected (Parser)
Where a function declaration is encountered with K&R style arguments (i.e., argument names; but, no types inside the parentheses) a function body is expected to follow, for example:

```
/* the function block must follow, not a semicolon */
int get_value(a, b);
```

(314) ";" expected (Parser)
A semicolon is missing from a statement. A close brace or keyword was found following a statement with no terminating semicolon , for example:

```
while(a) {
    b = a--  /* oops -- where is the semicolon? */
}          /* error is flagged here */
```

Note: Omitting a semicolon from statements not preceding a close brace or keyword typically results in some other error being issued for the following code which the parser assumes to be part of the original statement.

(315) "{" expected (Parser)
An opening brace was expected here. This error can be the result of a function definition missing the opening brace, for example:

```
/* oops! no opening brace after the prototype */
void process(char c)
```

```
    return max(c, 10) * 2; /* error flagged here */
}
```

(316) “}” expected (Parser)

A closing brace was expected here. This error can be the result of a initialized array missing the closing brace, for example:

```
char carray[4] = { 1, 2, 3, 4;  /* oops -- no closing brace */
```

(317) “(” expected (Parser)

An opening parenthesis , `(`, was expected here. This must be the first token after a `while` , `for` , `if` , `do` or `asm` keyword, for example:

```
if a == b  /* should be: if(a == b) */
b = 0;
```

(318) string expected (Parser)

The operand to an `asm` statement must be a string enclosed in parentheses, for example:

```
asm(nop);  /* that should be asm("nop");
```

(319) while expected (Parser)

The keyword `while` is expected at the end of a `do` statement, for example:

```
do {
    func(i++);
}              /* do the block while what condition is true? */
if(i > 5)      /* error flagged here */
    end();
```

(320) “:” expected (Parser)

A colon is missing after a `case` label, or after the keyword `default`. This often occurs when a semicolon is accidentally typed instead of a colon, for example:

```
switch(input) {
case 0;         /* oops -- that should have been: case 0: */
    state = NEW;
```

(321) label identifier expected (Parser)

An identifier denoting a label must appear after `goto`, for example:

```
if(a)
    goto 20;
    /* this is not BASIC -- a valid C label must follow a goto */
```

(322) enum tag or “{” expected (Parser)

After the keyword `enum`, must come either an identifier that is, or will be, defined as an `enum` tag, or an opening brace, for example:

```
enum 1, 2;  /* should be, for example: enum {one=1, two }; */
```

(323) struct/union tag or “{” expected (Parser)

An identifier denoting a structure or union or an opening brace must follow a `struct` or `union` keyword, for example:

```
struct int a;  /* this is not how you define a structure */
```

You might mean something like:

```
struct {
    int a;
} my_struct;
```

(324) too many arguments for printf-style format string (Parser)
There are too many arguments for this format string. This is harmless, but can represent an incorrect format string, for example:

```
/* oops -- missed a placeholder? */
printf("%d - %d", low, high, median);
```

(325) error in printf-style format string (Parser)
There is an error in the format string here. The string has been interpreted as a `printf()` style format string, and it is not syntactically correct. If not corrected, this will cause unexpected behavior at runtime, for example:

```
printf("%l", lll);  /* oops -- possibly: printf("%ld", lll); */
```

(326) long int argument required in printf-style format string (Parser)
A `long` argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, for example:

```
printf("%lx", 2);  // possibly you meant: printf("%lx", 2L);
```

(327) long long int argument required in printf-style format string (Parser)
A `long long` argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, for example:

```
printf("%llx", 2);  // possibly you meant: printf("%llx", 2LL);
```

(328) int argument required in printf-style format string (Parser)
An integral argument is required for this printf-style format specifier. Check the number and order of format specifiers and corresponding arguments, for example:

```
printf("%d", 1.23); /* wrong number or wrong placeholder */
```

(329) double argument required in printf-style format string (Parser)
The printf format specifier corresponding to this argument is `%f` or similar, and requires a floating-point expression. Check for missing or extra format specifiers or arguments to printf.

```
printf("%f", 44);  /* should be: printf("%f", 44.0); */
```

(330) pointer to * argument required in printf-style format string (Parser)
A pointer argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

(331) too few arguments for printf-style format string (Parser)
There are too few arguments for this format string. This would result in a garbage value being printed or converted at runtime, for example:

```
printf("%d - %d", low); /* oops! where is the other value to print? */
```

(332) "interrupt_level" should be 0 to 7 (Parser)

The `pragma interrupt_level` must have an argument from 0 to 7; however, mid-range devices only use level 1. PIC18 devices can use levels 1 or 2. For example:

```
#pragma interrupt_level  9 /* oops -- the level is too high */
void interrupt isr(void)
{
    /* isr code goes here */
}
```

(333) unrecognized qualifier name after "strings" (Parser)

The `pragma strings` was passed a qualifier that was not identified, for example:

```
/* oops -- should that be #pragma strings const ? */
#pragma strings cinst
```

(334) unrecognized qualifier name after "printf_check" (Parser)

The `#pragma printf_check` was passed a qualifier that could not be identified, for example:

```
/* oops -- should that be const not cinst? */
#pragma printf_check(printf) cinst
```

(335) unknown pragma "*" (Parser)

An unknown `pragma` directive was encountered, for example:

```
#pragma rugsused myFunc w  /* I think you meant regsused */
```

(336) string concatenation across lines (Parser)

Strings on two lines will be concatenated. Check that this is the desired result, for example:

```
char * cp = "hi"
"there";    /* this is okay, but is it what you had intended? */
```

(337) line does not have a newline on the end (Parser)

The last line in the file is missing the newline (operating system dependent character) from the end. Some editors will create such files, which can cause problems for include files. The C standard requires all source files to consist of complete lines only.

(338) can't create * file "*" (Any)

The application tried to create or open the named file, but it could not be created. Check that all file path names are correct.

(339) initializer in extern declaration (Parser)

A declaration containing the keyword `extern` has an initializer. This overrides the `extern` storage class, because to initialize an object it is necessary to define (i.e., allocate storage for) it, for example:

```
extern int other = 99;  /* if it’s extern and not allocated
storage, how can it be initialized? */
```

(340) string not terminated by null character (Parser)

A `char` array is being initialized with a string literal larger than the array. Hence there is insufficient space in the array to safely append a null terminating character, for example:

```
char foo[5] = "12345"; /* the string stored in foo won’t have
a null terminating, i.e.
foo = [’1’, ’2’, ’3’, ’4’, ’5’] */
```

(343) implicit return at end of non-void function (Parser)
A function that has been declared to return a value has an execution path that will allow it to reach the end of the function body, thus returning without a value. Either insert a `return` statement with a value, or if the function is not to return a value, declare it `void`, for example:

```
int mydiv(double a, int b)
{
    if(b != 0)
        return a/b;   /* what about when b is 0? */
}                  /* warning flagged here */
```

(344) non-void function returns no value (Parser)
A function that is declared as returning a value has a `return` statement that does not specify a return value, for example:

```
int get_value(void)
{
    if(flag)
        return val++;
    return;
    /* what is the return value in this instance? */
}
```

(345) unreachable code (Parser)
This section of code will never be executed, because there is no execution path by which it could be reached, for example:

```
while(1)          /* how does this loop finish? */
    process();
flag = FINISHED;  /* how do we get here? */
```

(346) declaration of "*" hides outer declaration (Parser)
An object has been declared that has the same name as an outer declaration (i.e., one outside and preceding the current function or block). This is legal, but can lead to accidental use of one variable when the outer one was intended, for example:

```
int input;         /* input has filescope */
void process(int a)
{
    int input;       /* local blockscope input */
    a = input;       /* this will use the local variable. Is this right? */
```

(347) external declaration inside function (Parser)
A function contains an `extern` declaration. This is legal but is invariably not desirable as it restricts the scope of the function declaration to the function body. This means that if the compiler encounters another declaration, use, or definition of the `extern` object later in the same file, it will no longer have the earlier declaration and thus will be unable to check that the declarations are consistent. This can lead to strange behavior of your program or signature errors at link time. It will also hide any previous declarations of the same thing, again subverting the compiler's type checking. As a general rule, always declare `extern` variables and functions outside any other functions. For example:

```
int process(int a)
{
    /* this would be better outside the function */
    extern int away;
    return away + a;
}
```

(348) auto variable "*" should not be qualified (Parser)
An auto variable should not have qualifiers such as `near` or `far` associated with it. Its storage class is implicitly defined by the stack organization. An auto variable can be qualified with `static`, but it is then no longer auto.

(349) non-prototyped function declaration for "*" (Parser)
A function has been declared using old-style (K&R) arguments. It is preferable to use prototype declarations for all functions, for example:

```
int process(input)
int input;          /* warning flagged here */
{
}
```

This would be better written:

```
int process(int input)
{
}
```

(350) unused * "*" (from line *) (Parser)
The indicated object was never used in the function or module being compiled. Either this object is redundant, or the code that was meant to use it was excluded from compilation or misspelled the name of the object. Note that the symbols `rcsid` and `sccsid` are never reported as being unused.

(352) float parameter coerced to double (Parser)
Where a non-prototyped function has a parameter declared as `float`, the compiler converts this to a `double`. This is because the default C type conversion conventions provide that when a floating-point number is passed to a non-prototyped function, it is converted to `double`. It is important that the function declaration be consistent with this convention, for example:

```
double inc_flt(f)  /* f will be converted to double */
float f;           /* warning flagged here */
{
    return f * 2;
}
```

(353) sizeof external array "*" is zero (Parser)
The size of an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

(354) possible pointer truncation (Parser)
A pointer qualified far has been assigned to a default pointer, or a pointer qualified near, or a default pointer has been assigned to a pointer qualified near. This can result in truncation of the pointer and loss of information, depending on the memory model in use.

(355) implicit signed to unsigned conversion (Parser)
A signed number is being assigned or otherwise converted to a larger unsigned type. Under the ANSI C "value preserving" rules, this will result in the signed value being first sign-extended to a signed number the size of the target type, then converted to unsigned (which involves no change in bit pattern). An unexpected sign extension can occur. To ensure this does not happen, first convert the signed value to an unsigned equivalent, for example:

```
signed char sc;
unsigned int ui;
ui = sc;    /* if sc contains 0xff, ui will contain 0xffff for example */
```

will perform a sign extension of the char variable to the longer type. If you do not want this to take place, use a cast, for example:

```
ui = (unsigned char)sc;
```

(356) implicit conversion of float to integer (Parser)

A floating-point value has been assigned or otherwise converted to an integral type. This could result in truncation of the floating-point value. A typecast will make this warning go away.

```
double dd;
int i;
i = dd;    /* is this really what you meant? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)dd;
```

(357) illegal conversion of integer to pointer (Parser)

An integer has been assigned to, or otherwise converted to, a pointer type. This will usually mean that you have used the wrong variable. But if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This can also mean that you have forgotten the `&` address operator, for example:

```
int * ip;
int i;
ip = i;    /* oops -- did you mean ip = &i ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
ip = (int *)i;
```

(358) illegal conversion of pointer to integer (Parser)

A pointer has been assigned to, or otherwise converted to, a integral type. This will usually mean that you have used the wrong variable. But if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This can also mean that you have forgotten the `*` dereference operator, for example:

```
int * ip;
int i;
i = ip;    /* oops -- did you mean i = *ip ? */
```

If you do intend to use an expression like this, indicate your intention by a cast:

```
i = (int)ip;
```

(359) illegal conversion between pointer types (Parser)

A pointer of one type (i.e., pointing to a particular kind of object) has been converted into a pointer of a different type. This usually means that you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed, for example:

```
long input;
char * cp;
cp = &input;  /* is this correct? */
```

This is a common way of accessing bytes within a multi-byte variable. To indicate that this is the intended operation of the program, use a cast:

```
cp = (char *)&input;  /* that’s better */
```

This warning can also occur when converting between pointers to objects that have the same type, but which have different qualifiers, for example:

```
char * cp;
/* yes, but what sort of characters? */
cp = "I am a string of characters";
```

If the default type for string literals is `const char *`, then this warning is quite valid. This should be written:

```
const char * cp;
cp = "I am a string of characters";  /* that's better */
```

Omitting a qualifier from a pointer type is often disastrous and almost certainly not what you intend.

(360) array index out of bounds (Parser)
An array is being indexed with a constant value that is less than zero, or greater than or equal to the number of elements in the array. This warning will not be issued when accessing an array element via a pointer variable, for example:

```
int i, * ip, input[10];
i = input[-2];          /* oops -- this element doesn't exist */
ip = &input[5];
i = ip[-2];             /* this is okay */
```

(361) function declared implicit int (Parser)
When the compiler encounters a function call of a function whose name is presently undefined, the compiler will automatically declare the function to be of type int, with unspecified (K&R style) parameters. If a definition of the function is subsequently encountered, it is possible that its type and arguments will be different from the earlier implicit declaration, causing a compiler error. The solution is to ensure that all functions are defined (or at least declared) before use, preferably with prototyped parameters. If it is necessary to make a forward declaration of a function, it should be preceded with the keywords `extern` or `static`, as appropriate. For example:

```
/* I can prevent an error arising from calls below */
extern void set(long a, int b);
int main(void)
{
    /* at this point, a prototype for set() has already been seen */
    set(10L, 6);
}
```

(362) redundant “&” applied to array (Parser)
The address operator `&` has been applied to an array. Because using the name of an array gives its address anyway, this is unnecessary and has been ignored, for example:

```
int array[5];
int * ip;
/* array is a constant, not a variable; the & is redundant. */
ip = &array;
```

(363) redundant “&” or “*” applied to function address (Parser)
The address operator `&` has been applied to a function. Because using the name of a function gives its address anyway, this is unnecessary and has been ignored, for example:

```
extern void foo(void);
int main(void)
{
    void(*bar)(void);
    /* both assignments are equivalent */
    bar = &foo;
    bar = foo;  /* the & is redundant */
}
```

(364) attempt to modify object qualified * (Parser)
Objects declared `const` or `code` cannot be assigned to or modified in any other way by your program. The effect of attempting to modify such an object is compiler specific.

```
const int out = 1234;  /* "out" is read only */
out = 0;               /* oops -- writing to a read-only object */
```

(365) pointer to non-static object returned (Parser)
This function returns a pointer to a non-static (e.g., auto) variable. This is likely to be an error, because the storage associated with automatic variables becomes invalid when the function returns, for example:

```
char * get_addr(void)
{
    char c;
    /* returning this is dangerous; the pointer could be dereferenced */
    return &c;
}
```

(366) operands of “*” not same pointer type (Parser)
The operands of this operator are of different pointer types. This probably means you have used the wrong pointer, but if the code is actually what you intended, use a typecast to suppress the error message.

(367) identifier is already extern; can’t be static (Parser)
This function was already declared `extern`, possibly through an implicit declaration. It has now been redeclared `static`, but this redeclaration is invalid.

```
int main(void)
{
    /* at this point the compiler assumes set is extern... */
    set(10L, 6);
}
/* now it finds out otherwise */
static void set(long a, int b)
{
    PORTA = a + b;
}
```

(368) array dimension on “*[ ]” ignored (Preprocessor)
An array dimension on a function parameter has been ignored because the argument is actually converted to a pointer when passed. Thus arrays of any size can be passed. Either remove the dimension from the parameter, or define the parameter using pointer syntax, for example:

```
/* param should be: "int array[]" or "int *" */
int get_first(int array[10])
{                              /* warning flagged here */
    return array[0];
}
```

(369) signed bitfields not supported (Parser)
Only unsigned bit-fields are supported. If a bit-field is declared to be type `int`, the compiler still treats it as unsigned, for example:

```
struct {
    signed int sign: 1;    /* oops -- this must be unsigned */
    signed int value: 7;
} ;
```

(370) illegal basic type; int assumed (Parser)
The basic type of a cast to a qualified basic type could not be recognized and the basic type was assumed to be int, for example:

```
/* here ling is assumed to be int */
unsigned char bar = (unsigned ling) ’a’;
```

(371) missing basic type; int assumed (Parser)
This declaration does not include a basic type, so `int` has been assumed. This declaration is not illegal, but it is preferable to include a basic type to make it clear what is intended, for example:

```
char c;
i;      /* don't let the compiler make assumptions, use : int i */
func(); /* ditto, use: extern int func(int); */
```

(372) "," expected (Parser)
A comma was expected here. This could mean you have left out the comma between two identifiers in a declaration list. It can also mean that the immediately preceding type name is misspelled and has been interpreted as an identifier, for example:

```
unsigned char a;
/* thinks: chat & b are unsigned, but where is the comma? */
unsigned chat b;
```

(373) implicit signed to unsigned conversion (Parser)
An `unsigned` type was expected where a `signed` type was given and was implicitly converted to `unsigned`, for example:

```
unsigned int foo = -1;
/* the above initialization is implicitly treated as:
unsigned int foo = (unsigned) -1; */
```

(374) missing basic type; int assumed (Parser)
The basic type of a cast to a qualified basic type was missing and assumed to be `int`., for example:

```
int i = (signed) 2; /* (signed) assumed to be (signed int) */
```

(375) unknown FNREC type "*" (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(376) bad non-zero node in call graph (Linker)
The linker has encountered a top level node in the call graph that is referenced from lower down in the call graph. This probably means the program has indirect recursion, which is not allowed when using a compiled stack.

(378) can't create * file "*" (Hexmate)
This type of file could not be created. Is the file, or a file by this name, already in use?

(379) bad record type "*" (Linker)
This is an internal compiler error. Ensure that the object file is a valid object file. Contact Microchip Technical Support with details.

(380) unknown record type (*) (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(381) record "*" too long (*) (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(382) incomplete record: type = *, length = * (Dump, Xstrip)
This message is produced by the `dump` or `xstrip` utilities and indicates that the object file is not a valid object file, or that it has been truncated. Contact Microchip Technical Support with details.

(383) text record has length (*) too small (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(384) assertion failed: file *, line *, expression * (Linker, Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(387) illegal or too many -G options (Linker)
There has been more than one linker `-g` option, or the `-g` option did not have any arguments following. The arguments specify how the segment addresses are calculated.

(388) duplicate -M option (Linker)
The map file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of this option is present on the command line.

(389) illegal or too many -O options (Linker)
This linker `-o` flag is illegal, or another `-o` option has been encountered. A `-o` option to the linker must be immediately followed by a filename with no intervening space.

(390) missing argument to -P (Linker)
There have been too many `-p` options passed to the linker, or a `-p` option was not followed by any arguments. The arguments of separate `-p` options can be combined and separated by commas.

(391) missing argument to -Q (Linker)
The `-Q` linker option requires the machine type for an argument.

(392) missing argument to -U (Linker)
The `-U` (undefine) option needs an argument.

(393) missing argument to -W (Linker)
The `-W` option (listing width) needs a numeric argument.

(394) duplicate -D or -H option (Linker)
The symbol file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of either of these options is present on the command line.

(395) missing argument to -J (Linker)
The maximum number of errors before aborting must be specified following the `-j` linker option.

(397) usage: hlink [-options] files.obj files.lib (Linker)
Improper usage of the command-line linker. If you are not invoking the linker directly, this could be an internal compiler error, and you should contact Microchip Technical Support with details.

(398) output file can’t be also an input file (Linker)
The linker has detected an attempt to write its output file over one of its input files. This cannot be done, because it needs to simultaneously read and write input and output files.

(400) bad object code format (Linker)
This is an internal compiler error. The object code format of an object file is invalid. Ensure it is a valid object file. Contact Microchip Technical Support with details.

(402) bad argument to -F (Objtohex)
The `-F` option for objtohex has been supplied an invalid argument. If you are not invoking this tool directly, this is an internal compiler error, and you should contact Microchip Technical Support with details.

(403) bad -E option: “*” (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(404) bad maximum length value to -<digits> (Objtohex)
The first value to the OBJTOHEX `-n,m` HEX length/rounding option is invalid.

(405) bad record size rounding value to -<digits> (Objtohex)
The second value to the OBJTOHEX `-n,m` HEX length/rounding option is invalid.

(406) bad argument to -A (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(407) bad argument to -U (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(408) bad argument to -B (Objtohex)
This option requires an integer argument in either base 8, 10, or 16. If you are not invoking this tool directly, this is an internal compiler error and you should contact Microchip Technical Support with details.

(409) bad argument to -P (Objtohex)
This option requires an integer argument in either base 8, 10, or 16. If you are not invoking this tool directly, this is an internal compiler error and you should contact Microchip Technical Support with details.

(410) bad combination of options (Objtohex)
The combination of options supplied to OBJTOHEX is invalid.

(412) text does not start at 0 (Objtohex)
Code in some things must start at zero. Here it doesn’t.

(413) write error on “*” (Assembler, Linker, Cromwell)
A write error occurred on the named file. This probably means you have run out of disk space.

(414) read error on “*” (Linker)
The linker encountered an error trying to read this file.

(415) text offset too low in COFF file (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(416) bad character (*) in extended TEKHEX line (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(417) seek error in “*” (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(418) image too big (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(419) object file is not absolute (Objtohex)
The object file passed to OBJTOHEX has relocation items in it. This can indicate it is the wrong object file, or that the linker or OBJTOHEX have been given invalid options. The object output files from the assembler are relocatable, not absolute. The object file output of the linker is absolute.

(420) too many relocation items (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(421) too many segments (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(422) no end record (Linker)
This object file has no end record. This probably means it is not an object file. Contact Microchip Technical Support if the object file was generated by the compiler.

(423) illegal record type (Linker)
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Contact Microchip Technical Support with details if the object file was created by the compiler.

(424) record too long (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(425) incomplete record (Objtohex, Libr)
The object file passed to OBJTOHEX or the librarian is corrupted. Contact Microchip Technical Support with details.

(427) syntax error in list (Objtohex)
There is a syntax error in a list read by OBJTOHEX. The list is read from standard input in response to an option.

(428) too many segment fixups (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(429) bad segment fixups (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(430) bad specification (Objtohex)
A list supplied to OBJTOHEX is syntactically incorrect.

(431) bad argument to -E (Objtoexe)
This option requires an integer argument in either base 8, 10, or 16. If you are invoking `objtoexe` directly then check this argument. Otherwise, this can be an internal compiler error and you should contact Microchip Technical Support with details.

(432) usage: objtohex [-ssymfile] [object-file [exe-file]] (Objtohex)
Improper usage of the command-line tool `objtohex`. If you are not invoking this tool directly, this is an internal compiler error and you should contact Microchip Technical Support with details.

(434) too many symbols (*) (Linker)
There are too many symbols in the symbol table, which has a limit of * symbols. Change some global symbols to local symbols to reduce the number of symbols.

(435) bad segment selector "*" (Linker)
The segment specification option (`-G`) to the linker is invalid, for example:

```
-GA/f0+10
```

Did you forget the radix?

```
-GA/f0h+10
```

(436) psect “*” re-orged (Linker)
This psect has had its start address specified more than once.

(437) missing “=” in class spec (Linker)
A class spec needs an `=` sign, e.g., `-Ctext=ROM`.

(438) bad size in -S option (Linker)
The address given in a `-S` specification is invalid, it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O`, for octal, or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, for example:

```
-SCODE=f000
```

Did you forget the radix?

```
-SCODE=f000h
```

(439) bad -D spec: “*” (Linker)
The format of a `-D` specification, giving a delta value to a class, is invalid, for example:

```
-DCODE
```

What is the delta value for this class? Possibly, you meant something like:

```
-DCODE=2
```

(440) bad delta value in -D spec (Linker)
The delta value supplied to a `-D` specification is invalid. This value should an integer of base 8, 10, or 16.

(441) bad -A spec: “*” (Linker)
The format of a `-A` specification, giving address ranges to the linker, is invalid, for example:

```
-ACODE
```

What is the range for this class? Possibly, you meant:

```
-ACODE=0h-1fffh
```

(442) missing address in -A spec (Linker)
The format of a `-A` specification, giving address ranges to the linker, is invalid, for example:

```
-ACODE=
```

What is the range for this class? Possibly, you meant:

```
-ACODE=0h-1fffh
```

(443) bad low address “*” in -A spec (Linker)
The low address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O` (for octal) or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, for example:

```
-ACODE=1fff-3fffh
```

Did you forget the radix?

```
-ACODE=1fffh-3fffh
```

(444) expected "-" in -A spec (Linker)
There should be a minus sign, `-`, between the high and low addresses in a `-A` linker option, for example:

```
-AROM=1000h
```

Possibly, you meant:

```
-AROM=1000h-1fffh
```

(445) bad high address "*" in -A spec (Linker)
The high address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O`, for octal, or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, for example:

```
-ACODE=0h-ffff
```

Did you forget the radix?

```
-ACODE=0h-ffffh
```

(446) bad overrun address "*" in -A spec (Linker)
The overrun address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O` (for octal) or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, for example:

```
-AENTRY=0-0FFh-1FF
```

Did you forget the radix?

```
-AENTRY=0-0FFh-1FFh
```

(447) bad load address "*" in -A spec (Linker)
The load address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O` (for octal) or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, for example:

```
-ACODE=0h-3fffh/a000
```

Did you forget the radix?

```
-ACODE=0h-3fffh/a000h
```

(448) bad repeat count "*" in -A spec (Linker)
The repeat count given in a `-A` specification is invalid, for example:

```
-AENTRY=0-0FFhxf
```

Did you forget the radix?

```
-AENTRY=0-0FFhxfh
```

(449) syntax error in -A spec: * (Linker)
The `-A` spec is invalid. A valid `-A` spec should be something like:

```
-AROM=1000h-1FFFh
```

(450) psect "*" was never defined, or is local (Linker)
This psect has been listed in a `-P` option, but is not defined in any module within the program. Alternatively, the psect is defined using the `local` psect flag, but with no `class` flag; and, so, cannot be linked to an address. Check the assembly list file to ensure that the psect exists and that it is does not specify the local psect flag.

(451) bad psect origin format in -P option (Linker)
The origin format in a `-p` option is not a validly formed decimal, octal, or HEX number, nor is it the name of an existing psect. A HEX number must have a trailing `H`, for example:

```
-pbss=f000
```

Did you forget the radix?

```
-pbss=f000h
```

(452) bad "+" (minimum address) format in -P option (Linker)
The minimum address specification in the linker's `-p` option is badly formatted, for example:

```
-pbss=data+f000
```

Did you forget the radix?

```
-pbss=data+f000h
```

(453) missing number after "%" in -P option (Linker)
The `%` operator in a `-p` option (for rounding boundaries) must have a number after it.

(454) link and load address can't both be set to "." in -P option (Linker)
The link and load address of a psect have both been specified with a dot character. Only one of these addresses can be specified in this manner, for example:

```
-Pmypsect=1000h/.
-Pmypsect=./1000h
```

Both of these options are valid and equivalent. However, the following usage is ambiguous:

```
-Pmypsect=./.
```

What is the link or load address of this psect?

(455) psect "*" not relocated on 0x* byte boundary (Linker)
This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the `-p` option. if necessary.

(456) psect "*" not loaded on 0x* boundary (Linker)
This psect has a relocatability requirement that is not met by the load address given in a `-p` option. For example, if a psect must be on a 4K byte boundary, you could not start it at 100H.

(459) remove failed; error: *, * (Xstrip)
The creation of the output file failed when removing an intermediate file.

(460) rename failed; error: *, * (Xstrip)
The creation of the output file failed when renaming an intermediate file.

(461) can't create * file "*" (Assembler, Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(464) missing key in avmap file (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(465) undefined symbol “*” in FNBREAK record (Linker)
The linker has found an undefined symbol in the `FNBREAK` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(466) undefined symbol “*” in FNINDIR record (Linker)
The linker has found an undefined symbol in the `FNINDIR` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(467) undefined symbol “*” in FNADDR record (Linker)
The linker has found an undefined symbol in the `FNADDR` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(468) undefined symbol “*” in FNCALL record (Linker)
The linker has found an undefined symbol in the `FNCALL` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(469) undefined symbol “*” in FNROOT record (Linker)
The linker has found an undefined symbol in the `FNROOT` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(470) undefined symbol “*” in FNSIZE record (Linker)
The linker has found an undefined symbol in the `FNSIZE` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(471) recursive function calls: (Linker)
These functions (or function) call each other recursively. One or more of these functions has statically allocated local variables (compiled stack). Either use the `reentrant` keyword (if supported with this compiler) or recode to avoid recursion, for example:

```
int test(int a)
{
    if(a == 5) {
    /* recursion cannot be supported by some compilers */
        return test(a++);
     }
    return 0;
}
```

(472) non-reentrant function “*” appears in multiple call graphs: rooted at “*” and “*” (Linker)
This function can be called from both main-line code and interrupt code. Use the `reentrant` keyword, if this compiler supports it, or recode to avoid using local variables or parameters, or duplicate the function, for example:

```
void interrupt my_isr(void)
{
    scan(6);     /* scan is called from an interrupt function */
}
void process(int a)
{
    scan(a);     /* scan is also called from main-line code */
}
```

(473) function “*” is not called from specified interrupt_level (Linker)
The indicated function is never called from an interrupt function of the same interrupt level, for example:

```
#pragma interrupt_level 1
void foo(void)
{
    ...
}
#pragma interrupt_level 1
void interrupt bar(void)
{
    // this function never calls foo()
}
```

(474) no psect specified for function variable/argument allocation (Linker)
The `FNCONF` assembler directive which specifies to the linker information regarding the auto/parameter block was never seen. This is supplied in the standard runtime files if necessary. This error can imply that the correct run-time startup module was not linked. Ensure you have used the `FNCONF` directive if the runtime startup module is hand-written.

(475) conflicting FNCONF records (Linker)
The linker has seen two conflicting `FNCONF` directives. This directive should be specified only once and is included in the standard runtime startup code which is normally linked into every program.

(476) fixup overflow referencing * * (location 0x* (0x*+*), size *, value 0x*) (Linker)
The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. See the following error message (1356) for more information.

(477) fixup overflow in expression (location 0x* (0x*+*), size *, value 0x*) (Linker)
The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. See the following error message (1356) for more information.

(478) * range check failed (location 0x* (0x*+*), value 0x* > limit 0x*) (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(479) circular indirect definition of symbol “*” (Linker)
The specified symbol has been equated to an external symbol which, in turn, has been equated to the first symbol.

(480) function signatures do not match: * (*): 0x*/0x* (Linker)
The specified function has different signatures in different modules. This means it has been declared differently; i.e., it can have been prototyped in one module and not another. Check what declarations for the function are visible in the two modules specified and make sure they are compatible, for example:

```
extern int get_value(int in);
/* and in another module: */
/* this is different to the declaration */
int get_value(int in, char type)
{
```

(481) common symbol “*” psect conflict (Linker)
A common symbol has been defined to be in more than one psect.

(482) symbol “*” is defined more than once in “*” (Assembler)
This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, for example:

```
_next:
    MOVE r0, #55
```

```
    MOVE [r1], r0
_next:           ; oops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an underscore prepended to their name after compilation.

(483) symbol "*" can't be global (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(484) psect "*" can't be in classes "*" and "*" (Linker)
A psect cannot be in more than one class. This is either due to assembler modules with conflicting `class=` options to the `PSECT` directive, or use of the `-C` option to the linker, for example:

```
psect final,class=CODE
finish:
    /* elsewhere: */
psect final,class=ENTRY
```

(485) unknown "with" psect referenced by psect "*" (Linker)
The specified psect has been placed with a psect using the psect `with` flag. The psect it has been placed with does not exist, for example:

```
psect starttext,class=CODE,with=rext
; was that meant to be with text?
```

(486) psect "*" selector value redefined (Linker)
The `selector` value for this psect has been defined more than once.

(487) psect "*" type redefined: */* (Linker)
This psect has had its type defined differently by different modules. This probably means you are trying to link incompatible object modules, i.e., linking 386 flat model code with 8086 real mode code.

(488) psect "*" memory space redefined: */* (Linker)
A global psect has been defined in two different memory spaces. Either rename one of the psects or, if they are the same psect, place them in the same memory space using the space psect flag, for example:

```
psect spdata,class=RAM,space=0
    ds 6
; elsewhere:
psect spdata,class=RAM,space=1
```

(489) psect "*" memory delta redefined: */* (Linker)
A `global` psect has been defined with two different `delta` values, for example:

```
psect final,class=CODE,delta=2
finish:
; elsewhere:
psect final,class=CODE,delta=1
```

(490) class "*" memory space redefined: */* (Linker)
A class has been defined in two different memory spaces. Either rename one of the classes or, if they are the same class, place them in the same memory space.

(491) can't find 0x* words for psect "*" in segment "*" (Linker)
One of the main tasks the linker performs is positioning the blocks (or psects) of code and data that is generated from the program into the memory available for the target device. This error indicates that the linker was unable to find an area of free memory large enough to accommodate one of the psects. The error message indicates the name of the

psect that the linker was attempting to position and the segment name which is typically the name of a class which is defined by a `-A` linker option issued by the compiler driver. Typically psect names which are, or include, `text` relate to program code. Names such as `bss` or `data` refer to variable blocks. This error can be due to two reasons.

First, the size of the program or the program's data has exceeded the total amount of space on the selected device. In other words, some part of your device's memory has completely filled. If this is the case, then the size of the specified psect must be reduced.

The second cause of this message is when the total amount of memory needed by the psect being positioned is sufficient, but that this memory is fragmented in such a way that the largest contiguous block is too small to accommodate the psect. The linker is unable to split psects in this situation. That is, the linker cannot place part of a psect at one location and part somewhere else. Thus, the linker must be able to find a contiguous block of memory large enough for every psect. If this is the cause of the error, then the psect must be split into smaller psects if possible.

To find out what memory is still available, generate and look in the map file. Search for the string **UNUSED ADDRESS RANGES**. Under this heading, look for the name of the segment specified in the error message. If the name is not present, then all the memory available for this psect has been allocated. If it is present, there will be one address range specified under this segment for each free block of memory. Determine the size of each block and compare this with the number of words specified in the error message.

Psects containing code can be reduced by using all the compiler's optimizations, or restructuring the program. If a code psect must be split into two or more small psects, this requires splitting a function into two or more smaller functions (which can call each other). These functions can need to be placed in new modules.

Psects containing data can be reduced when invoking the compiler optimizations, but the effect is less dramatic. The program can need to be rewritten so that it needs less variables. If the default linker options must be changed, this can be done indirectly through the driver using the driver `-Wl,` option. If a data psect cannot be positioned, then you typically need to reduce the total size of variables being used.

For example, after receiving the message:

```
Can't find 0x34 words (0x34 withtotal) for psect text in segment CODE (error)
```

look in the map file for the ranges of unused memory.

```
UNUSED ADDRESS RANGES
CODE             00000244-0000025F
                 00001000-0000102f
RAM              00300014-00301FFB
```

In the `CODE` segment, there is 0x1c (0x25f-0x244+1) bytes of space available in one block and 0x30 available in another block. Neither of these are large enough to accommodate the psect `text` which is 0x34 bytes long. Notice that the total amount of memory available is larger than 0x34 bytes. If the function that is encoded into the `text` psect can be split into two smaller functions, there is a chance the program will link correctly.

(492) attempt to position absolute psect "*" is illegal (Linker)
This psect is absolute and should not have an address specified in a `-P` option. Either remove the `abs` psect flag, or remove the `-P` linker option.

(493) origin of psect "*" is defined more than once (Linker)
The origin of this psect is defined more than once. There is most likely more than one `-p` linker option specifying this psect.

(494) bad -P format "*/*" (Linker)
The `-P` option given to the linker is malformed. This option specifies placement of a psect, for example:

```
-Ptext=10g0h
```

Possibly, you meant:

```
-Ptext=10f0h
```

(495) use of both “with=” and “INCLASS/INCLASS” allocation is illegal (Linker)
It is not legal to specify both the link and location of a psect as within a class, when that psect was also defined using a with psect flag.

(497) psect “*” exceeds max size: *h > *h (Linker)
The psect has more bytes in it than the maximum allowed as specified using the `size` psect flag.

(498) psect “*” exceeds address limit: *h > *h (Linker)
The maximum address of the psect exceeds the limit placed on it using the `limit` psect flag. Either the psect needs to be linked at a different location or there is too much code/data in the psect.

(499) undefined symbol: (Assembler, Linker)
The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

10.2 Messages 500 Thru 999

(500) undefined symbols: (Linker)
A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(501) program entry point is defined more than once (Linker)
There is more than one entry point defined in the object files given the linker. End entry point is specified after the `END` directive. The runtime startup code defines the entry point, for example:

```
powerup:
goto start
END  powerup  ; end of file and define entry point
; other files that use END should not define another entry point
```

(502) incomplete * record body: length = * (Linker)
An object file contained a record with an illegal size. This probably means the file is truncated or not an object file. Contact Microchip Technical Support with details.

(503) ident records do not match (Linker)
The object files passed to the linker do not have matching `ident` records. This means they are for different device types.

(504) object code version is greater than *.* (Linker)
The object code version of an object module is higher than the highest version the linker is known to work with. Check that you are using the correct linker. Contact Microchip Technical Support if you have not patched the linker.

(505) no end record found inobject file (Linker)
An object file did not contain an end record. This probably means the file is corrupted or not an object file. Contact Microchip Technical Support if the object file was generated by the compiler.

(506) object file record too long: *+* (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(507) unexpected end of file in object file (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(508) relocation offset (*) out of range 0..*-*-1 (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(509) illegal relocation size: * (Linker)
There is an error in the object code format read by the linker. This either means you are using a linker that is out of date, or that there is an internal error in the assembler or linker. Contact Microchip Technical Support with details if the object file was created by the compiler.

(510) complex relocation not supported for -R or -L options (Linker)
The linker was given a `-R` or `-L` option with file that contain complex relocation.

(511) bad complex range check (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(512) unknown complex operator 0x* (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(513) bad complex relocation (Linker)
The linker has been asked to perform complex relocation that is not syntactically correct. Probably means an object file is corrupted.

(514) illegal relocation type: * (Linker)
An object file contained a relocation record with an illegal relocation type. This probably means the file is corrupted or not an object file. Contact Microchip Technical Support with details if the object file was created by the compiler.

(515) unknown symbol type * (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(516) text record has bad length: *-*-(*+1) < 0 (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(520) function “*” is never called (Linker)
This function is never called. This cannot represent a problem, but space could be saved by removing it. If you believe this function should be called, check your source code. Some assembler library routines are never called, although they are actually execute. In this case, the routines are linked in a special sequence so that program execution falls through from one routine to the next.

(521) call depth exceeded by function “*” (Linker)
The call graph shows that functions are nested to a depth greater than specified.

(522) library “*” is badly ordered (Linker)
This library is badly ordered. It will still link correctly, but it will link faster if better ordered.

(523) argument to -W option (*) illegal and ignored (Linker)
The argument to the linker option `-w` is out of range. This option controls two features. For warning levels, the range is -9 to 9. For the map file width, the range is greater than or equal to 10.

(524) unable to open list file "*": * (Linker)
The named list file could not be opened. The linker would be trying to fixup the list file so that it will contain absolute addresses. Ensure that an assembler list file was generated during the compilation stage. Alternatively, remove the assembler list file generation option from the link step.

(525) too many address (memory) spaces; space (*) ignored (Linker)
The limit to the number of address spaces (specified with the `PSECT` assembler directive) is currently 16.

(526) psect "*" not specified in -P option (first appears in "*") (Linker)
This psect was not specified in a `-P` or `-A` option to the linker. It has been linked at the end of the program, which is probably not where you wanted it.

(528) no start record; entry point defaults to zero (Linker)
None of the object files passed to the linker contained a start record. The start address of the program has been set to zero. This can be harmless, but it is recommended that you define a start address in your startup module by using the `END` directive.

(529) usage: objtohex [-Ssymfile] [object-file [HEX-file]] (Objtohex)
Improper usage of the command-line tool `objtohex`. If you are not invoking this tool directly, this is an internal compiler error, and you should contact Microchip Technical Support with details.

(593) can't find 0x* words (0x* withtotal) for psect "*" in segment "*" (Linker)
See message (491).

(594) undefined symbol: (Linker)
The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

(595) undefined symbols: (Linker)
A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(596) segment "*" (*-*) overlaps segment "*" (*-*) (Linker)
The named segments have overlapping code or data. Check the addresses being assigned by the `-P` linker option.

(599) No psect classes given for COFF write (Cromwell)
CROMWELL requires that the program memory psect classes be specified to produce a COFF file. Ensure that you are using the `-N` option.

(600) No chip arch given for COFF write (Cromwell)
CROMWELL requires that the chip architecture be specified to produce a COFF file. Ensure that you are using the `-P` option.

(601) Unknown chip arch "*" for COFF write (Cromwell)
The chip architecture specified for producing a COFF file isn't recognized by
CROMWELL. Ensure that you are using the `-P` option, and that the architecture is correctly specified.

(602) null file format name (Cromwell)
The `-I` or `-O` option to CROMWELL must specify a file format.

(603) ambiguous file format name "*" (Cromwell)
The input or output format specified to CROMWELL is ambiguous. These formats are specified with the `-i` key and `-o` key options respectively.

(604) unknown file format name "*" (Cromwell)
The output format specified to CROMWELL is unknown, for example:

```
cromwell -m -P16F877 main.HEX main.sym -ocot
```

There is no output file type of cot. Did you mean cof?

(605) did not recognize format of input file (Cromwell)
The input file to CROMWELL is required to have a Cromwell map file (CMF), COD, Intel HEX, Motorola HEX, COFF, OMF51, ELF, UBROF or HI-TECH format.

(606) inconsistent symbol tables (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(607) inconsistent line number tables (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(608) bad path specification (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(609) missing device spec after -P (Cromwell)
The `-p` option to CROMWELL must specify a device name.

(610) missing psect classes after -N (Cromwell)
CROMWELL requires that the `-N` option be given a list of the names of psect classes.

(611) too many input files (Cromwell)
To many input files have been specified to be converted by CROMWELL.

(612) too many output files (Cromwell)
To many output file formats have been specified to CROMWELL.

(613) no output file format specified (Cromwell)
The output format must be specified to CROMWELL.

(614) no input files specified (Cromwell)
CROMWELL must have an input file to convert.

(616) option -Cbaseaddr is illegal with options -R or -L (Linker)
The linker option `-C`*`baseaddr`* cannot be used in conjunction with either the `-R` or `-L` linker options.

(618) error reading COD file data (Cromwell)
An error occurred reading the input COD file. Confirm the spelling and path of the file specified on the command line.

(619) I/O error reading symbol table (Cromwell)
The COD file has an invalid format in the specified record.

(620) filename index out of range in line number record (Cromwell)
The COD file has an invalid value in the specified record.

(621) error writing ELF/DWARF section "*" on "*" (Cromwell)
An error occurred writing the indicated section to the given file. Confirm the spelling and path of the file specified on the command line.

(622) too many type entries (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(623) bad class in type hashing (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(624) bad class in type compare (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(625) too many files in COFF file (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(626) string lookup failed in COFF: get_string() (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(627) missing "*" in SDB file "*" line * column * (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(629) bad storage class "*" in SDB file "*" line * column * (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(630) invalid syntax for prefix list in SDB file "*" (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(631) syntax error at token "*" in SDB file "*" line * column * (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(632) can't handle address size (*) (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(633) unknown symbol class (*) (Cromwell)
CROMWELL has encountered a symbol class in the symbol table of a COFF, Microchip COFF, or ICOFF file which it cannot identify.

(634) error dumping "*" (Cromwell)
Either the input file to CROMWELL is of an unsupported type or that file cannot be dumped to the screen.

(635) invalid HEX file "*" on line * (Cromwell)
The specified HEX file contains an invalid line. Contact Microchip Technical Support if the HEX file was generated by the compiler.

(636) error in Intel HEX file "*" on line * (Cromwell, Hexmate)
An error was found at the specified line in the specified Intel HEX file. The HEX file may be corrupt.

(637) unknown prefix "*" in SDB file "*" (Cromwell)
This is an internal compiler warning. Contact Microchip Technical Support with details.

(638) version mismatch: 0x* expected (Cromwell)
The input Microchip COFF file wasn't produced using CROMWELL.

(639) zero bit width in Microchip optional header (Cromwell)
The optional header in the input Microchip COFF file indicates that the program or data memory spaces are zero bits wide.

(668) prefix list did not match any SDB types (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(669) prefix list matched more than one SDB type (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(670) bad argument to -T (Clist)
The argument to the `-T` option to specify tab size was not present or correctly formed. The option expects a decimal integer argument.

(671) argument to -T should be in range 1 to 64 (Clist)
The argument to the `-T` option to specify tab size was not in the expected range. The option expects a decimal integer argument ranging from 1 to 64 inclusive.

(673) missing filename after * option (Objtohex)
The indicated option requires a valid file name. Ensure that the filename argument supplied to this option exists and is spelled correctly.

(674) too many references to "*" (Cref)
This is an internal compiler error. Contact Microchip Technical Support with details.

(679) unknown extraspecial: * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(680) bad format for -P option (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(681) bad common spec in -P option (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(685) bad putwsize() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(686) bad switch size (*) (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(687) bad pushreg "*" (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(688) bad popreg "*" (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(689) unknown predicate "*" (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(691) interrupt functions not implemented for 12 bit PIC MCU (Code Generator)
The 12-bit (Baseline) range of PIC MCU processors do not support interrupts.

(692) more than one interrupt level is associated with the interrupt function "*" (Code Generator)

Only one interrupt level can be associated with an interrupt function. Check to ensure that only one `interrupt_level` pragma has been used with the function specified. This pragma can be used more than once on main-line functions that are called from interrupt functions. For example:

```
#pragma interrupt_level 0
#pragma interrupt_level 1  /* oops -- which is it to be: 0 or 1? */
void interrupt isr(void)
{
```

(693) 0 (default) or 1 are the only acceptable interrupt levels for this function (Code Generator)

The only possible interrupt levels are 0 or 1. Check to ensure that all `interrupt_level` pragmas use these levels.

```
#pragma interrupt_level 2  /* oops -- only 0 or 1 */
void interrupt isr(void)
{
    /* isr code goes here */
}
```

(694) no interrupt strategy available (Code Generator)

The device does not support saving and subsequent restoring of registers during an interrupt service routine.

(695) duplicate case label (*) (Code Generator)

There are two `case` labels with the same value in this `switch` statement, for example:

```
switch(in) {
case '0':  /* if this is case '0'... */
    b++;
    break;
case '0':  /* then what is this case? */
    b--;
    break;
}
```

(696) out-of-range case label (*) (Code Generator)

This case label is not a value that the controlling expression can yield, thus this label will never be selected.

(697) non-constant case label (Code Generator)

A `case` label in this switch statement has a value which is not a constant.

(698) bit variables must be global or static (Code Generator)

A `__bit` variable cannot be of type auto. If you require a bit variable with scope local to a block of code or function, qualify it static, for example:

```
bit proc(int a)
{
    bit bb;        /* oops --  this should be: static bit bb; */
    bb = (a > 66);
    return bb;
}
```

(699) no case labels in switch (Code Generator)

There are no `case` labels in this `switch` statement, for example:

```
switch(input) {
}               /* there is nothing to match the value of input */
```

(700) truncation of enumerated value (Code Generator)
An enumerated value larger than the maximum value supported by this compiler was detected and has been truncated, for example:

```
enum { ZERO, ONE, BIG=0x99999999 } test_case;
```

(701) unreasonable matching depth (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(702) regused(): bad arg to G (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(703) bad GN (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(704) bad RET_MASK (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(705) bad which (*) after I (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(706) bad which in expand() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(707) bad SX (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(708) bad mod "+" for how = "*" (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(709) metaregister "*" can't be used directly (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(710) bad U usage (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(711) bad how in expand() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(712) can't generate code for this expression (Code Generator)
This error indicates that a C expression is too difficult for the code generator to actually compile. For successful code generation, the code generator must know how to compile an expression and there must be enough resources (i.e., registers or temporary memory locations) available. Simplifying the expression, i.e., using a temporary variable to hold an intermediate result, can often bypass this situation.

This error can also be issued if the code being compiled is unusual. For example, code which writes to a `const`-qualified object is illegal and will result in warning messages, but the code generator can unsuccessfully try to produce code to perform the write.

This error can also result from an attempt to redefine a function that uses the `intrinsic` pragma.

(713) bad initialization list (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(714) bad intermediate code (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(715) bad pragma "*" (Code Generator)
The code generator has been passed a `pragma` directive that it does not understand. This implies that the `pragma` you have used is not implemented for the target device.

(716) bad argument to -M option "*" (Code Generator)
The code generator has been passed a `-M` option that it does not understand. This should not happen if it is being invoked by a standard compiler driver.

(718) incompatible intermediate code version; should be *.* (Code Generator)
The intermediate code file produced by P1 is not the correct version for use with this code generator. This is either that incompatible versions of one or more compilers have been installed in the same directory, or a temporary file error has occurred leading to corruption of a temporary file. Check the setting of the `TEMP` environment variable. If it refers to a long path name, change it to something shorter. Contact Microchip Technical Support with details if required.

(720) multiple free: * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(721) element count must be constant expression (Code Generator)
The expression that determines the number of elements in an array must be a constant expression. Variables qualified as `const` do not form such an expression.

```
const unsigned char mCount = 5;
int mDeadtimeArr[mCount];  // oops -- the size cannot be a variable
```

(722) bad variable syntax in intermediate code (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(723) function definitions nested too deep (Code Generator)
This error is unlikely to happen with C code, because C cannot have nested functions! Contact Microchip Technical Support with details.

(724) bad op (*) in revlog() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(726) bad op "*" in uconval() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(727) bad op "*" in bconfloat() (Code Generator)
This is an internal code generator error. Contact Microchip Technical Support with details.

(728) bad op "*" in confloat() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(729) bad op "*" in conval() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(730) bad op "*" (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(731) expression error with reserved word (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(732) initialization of bit types is illegal (Code Generator)
Variables of type `__bit` cannot be initialized, for example:

```
__bit b1 = 1; /* oops!  b1 must be assigned after its definition */
```

(733) bad string “*” in pragma “psect” (Code Generator)
The code generator has been passed a `#pragma psect` directive that has a badly formed string, for example:

```
#pragma psect text  /* redirect text psect into what? */
```

Possibly, you meant something like:

```
#pragma psect text=special_text
```

(734) too many “psect” pragmas (Code Generator)
Too many `#pragma psect` directives have been used.

(735) bad string “*” in pragma “stack_size” (Code Generator)
The argument to the `stack_size` pragma is malformed. This pragma must be followed by a number representing the maximum allowed stack size.

(737) unknown argument “*” to pragma “switch” (Code Generator)
The `#pragma` switch directive has been used with an invalid `switch` code generation method. Possible arguments are: `auto`, `simple` and `direct`.

(739) error closing output file (Code Generator)
The compiler detected an error when closing a file. Contact Microchip Technical Support with details.

(740) zero dimension array is illegal (Code Generator)
The code generator has been passed a declaration that results in an array having a zero dimension.

(741) bitfield too large (* bits) (Code Generator)
The maximum number of bits in a bit-field is 8, the same size as the storage unit width.

```
struct {
    unsigned flag  : 1;
    unsigned value : 12;  /* oops -- that’s larger than 8 bits wide */
    unsigned cont  : 6;
} object;
```

(742) function “*” argument evaluation overlapped (Linker)
A function call involves arguments which overlap between two functions. This could occur with a call like:

```
void fn1(void)
{
    fn3( 7, fn2(3), fn2(9));    /* Offending call */
}
char fn2(char fred)
{
    return fred + fn3(5,1,0);
}
char fn3(char one, char two, char three)
{
    return one+two+three;
}
```

where fn1 is calling fn3 , and two arguments are evaluated by calling fn2 , which in turn calls fn3. The program structure should be modified to prevent this type of call sequence.

(743) divide by zero (Code Generator)
An expression involving a division by zero has been detected in your code.

(744) static object "*" has zero size (Code Generator)
A static object has been declared, but has a size of zero.

(745) nodecount = * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(746) object "*" qualified const but not initialized (Code Generator)
An object has been qualified as const, but there is no initial value supplied at the definition. As this object cannot be written by the C program, this can imply the initial value was accidentally omitted.

(747) unrecognized option "*" to -Z (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(748) variable "*" possibly used before being assigned a value (Code Generator)
This variable has possibly been used before it was assigned a value. Because it is an auto variable, this will result in it having an unpredictable value, for example:

```
int main(void)
{
    int a;
    if(a)         /* oops -- 'a' has never been assigned a value */
        process();
}
```

(749) unknown register name "*" used with pragma (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(750) constant operand to || or && (Code Generator)
One operand to the logical operators || or && is a constant. Check the expression for missing or badly placed parentheses. This message can also occur if the global optimizer is enabled and one of the operands is an auto or static local variable whose value has been tracked by the code generator, for example:

```
{
    int a;
    a = 6;
    if(a || b)  /* a is 6, therefore this is always true */
        b++;
```

(751) arithmetic overflow in constant expression (Code Generator)
A constant expression has been evaluated by the code generator that has resulted in a value that is too big for the type of the expression. The most common code to trigger this warning is assignments to signed data types. For example:

```
signed char c;
c = 0xFF;
```

As a signed 8-bit quantity, c can only be assigned values -128 to 127. The constant is equal to 255 and is outside this range. If you mean to set all bits in this variable, then use either of:

```
c = ~0x0;
c = -1;
```

which sets all the bits in the variable, regardless of variable size and without warning.

This warning can also be triggered by intermediate values overflowing. For example:

```
unsigned int i;  /* assume ints are 16 bits wide */
i = 240 * 137;   /* this should be okay, right? */
```

A quick check with your calculator reveals that 240 * 137 is 32880 which can easily be stored in an `unsigned int`, but a warning is produced. Why? Because 240 and 137 and both `signed int` values. Therefore the result of the multiplication must also be a `signed int` value, but a `signed int` cannot hold the value 32880. Both operands are constant values so the code generator can evaluate this expression at compile time, but it must do so following all the ANSI C rules. The following code forces the multiplication to be performed with an unsigned result:

```
i = 240u * 137;  /* force at least one operand
to be unsigned */
```

(752) conversion to shorter data type (Code Generator)
Truncation can occur in this expression as the lvalue is of shorter type than the rvalue, for example:

```
char a;
int b, c;
a = b + c;  /* int to char conversion can result in truncation */
```

(753) undefined shift (* bits) (Code Generator)
An attempt has been made to shift a value by a number of bits equal to or greater than the number of bits in the data type. This will produce an undefined result on many processors. This is non-portable code and is flagged as having undefined results by the C Standard, for example:

```
int input;
input <<= 33;  /* oops -- that shifts the entire value out */
```

(754) bitfield comparison out of range (Code Generator)
This is the result of comparing a bit-field with a value when the value is out of range of the bit-field. That is, comparing a 2-bit bit-field to the value 5 will never be true as a 2-bit bit-field has a range from 0 to 3. For example:

```
struct {
    unsigned mask : 2;  /* mask can hold values 0 to 3 */
} value;
int compare(void)
{
    return (value.mask == 6);  /* test can
}
```

(755) divide by zero (Code Generator)
A constant expression that was being evaluated involved a division by zero, for example:

```
a /= 0;  /* divide by 0: was this what you were intending */
```

(757) constant conditional branch (Code Generator)
A conditional branch (generated by an `if`, `for`, `while` statement etc.) always follows the same path. This will be some sort of comparison involving a variable and a constant expression. For the code generator to issue this message, the variable must have local scope (either auto or static local) and the global optimizer must be enabled, possibly at higher level than 1, and the warning level threshold can need to be lower than the default level of 0.

The global optimizer keeps track of the contents of local variables for as long as is possible during a function. For C code that compares these variables to constants, the result of the comparison can be deduced at compile time and the output code hard coded to avoid the comparison, for example:

```
{
int a, b;
a = 5;
/* this can never be false; always perform the true statement */
```

```
if(a == 5)
    b = 6;
```

will produce code that sets a to 5, then immediately sets b to 6.

No code will be produced for the comparison `if(a == 5)`. If a was a global variable, it can be that other functions (particularly interrupt functions) can modify it and so tracking the variable cannot be performed.

This warning can indicate more than an optimization made by the compiler. It can indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected.

This warning can also be issued because you have written something like `while(1)`. To produce an infinite loop, use `for(;;)`.

A similar situation arises with for loops, for example:

```
{
int a, b;
/* this loop must iterate at least once */
for(a=0; a!=10; a++)
    b = func(a);
```

In this case the code generator can again pick up that a is assigned the value 0, then immediately checked to see if it is equal to 10. Because a is modified during the for loop, the comparison code cannot be removed, but the code generator will adjust the code so that the comparison is not performed on the first pass of the loop; only on the subsequent passes. This cannot reduce code size, but it will speed program execution.

(758) constant conditional branch: possible use of “=” instead of “==” (Code Generator)
There is an expression inside an `if` or other conditional construct, where a constant is being assigned to a variable. This can mean you have inadvertently used an assignment `=` instead of a compare `==`, for example:

```
int a, b;
/* this can never be false; always perform the true statement */
if(a = 4)
    b = 6;
```

will assign the value 4 to `a`, then , as the value of the assignment is always true, the comparison can be omitted and the assignment to b always made. Did you mean:

```
/* this can never be false;
always perform the true statement */
if(a == 4)
    b = 6;
```

which checks to see if a is equal to 4.

(759) expression generates no code (Code Generator)
This expression generates no output code. Check for things like leaving off the parentheses in a function call, for example:

```
int fred;
fred;     /* this is valid, but has no effect at all */
```

Some devices require that special function register need to be read to clear hardware flags. To accommodate this, in some instances the code generator does produce code for a statement which only consists of a variable ID. This can happen for variables which are qualified as volatile. Typically the output code will read the variable, but not do anything with the value read.

(760) portion of expression has no effect (Code Generator)
Part of this expression has no side effects and no effect on the value of the expression, for example:

```
int a, b, c;
a = b,c;  /* "b" has no effect, was that meant to be a comma? */
```

(761) size of yields 0 (Code Generator)
The code generator has taken the size of an object and found it to be zero. This almost certainly indicates an error in your declaration of a pointer; i.e., you can have declared a pointer to a zero length array. In general, pointers to arrays are of little use. If you require a pointer to an array of objects of unknown length, you only need a pointer to a single object that can then be indexed or incremented.

(762) constant truncated when assigned to bitfield (Code Generator)
A constant value is too large for a bit-field structure member to which it is being assigned, for example:

```
struct INPUT {
    unsigned a : 3;
    unsigned b : 5;
} input_grp;
input_grp.a = 0x12;  /* oops -- 0x12 cannot fit into a 3-bit wide object */
```

(763) constant left operand to “? :” operator (Code Generator)
The left operand to a conditional operator `?` is constant, thus the result of the tertiary operator `? :` will always be the same, for example:

```
a = 8 ? b : c;  /* this is the same as saying a = b; */
```

(764) mismatched comparison (Code Generator)
A comparison is being made between a variable or expression and a constant value which is not in the range of possible values for that expression, for example:

```
unsigned char c;
if(c > 300)      /* oops -- how can this be true? */
    close();
```

(765) degenerate unsigned comparison (Code Generator)
There is a comparison of an unsigned value with zero, which will always be true or false, for example:

```
unsigned char c;
if(c >= 0)
    ...
```

will always be true, because an unsigned value can never be less than zero.

(766) degenerate signed comparison (Code Generator)
There is a comparison of a signed value with the most negative value possible for this type, such that the comparison will always be true or false, for example:

```
char c;
if(c >= -128)
    ...
```

will always be true, because an 8 bit signed char has a maximum negative value of -128.

(767) constant truncated to bitfield width (Code Generator)
A constant value is too large for a bit-field structure member on which it is operating, for example:

```
struct INPUT {
    unsigned a : 3;
    unsigned b : 5;
} input_grp;
input_grp.a |= 0x13;   /* oops -- 0x13 to large for 3-bit wide object */
```

(768) constant relational expression (Code Generator)
There is a relational expression that will always be true or false. This, for example, can be the result of comparing an unsigned number with a negative value; or comparing a variable with a value greater than the largest number it can represent, for example:

```
unsigned int a;
if(a == -10)    /* if a is unsigned, how can it be -10? */
b = 9;
```

(769) no space for macro definition (Assembler)
The assembler has run out of memory.

(772) include files nested too deep (Assembler)
Macro expansions and include file handling have filled up the assembler's internal stack. The maximum number of open macros and include files is 30.

(773) macro expansions nested too deep (Assembler)
Macro expansions in the assembler are nested too deep. The limit is 30 macros and include files nested at one time.

(774) too many macro parameters (Assembler)
There are too many macro parameters on this macro definition.

(776) can't allocate space for object "*" (offs: *) (Assembler)
The assembler has run out of memory.

(777) can't allocate space for opnd structure within object "*" (offs: *) (Assembler)
The assembler has run out of memory.

(780) too many psects defined (Assembler)
There are too many psects defined! Boy, what a program!

(781) can't enter abs psect (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(782) REMSYM error (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(783) "with" psects are cyclic (Assembler)
If Psect A is to be placed "with" Psect B, and Psect B is to be placed "with" Psect A, there is no hierarchy. The `with` flag is an attribute of a psect and indicates that this psect must be placed in the same memory page as the specified psect.

Remove a with flag from one of the psect declarations. Such an assembler declaration can look like:

```
psect my_text,local,class=CODE,with=basecode
```

which will define a psect called `my_text` and place this in the same page as the psect basecode.

(784) overfreed (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(785) too many temporary labels (Assembler)
There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

(787) can't handle "v_rtype" of * in copyexpr (Assembler)

This is an internal compiler error. Contact Microchip Technical Support with details.

(788) invalid character "*" in number (Assembler)

A number contained a character that was not part of the range 0-9 or 0-F.

(790) end of file inside conditional (Assembler)

END-of-FILE was encountered while scanning for an `endif` to match a previous `if`.

(793) unterminated macro argument (Assembler)

An argument to a macro is not terminated. Note that angle brackets, `< >`, are used to quote macro arguments.

(794) invalid number syntax (Assembler)

The syntax of a number is invalid. This, for example, can be use of 8 or 9 in an octal number, or other malformed numbers.

(796) use of LOCAL outside macros is illegal (Assembler)

The `LOCAL` directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

(797) syntax error in LOCAL argument (Assembler)

A symbol defined using the `LOCAL` assembler directive in an assembler macro is syntactically incorrect. Ensure that all symbols and all other assembler identifiers conform with the assembly language of the target device.

(798) use of macro arguments in a LOCAL directive is illegal (Assembler)

The list of labels after the directive `LOCAL` cannot include any of the formal parameters to an enclosing macro, for example:

```
mmm MACRO a1
MOVE        r0, #a1
LOCAL        a1    ; oops -- the parameter cannot be used with LOCAL
ENDM
```

(799) REPT argument must be >= 0 (Assembler)

The argument to a `REPT` directive must be greater than zero, for example:

```
REPT -2                        ; -2 copies of this code? */
MOVE            r0, [r1]++
ENDM
```

(800) undefined symbol "*" (Assembler)

The named symbol is not defined in this module and has not been specified `GLOBAL`.

(801) range check too complex (Assembler)

This is an internal compiler error. Contact Microchip Technical Support with details.

(802) invalid address after END directive (Assembler)

The start address of the program which is specified after the assembler `END` directive must be a label in the current file.

(803) undefined temporary label (Assembler)

A temporary label has been referenced that is not defined. Note that a temporary label must have a number >= 0.

(804) write error on object file (Assembler)
The assembler failed to write to an object file. This can be an internal compiler error. Contact Microchip Technical Support with details.

(806) attempted to get an undefined object (*) (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(807) attempted to set an undefined object (*) (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(808) bad size in add_reloc() (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(809) unknown addressing mode (*) (Assembler)
An unknown addressing mode was used in the assembly file.

(811) “cnt” too large (*) in display() (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(814) device type not defined (Assembler)
The device must be defined either from the command line (e.g., `-16c84`), via the device assembler directive, or via the `LIST` assembler directive.

(815) syntax error in chipinfo file at line * (Assembler)
The chipinfo file contains non-standard syntax at the specified line.

(816) duplicate ARCH specification in chipinfo file “*” at line * (Assembler, Driver)
The chipinfo file has a device section with multiple `ARCH` values. Only one `ARCH` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(817) unknown architecture in chipinfo file at line * (Assembler, Driver)
An chip architecture (family) that is unknown was encountered when reading the chip INI file.

(818) duplicate BANKS for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `BANKS` values. Only one `BANKS` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(819) duplicate ZEROREG for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `ZEROREG` values. Only one `ZEROREG` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(820) duplicate SPAREBIT for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `SPAREBIT` values. Only one `SPAREBIT` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(821) duplicate INTSAVE for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `INTSAVE` values. Only one `INTSAVE` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(822) duplicate ROMSIZE for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `ROMSIZE` values. Only one `ROMSIZE` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(823) duplicate START for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `START` values. Only one `START` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(824) duplicate LIB for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `LIB` values. Only one `LIB` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(825) too many RAMBANK lines in chipinfo file for “*” (Assembler)
The chipinfo file contains a device section with too many `RAMBANK` fields. Reduce the number of values.

(826) inverted ram bank in chipinfo file at line * (Assembler, Driver)
The second HEX number specified in the `RAM` field in the chipinfo file must be greater in value than the first.

(827) too many COMMON lines in chipinfo file for “*” (Assembler)
There are too many lines specifying common (access bank) memory in the chip configuration file.

(828) inverted common bank in chipinfo file at line * (Assembler, Driver)
The second HEX number specified in the `COMMON` field in the chipinfo file must be greater in value than the first. Contact Microchip Technical Support if you have not modified the chipinfo INI file.

(829) unrecognized line in chipinfo file at line * (Assembler)
The chipinfo file contains a device section with an unrecognized line. Contact Microchip Technical Support if the INI has not been edited.

(830) missing ARCH specification for “*” in chipinfo file (Assembler)
The chipinfo file has a device section without an `ARCH` values. The architecture of the device must be specified. Contact Microchip Technical Support if the chipinfo file has not been modified.

(832) empty chip info file “*” (Assembler)
The chipinfo file contains no data. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(833) no valid entries in chipinfo file (Assembler)
The chipinfo file contains no valid device descriptions.

(834) page width must be >= 60 (Assembler)
The listing page width must be at least 60 characters. Any less will not allow a properly formatted listing to be produced, for example:

```
LIST C=10  ; the page width will need to be wider than this
```

(835) form length must be >= 15 (Assembler)
The form length specified using the `-F` length option must be at least 15 lines. Setting this length to zero is allowed and turns off paging altogether. The default value is zero (pageless).

(836) no file arguments (Assembler)
The assembler has been invoked without any file arguments. It cannot assemble anything.

(839) relocation too complex (Assembler)
The complex relocation in this expression is too big to be inserted into the object file.

(840) phase error (Assembler)
The assembler has calculated a different value for a symbol on two different passes. This is commonly due to the redefinition of a psect with conflicting delta values.

(841) bad source/destination for movfp/movpf instruction (Assembler)
The absolute address specified with the `movfp`/`movpf` instruction is too large.

(842) bad bit number (Assembler)
A bit number must be an absolute expression in the range 0-7.

(843) a macro name can't also be an EQU/SET symbol (Assembler)
An `EQU` or `SET` symbol has been found with the same name as a macro. This is not allowed. For example:

```
getval MACRO
MOV  r0, r1
ENDM
getval EQU 55h  ; oops -- choose a different name to the macro
```

(844) lexical error (Assembler)
An unrecognized character or token has been seen in the input.

(845) symbol "*" defined more than once (Assembler)
This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, for example:

```
_next:
MOVE        r0, #55
MOVE        [r1], r0
_next:           ; oops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an underscore prepended to their name after compilation.

(846) relocation error (Assembler)
It is not possible to add together two relocatable quantities. A constant can be added to a relocatable value and two relocatable addresses in the same psect can be subtracted. An absolute value must be used in various places where the assembler must know a value at assembly time.

(847) operand error (Assembler)
The operand to this opcode is invalid. Check your assembler reference manual for the proper form of operands for this instruction.

(848) label defined in this module has also been declared EXTRN (Assembler)
The definition for an assembly label and an `EXTRN` declaration for the same symbol, appear in the same module. Use `GLOBAL` instead of `EXTRN` if you want this symbol to be accessible from other modules.

(849) illegal instruction for this device (Assembler)
The instruction is not supported by this device.

(850) PAGESEL not usable with this device (Assembler)
The `PAGESEL` pseudo-instruction is not usable with the device selected.

(851) illegal destination (Assembler)
The destination (either `,f` or `,w`) is not correct for this instruction.

(852) radix must be from 2 - 16 (Assembler)
The radix specified using the `RADIX` assembler directive must be in the range from 2 (binary) to 16 (hexadecimal).

(853) invalid size for FNSIZE directive (Assembler)
The assembler `FNSIZE` assembler directive arguments must be positive constants.

(855) ORG argument must be a positive constant (Assembler)
An argument to the `ORG` assembler directive must be a positive constant or a symbol which has been equated to a positive constant, for example:

```
ORG -10  /* this must a positive offset to the current psect */
```

(856) ALIGN argument must be a positive constant (Assembler)
The `ALIGN` assembler directive requires a non-zero positive integer argument.

(857) use of both local and global psect flags is illegal with same psect (Linker)
A `local` psect cannot have the same name as a `global` psect, for example:

```
psect text,class=CODE       ; the text psect is implicitly global
MOVE        r0, r1
; elsewhere:
psect text,local,class=CODE
MOVE        r2, r4
```

The `global` flag is the default for a psect if its scope is not explicitly stated.

(859) argument to C option must specify a positive constant (Assembler)
The parameter to the `LIST` assembler control's `C=` option (which sets the column width of the listing output) must be a positive decimal constant number, for example:

```
LIST C=a0h  ; constant must be decimal and positive,
try: LIST C=80
```

(860) page width must be >= 49 (Assembler)
The page width suboption to the `LIST` assembler directive must specify a width of at least 49.

(861) argument to N option must specify a positive constant (Assembler)
The parameter to the `LIST` assembler control's `N` option (which sets the page length for the listing output) must be a positive constant number, for example:

```
LIST N=-3  ; page length must be positive
```

(862) symbol is not external (Assembler)
A symbol has been declared as `EXTRN` but is also defined in the current module.

(863) symbol can't be both extern and public (Assembler)
If the symbol is declared as `extern`, it is to be imported. If it is declared as `public`, it is to be exported from the current module. It is not possible for a symbol to be both.

(864) argument to "size" psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive's `size` flag must be a positive constant number, for example:

```
PSECT text,class=CODE,size=-200  ; a negative size?
```

(865) psect flag “size” redefined (Assembler)
The `size` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, for example:

```
psect spdata,class=RAM,size=400
; elsewhere:
psect spdata,class=RAM,size=500
```

(866) argument to “reloc” psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive’s `reloc` flag must be a positive constant number, for example:

```
psect test,class=CODE,reloc=-4  ; the reloc must be positive
```

(867) psect flag “reloc” redefined (Assembler)
The `reloc` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, for example:

```
psect spdata,class=RAM,reloc=4
; elsewhere:
psect spdata,class=RAM,reloc=8
```

(868) argument to “delta” psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive’s `DELTA` flag must be a positive constant number, for example:

```
PSECT text,class=CODE,delta=-2  ; negative delta value doesn’t make sense
```

(869) psect flag “delta” redefined (Assembler)
The `DELTA` option of a psect has been redefined more than once in the same module.

(870) argument to “pad” psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive’s `PAD` flag must be a non-zero positive integer.

(871) argument to “space” psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive’s `space` flag must be a positive constant number, for example:

```
PSECT text,class=CODE,space=-1  ; space values start at zero
```

(872) psect flag “space” redefined (Assembler)
The `space` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, for example:

```
psect spdata,class=RAM,space=0
; elsewhere:
psect spdata,class=RAM,space=1
```

(873) a psect can only be in one class (Assembler)
You cannot assign a psect to more than one class. The psect was defined differently at this point than when it was defined elsewhere. A psect’s class is specified via a flag as in the following:

```
psect text,class=CODE
```

Look for other psect definitions that specify a different class name.

(874) a psect can only have one “with” option (Assembler)
A psect can only be placed with one other psect. Look for other psect definitions that specify a different with psect name. A psect’s with option is specified via a flag, as shown in the following:

```
psect bss,with=data
; elsewhere
psect bss,with=lktab  ; oops -- bss is to be linked with two psects
```

(875) bad character constant in expression (Assembler)
The character constant was expected to consist of only one character, but was found to be greater than one character or none at all. An assembler specific example:

```
MOV  r0, #'12'    ; '12' specifies two characters
```

(876) syntax error (Assembler)
A syntax error has been detected. This could be caused a number of things.

(877) yacc stack overflow (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(878) -S option used: "*" ignored (Driver)
The indicated assembly file has been supplied to the driver in conjunction with the `-S` option. The driver really has nothing to do because the file is already an assembly file.

(880) invalid number of parameters. Use "* –HELP" for help (Driver)
Improper command-line usage of the of the compiler's driver.

(881) setup succeeded (Driver)
The compiler has been successfully setup using the `--setup` driver option.

(883) setup failed (Driver)
The compiler was not successfully setup using the `--setup` driver option. Ensure that the directory argument to this option is spelled correctly, is syntactically correct for your host operating system and it exists.

(884) please ensure you have write permissions to the configuration file (Driver)
The compiler was not successfully setup using the `--setup` driver option because the driver was unable to access the XML configuration file. Ensure that you have write permission to this file. The driver will search the following configuration files in order:

- the file specified by the environment variable XC_XML
- the file /etc/xc.xml if the directory '/etc ' is writable and there is no .xc.xml file in your home directory
- the file .xc.xml file in your home directory

If none of the files can be located, then the above error will occur.

(889) this * compiler has expired (Driver)
The demo period for this compiler has concluded.

(890) contact Microchip to purchase and re-activate this compiler (Driver)
The evaluation period of this demo installation of the compiler has expired. You will need to purchase the compiler to re-activate it. If you sincerely believe the evaluation period has ended prematurely, contact Microchip technical support.

(891) can't open psect usage map file "*": * (Driver)
The driver was unable to open the indicated file. The psect usage map file is generated by the driver when the driver option `--summary=`*`file`* is used. Ensure that the file is not open in another application.

(892) can't open memory usage map file "*": * (Driver)
The driver was unable to open the indicated file. The memory usage map file is generated by the driver when the driver option `--summary=`*`file`* is used. Ensure that the file is not open in another application.

(893) can't open HEX usage map file "*": * (Driver)
The driver was unable to open the indicated file. The HEX usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(894) unknown source file type "*" (Driver)
The extension of the indicated input file could not be determined. Only files with the extensions `.as`, `.c`, `.obj`, `.usb`, `.p1`, `.lib` or `.hex` are identified by the driver.

(895) can't request and specify options in the one command (Driver)
The usage of the driver options `--getoption` and `--setoption` is mutually exclusive.

(896) no memory ranges specified for data space (Driver)
No on-chip or external memory ranges have been specified for the data space memory for the device specified.

(897) no memory ranges specified for program space (Driver)
No on-chip or external memory ranges have been specified for the program space memory for the device specified.

(899) can't open option file "*" for application "*": * (Driver)
An option file specified by a `--getoption` or `--setoption` driver option could not be opened. If you are using the `--setoption` option, ensure that the name of the file is spelled correctly and that it exists. If you are using the `--getoption` option ensure that this file can be created at the given location or that it is not in use by any other application.

(900) exec failed: * (Driver)
The subcomponent listed failed to execute. Does the file exist? Try re-installing the compiler.

(902) no chip name specified; use "* –CHIPINFO" to see available chip names (Driver)
The driver was invoked without selecting what chip to build for. Running the driver with the `–CHIPINFO` option will display a list of all chips that could be selected to build for.

(904) illegal format specified in "*" option (Driver)
The usage of this option was incorrect. Confirm correct usage with `--help` or refer to the part of the manual that discusses this option.

(905) illegal application specified in "*" option (Driver)
The application given to this option is not understood or does not belong to the compiler.

(907) unknown memory space tag "*" in "*" option specification (Driver)
A parameter to this memory option was a string but did not match any valid tags. Refer to the section of this manual that describes this option to see what tags (if any) are valid for this device.

(908) exit status = * (Driver)
One of the subcomponents being executed encountered a problem and returned an error code. Other messages should have been reported by the subcomponent to explain the problem that was encountered.

(913) "*" option can cause compiler errors in some standard header files (Driver)
Using this option will invalidate some of the qualifiers used in the standard header files, resulting in errors. This issue and its solution are detailed in the section of this manual that specifically discusses this option.

(915) no room for arguments (Preprocessor, Parser, Code Generator, Linker, Objtohex)
The code generator could not allocate any more memory.

(917) argument too long (Preprocessor, Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(918) *: no match (Preprocessor, Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(919) * in chipinfo file "*" at line * (Driver)
The specified parameter in the chip configuration file is illegal.

(920) empty chipinfo file (Driver, Assembler)
The chip configuration file was able to be opened but it was empty. Try re-installing the compiler.

(922) chip "*" not present in chipinfo file "*" (Driver)
The chip selected does not appear in the compiler's chip configuration file. Contact Microchip to see whether support for this device is available or it is necessary to upgrade the version of your compiler.

(923) unknown suboption "*" (Driver)
This option can take suboptions, but this suboption is not understood. This can just be a simple spelling error. If not,`--help` to look up what suboptions are permitted here.

(924) missing argument to "*" option (Driver)
This option expects more data but none was given. Check the usage of this option.

(925) extraneous argument to "*" option (Driver)
This option does not accept additional data, yet additional data was given. Check the usage of this option.

(926) duplicate "*" option (Driver)
This option can only appear once, but appeared more than once.

(928) bad "*" option value (Driver, Assembler)
The indicated option was expecting a valid hexadecimal integer argument.

(929) bad "*" option ranges (Driver)
This option was expecting a parameter in a range format (*`start_of_range-end_of_range`*), but the parameter did not conform to this syntax.

(930) bad "*" option specification (Driver)
The parameters to this option were not specified correctly. Run the driver with `--help` or refer to the driver's chapter in this manual to verify the correct usage of this option.

(931) command file not specified (Driver)
Command file to this application, expected to be found after '`@`' or '`<`' on the command line was not found.

(939) no file arguments (Driver)
The driver has been invoked with no input files listed on its command line. If you are getting this message while building through a third party IDE, perhaps the IDE could not verify the source files to compile or object files to link and withheld them from the command line.

(940) *-bit * placed at * (Objtohex)
Presenting the result of the requested calculation.

(941) bad "*" assignment; USAGE: ** (Hexmate)
An option to Hexmate was incorrectly used or incomplete. Follow the usage supplied by the message and ensure that the option has been formed correctly and completely.

(942) unexpected character on line * of file "*" (Hexmate)
File contains a character that was not valid for this type of file, the file can be corrupt. For example, an Intel HEX file is expected to contain only ASCII representations of hexadecimal digits, colons (`:`) and line formatting. The presence of any other characters will result in this error.

(944) data conflict at address *h between * and * (Hexmate)
Sources to Hexmate request differing data to be stored to the same address. To force one data source to override the other, use the '`+`' specifier. If the two named sources of conflict are the same source, then the source can contain an error.

(945) range (*h to *h) contained an indeterminate value (Hexmate)
The range for this calculation contained a value that could not be resolved. This can happen if the result was to be stored within the address range of the calculation.

(948) result width must be between 1 and 4 bytes (Hexmate)
The requested byte size is illegal. Checksum results must be within 1 to 4 bytes wide. Check the parameters to the `-CKSUM` option.

(949) start of range must be less than end of range (Hexmate)
The `-CKSUM` option has been given a range where the start is greater than the end. The parameters can be incomplete or entered in the wrong order.

(951) start of fill range must be less than end of range (Hexmate)
The `-FILL` option has been given a range where the start is greater than the end. The parameters can be incomplete or entered in the wrong order.

(953) unknown -HELP sub-option: * (Hexmate)
Invalid sub-option passed to `-HELP`. Check the spelling of the sub-option or use `-HELP` with no sub-option to list all options.

(956) -SERIAL value must be between 1 and * bytes long (Hexmate)
The serial number being stored was out of range. Ensure that the serial number can be stored in the number of bytes permissible by this option.

(958) too many input files specified; * file maximum (Hexmate)
Too many file arguments have been used. Try merging these files in several stages rather than in one command.

(960) unexpected record type (*) on line * of "*" (Hexmate)
Intel HEX file contained an invalid record type. Consult the Intel HEX format specification for valid record types.

(962) forced data conflict at address *h between * and * (Hexmate)
Sources to Hexmate force differing data to be stored to the same address. More than one source using the '`+`' specifier store data at the same address. The actual data stored there cannot be what you expect.

(963) range includes voids or unspecified memory locations (Hexmate)
The hash (checksum) range had gaps in data content. The runtime hash calculated is likely to differ from the compile-time hash due to gaps/unused byes within the address range that the hash is calculated over. Filling unused locations with a known value will correct this.

(964) unpaired nibble in -FILL value will be truncated (Hexmate)
The hexadecimal code given to the `-FILL` option contained an incomplete byte. The incomplete byte (nibble) will be disregarded.

(965) -STRPACK option not yet implemented; option will be ignored (Hexmate)
This option currently is not available and will be ignored.

(966) no END record for HEX file "*" (Hexmate)
Intel HEX file did not contain a record of type `END`. The HEX file can be incomplete.

(967) unused function definition "*" (from line *) (Parser)
The indicated `static` function was never called in the module being compiled. Being `static`, the function cannot be called from other modules so this warning implies the function is never used. Either the function is redundant, or the code that was meant to call it was excluded from compilation or misspelled the name of the function.

(968) unterminated string (Assembler)
A string constant appears not to have a closing quote.

(969) end of string in format specifier (Parser)
The format specifier for the `printf()` style function is malformed.

(970) character not valid at this point in format specifier (Parser)
The `printf()` style format specifier has an illegal character.

(971) type modifiers not valid with this format (Parser)
Type modifiers cannot be used with this format.

(972) only modifiers "h" and "l" valid with this format (Parser)
Only modifiers `h` (`short`) and `l` (`long`) are legal with this printf format specifier.

(973) only modifier "l" valid with this format (Parser)
The only modifier that is legal with this format is `l` (for `long`).

(974) type modifier already specified (Parser)
This type modifier has already be specified in this type.

(975) invalid format specifier or type modifier (Parser)
The format specifier or modifier in the printf-style string is illegal for this particular format.

(976) field width not valid at this point (Parser)
A field width cannot appear at this point in a `printf()` type format specifier.

(978) this identifier is already an enum tag (Parser)
This identifier following a `struct` or `union` keyword is already the tag for an enumerated type, and thus should only follow the keyword `enum`, for example:

```
enum IN {ONE=1, TWO};
struct IN {              /* oops -- IN is already defined */
    int a, b;
};
```

(979) this identifier is already a struct tag (Parser)
This identifier following a `union` or `enum` keyword is already the tag for a structure and should only follow the keyword `struct`, for example:

```
struct IN {
    int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(980) this identifier is already a union tag (Parser)
This identifier following a `struct` or `enum` keyword is already the tag for a `union` and should only follow the keyword `union`, for example:

```
union IN {
    int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(981) pointer required (Parser)
A pointer is required here, for example:

```
struct DATA data;
data->a = 9;      /* data is a structure, not a pointer to a structure */
```

(982) unknown op “*” in nxtuse() (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(983) storage class redeclared (Parser)
A variable previously declared as being `static` , has now be redeclared as `extern`.

(984) type redeclared (Parser)
The type of this function or object has been redeclared. This can occur because of two incompatible declarations, or because an implicit declaration is followed by an incompatible declaration, for example:

```
int a;
char a;  /* oops -- what is the correct type? */
```

(985) qualifiers redeclared (Parser)
This function or variable has different qualifiers in different declarations.

(986) enum member redeclared (Parser)
A member of an enumeration is defined twice or more with differing values. Does the member appear twice in the same list or does the name of the member appear in more than one enum list?

(987) arguments redeclared (Parser)
The data types of the parameters passed to this function do not match its prototype.

(988) number of arguments redeclared (Parser)
The number of arguments in this function declaration does not agree with a previous declaration of the same function.

(989) module has code below file base of *h (Linker)
This module has code below the address given, but the `-C` option has been used to specify that a binary output file is to be created that is mapped to this address. This would mean code from this module would have to be placed before the beginning of the file! Check for missing `psect` directives in assembler files.

(990) modulus by zero in #if; zero result assumed (Preprocessor)
A modulus operation in a `#if` expression has a zero divisor. The result has been assumed to be zero, for example:

```
#define ZERO 0
#if FOO%ZERO    /* this will have an assumed result of 0 */
#define INTERESTING
#endif
```

(991) integer expression required (Parser)
In an `enum` declaration, values can be assigned to the members, but the expression must evaluate to a constant of type `int`, for example:

```
enum {one = 1, two, about_three = 3.12};
/* no non-int values allowed */
```

(992) can't find op (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(993) some command-line options are disabled (Driver)
The compiler is operating in demo mode. Some command-line options are disabled.

(994) some command-line options are disabled and compilation is delayed (Driver)
The compiler is operating in demo mode. Some command-line options are disabled, the compilation speed will be slower.

(995) some command-line options are disabled; code size is limited to 16kB, compilation is delayed (Driver)
The compiler is operating in demo mode. Some command-line options are disabled; the compilation speed will be slower, and the maximum allowed code size is limited to 16 KB.

10.3 Messages 1000 Thru 1499

(1015) missing "*" specification in chipinfo file "*" at line * (Driver)
This attribute was expected to appear at least once but was not defined for this chip.

(1016) missing argument* to "*" specification in chipinfo file "*" at line * (Driver)
This value of this attribute is blank in the chip configuration file.

(1017) extraneous argument* to "*" specification in chipinfo file "*" at line * (Driver)
There are too many attributes for the listed specification in the chip configuration file.

(1018) illegal number of "*" specification* (* found; * expected) in chipinfo file "*" at line * (Driver)
This attribute was expected to appear a certain number of times; but it did not appear for this chip.

(1019) duplicate "*" specification in chipinfo file "*" at line * (Driver)
This attribute can only be defined once, but has been defined more than once for this chip.

(1020) unknown attribute "*" in chipinfo file "*" at line * (Driver)
The chip configuration file contains an attribute that is not understood by this version of the compiler. Has the chip configuration file or the driver been replaced with an equivalent component from another version of this compiler?

(1021) syntax error reading “*” value in chipinfo file “*” at line * (Driver)
The chip configuration file incorrectly defines the specified value for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1022) syntax error reading “*” range in chipinfo file “*” at line * (Driver)
The chip configuration file incorrectly defines the specified range for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1024) syntax error in chipinfo file “*” at line * (Driver)
The chip configuration file contains a syntax error at the line specified.

(1025) unknown architecture in chipinfo file “*” at line * (Driver)
The attribute at the line indicated defines an architecture that is unknown to this compiler.

(1026) missing architecture in chipinfo file “*” at line * (Assembler)
The chipinfo file has a device section without an `ARCH` values. The architecture of the device must be specified. Contact Microchip Technical Support if the chipinfo file has not been modified.

(1027) activation was successful (Driver)
The compiler was successfully activated.

(1028) activation was not successful - error code (*) (Driver)
The compiler did not activated successfully.

(1029) compiler not installed correctly - error code (*) (Driver)
This compiler has failed to find any activation information and cannot proceed to execute. The compiler can have been installed incorrectly or incompletely. The error code quoted can help diagnose the reason for this failure. You can be asked for this failure code if contacting Microchip for assistance with this problem.

(1030) Hexmate - Intel HEX editing utility (Build 1.%i) (Hexmate)
Indicating the version number of the Hexmate being executed.

(1031) USAGE: * [input1.HEX] [input2.HEX]... [inputN.HEX] [options] (Hexmate)
The suggested usage of Hexmate.

(1032) use –HELP=<option> for usage of these command line options (Hexmate)
More detailed information is available for a specific option by passing that option to the `-HELP` option.

(1033) available command-line options: (Hexmate)
This is a simple heading that appears before the list of available options for this application.

(1034) type “*” for available options (Hexmate)
It looks like you need help. This advisory suggests how to get more information about the options available to this application or the usage of these options.

(1035) bad argument count (*) (Parser)
The number of arguments to a function is unreasonable. This is an internal compiler error. Contact Microchip Technical Support with details.

(1036) bad “*” optional header length (0x* expected) (Cromwell)
The length of the optional header in this COFF file was of an incorrect length.

(1037) short read on * (Cromwell)
When reading the type of data indicated in this message, it terminated before reaching its specified length.

(1038) string table length too short (Cromwell)
The specified length of the COFF string table is less than the minimum.

(1039) inconsistent symbol count (Cromwell)
The number of symbols in the symbol table has exceeded the number indicated in the COFF header.

(1040) bad : record 0x*, 0x* (Cromwell)
A record of the type specified failed to match its own value.

(1041) short record (Cromwell)
While reading a file, one of the file's records ended short of its specified length.

(1042) unknown * record type 0x* (Cromwell)
The type indicator of this record did not match any valid types for this file format.

(1043) unknown optional header (Cromwell)
When reading this Microchip COFF file, the optional header within the file header was of an incorrect length.

(1044) end of file encountered (Cromwell, Linker)
The end of the file was found while more data was expected. Has this input file been truncated?

(1045) short read on block of * bytes (Cromwell)
A while reading a block of byte data from a UBROF record, the block ended before the expected length.

(1046) short string read (Cromwell)
A while reading a string from a UBROF record, the string ended before the specified length.

(1047) bad type byte for UBROF file (Cromwell)
This UBROF file did not begin with the correct record.

(1048) bad time/date stamp (Cromwell)
This UBROF file has a bad time/date stamp.

(1049) wrong CRC on 0x* bytes; should be * (Cromwell)
An end record has a mismatching CRC value in this UBROF file.

(1050) bad date in 0x52 record (Cromwell)
A debug record has a bad date component in this UBROF file.

(1051) bad date in 0x01 record (Cromwell)
A start of program record or segment record has a bad date component in this UBROF file.

(1052) unknown record type (Cromwell)
A record type could not be determined when reading this UBROF file.

(1053) additional RAM ranges larger than bank size (Driver)
A block of additional RAM being requested exceeds the size of a bank. Try breaking the block into multiple ranges that do not cross bank boundaries.

(1054) additional RAM range out of bounds (Driver)
The RAM memory range as defined through custom RAM configuration is out of range.

(1055) RAM range out of bounds (*) (Driver)
The RAM memory range as defined in the chip configuration file or through custom configuration is out of range.

(1056) unknown chip architecture (Driver)
The compiler is attempting to compile for a device of an architecture that is either unsupported or disabled.

(1058) assertion (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1059) rewrite loop (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1081) static initialization of persistent variable "*" (Parser, Code Generator)
A `persistent` variable has been assigned an initial value. This is somewhat contradictory as the initial value will be assigned to the variable during execution of the compiler's startup code; however, the persistent qualifier requests that this variable shall be unchanged by the compiler's startup code.

(1082) size of initialized array element is zero (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1088) function pointer "*" is used but never assigned a value (Code Generator)
A function call involving a function pointer was made, but the pointer was never assigned a target address, for example:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1089) recursive function call to "*" (Code Generator)
A recursive call to the specified function has been found. The call can be direct or indirect (using function pointers) and can be either a function calling itself, or calling another function whose call graph includes the function under consideration.

(1090) variable "*" is not used (Code Generator)
This variable is declared but has not been used by the program. Consider removing it from the program.

(1091) main function "*" not defined (Code Generator)
The main function has not been defined. Every C program must have a function called `main()`.

(1094) bad derived type (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1095) bad call to typeSub() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1096) type should be unqualified (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1097) unknown type string "*" (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1098) conflicting declarations for variable "*" (*:*) (Parser, Code Generator)
Differing type information has been detected in the declarations for a variable, or between a declaration and the definition of a variable, for example:

```
extern long int test;
int test;    /* oops -- which is right? int or long int ? */
```

(1104) unqualified error (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1118) bad string "*" in getexpr(J) (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1119) bad string "*" in getexpr(LRN) (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1121) expression error (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1137) match() error: * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1157) W register must be W9 (Assembler)
The working register required here has to be W9, but an other working register was selected.

(1159) W register must be W11 (Assembler)
The working register required here has to be W11, but an other working register was selected.

(1178) the "*" option has been removed and has no effect (Driver)
This option no longer exists in this version of the compiler and has been ignored. Use the compiler's `--help` option or refer to the manual to find a replacement option.

(1179) interrupt level for function "*" cannot exceed * (Code Generator)
The interrupt level for the function specified is too high. Each interrupt function is assigned a unique interrupt level. This level is considered when analyzing the call graph and reentrantly called functions. If using the `interrupt_level` pragma, check the value specified.

(1180) directory "*" does not exist (Driver)
The directory specified in the setup option does not exist. Create the directory and try again.

(1182) near variables must be global or static (Code Generator)
A variable qualified as `near` must also be qualified with `static` or made global. An auto variable cannot be qualified as near.

(1183) invalid version number (Activation)
During activation, no matching version number was found on the Microchip activation server database for the serial number specified.

(1184) activation limit reached (Activation)
The number of activations of the serial number specified has exceeded the maximum number allowed for the license.

(1185) invalid serial number (Activation)
During activation, no matching serial number was found on the Microchip activation server database.

(1186) license has expired (Driver)
The time-limited license for this compiler has expired.

(1187) invalid activation request (Driver)
The compiler has not been correctly activated.

(1188) network error * (Activation)
The compiler activation software was unable to connect to the Microchip activation server via the network.

(1190) FAE license only - not for use in commercial applications (Driver)
Indicates that this compiler has been activated with an FAE license. This license does not permit the product to be used for the development of commercial applications.

(1191) licensed for educational use only (Driver)
Indicates that this compiler has been activated with an education license. The educational license is only available to educational facilities and does not permit the product to be used for the development of commercial applications.

(1192) licensed for evaluation purposes only (Driver)
Indicates that this compiler has been activated with an evaluation license.

(1193) this license will expire on * (Driver)
The compiler has been installed as a time-limited trial. This trial will end on the date specified.

(1195) invalid syntax for "*" option (Driver)
A command line option that accepts additional parameters was given inappropriate data or insufficient data. For example, an option can expect two parameters with both being integers. Passing a string as one of these parameters or supplying only one parameter could result in this error.

(1198) too many "*" specifications; * maximum (Hexmate)
This option has been specified too many times. If possible, try performing these operations over several command lines.

(1199) compiler has not been activated (Driver)
The trial period for this compiler has expired. The compiler is now inoperable until activated with a valid serial number. Contact Microchip to purchase this software and obtain a serial number.

(1200) Found %0*lXh at address *h (Hexmate)
The code sequence specified in a `-FIND` option has been found at this address.

(1201) all FIND/REPLACE code specifications must be of equal width (Hexmate)
All find, replace and mask attributes in this option must be of the same byte width. Check the parameters supplied to this option. For example, finding 1234h (2 bytes) masked with FFh (1 byte) results in an error; but, masking with 00FFh (2 bytes) works.

(1202) unknown format requested in -FORMAT: * (Hexmate)
An unknown or unsupported INHX format has been requested. Refer to documentation for supported INHX formats.

(1203) unpaired nibble in * value will be truncated (Hexmate)
Data to this option was not entered as whole bytes. Perhaps the data was incomplete or a leading zero was omitted. For example, the value Fh contains only four bits of significant data and is not a whole byte. The value 0Fh contains eight bits of significant data and is a whole byte.

(1204) * value must be between 1 and * bytes long (Hexmate)
An illegal length of data was given to this option. The value provided to this option exceeds the maximum or minimum bounds required by this option.

(1205) using the configuration file *; you can override this with the environment variable HTC_XML (Driver)
This is the compiler configuration file selected during compiler setup. This can be changed via the `HTC_XML` environment variable. This file is used to determine where the compiler has been installed.

(1207) some of the command line options you are using are now obsolete (Driver)
Some of the command line options passed to the driver have now been discontinued in this version of the compiler; however, during a grace period these old options will still be processed by the driver.

(1208) use –help option or refer to the user manual for option details (Driver)
An obsolete option was detected. Use `--help` or refer to the manual to find a replacement option that will not result in this advisory message.

(1209) An old MPLAB tool suite plug-in was detected. (Driver)
The options passed to the driver resemble those that the Microchip MPLAB 8 IDE would pass to a previous version of this compiler. Some of these options are now obsolete – however, they were still interpreted. It is recommended that you install an updated Microchip options plug-in for the IDE.

(1210) Visit the Microchip website (www.microchip.com) for a possible upgrade (Driver)
Visit our website to see if an upgrade is available to address the issue(s) listed in the previous compiler message. Navigate to the MPLAB XC8 C Compiler page and look for a version upgrade downloadable file. If your version is current, contact Microchip Technical Support for further information.

(1212) Found * (%0*lXh) at address *h (Hexmate)
The code sequence specified in a `-FIND` option has been found at this address.

(1213) duplicate ARCH for * in chipinfo file at line * (Assembler, Driver)
The chipinfo file has a device section with multiple ARCH values. Only one ARCH value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(1218) can't create cross reference file * (Assembler)
The assembler attempted to create a cross reference file; but it could not be created. Check that the file's path name is correct.

(1228) unable to locate installation directory (Driver)
The compiler cannot determine the directory where it has been installed.

(1230) dereferencing uninitialized pointer "*" (Code Generator)
A pointer that has not yet been assigned a value has been dereferenced. This can result in erroneous behavior at runtime.

(1235) unknown keyword * (Driver)
The token contained in the USB descriptor file was not recognized.

(1236) invalid argument to *: * (Driver)
An option that can take additional parameters was given an invalid parameter value. Check the usage of the option or the syntax or range of the expected parameter.

(1237) endpoint 0 is pre-defined (Driver)
An attempt has been made to define endpoint 0 in a USB file.

(1238) FNALIGN failure on * (Linker)
Two functions have their auto/parameter blocks aligned using the `FNALIGN` directive, but one function calls the other, which implies that must not be aligned. This will occur if a function pointer is assigned the address of each function, but one function calls the other. For example:

```
int one(int a) { return a; }
int two(int a) { return two(a)+2; }  /* ! */
int (*ip)(int);
ip = one;
ip(23);
ip = two;      /* ip references one and two; two calls one */
ip(67);
```

(1239) pointer * has no valid targets (Code Generator)
A function call involving a function pointer was made, but the pointer was never assigned a target address, for example:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1240) unknown algorithm type (%i) (Driver)
The error file specified after the `-Efile` or `-E+file` options could not be opened. Check to ensure that the file or directory is valid and that has read only access.

(1241) bad start address in * (Driver)
The start of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1242) bad end address in * (Driver)
The end of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1243) bad destination address in * (Driver)
The destination address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1245) value greater than zero required for * (Hexmate)
The align operand to the Hexmate `-FIND` option must be positive.

(1246) no RAM defined for variable placement (Code Generator)
No memory has been specified to cover the banked RAM memory.

(1247) no access RAM defined for variable placement (Code Generator)
No memory has been specified to cover the access bank memory.

(1248) symbol (*) encountered with undefined type size (Code Generator)
The code generator was asked to position a variable, but the size of the variable is not known. This is an internal compiler error. Contact Microchip Technical Support with details.

(1250) could not find space (* byte*) for variable * (Code Generator)
The code generator could not find space in the banked RAM for the variable specified.

(1253) could not find space (* byte*) for auto/param block (Code Generator)
The code generator could not find space in RAM for the psect that holds auto and parameter variables.

(1254) could not find space (* byte*) for data block (Code Generator)
The code generator could not find space in RAM for the data psect that holds initialized variables.

(1255) conflicting paths for output directory (Driver)
The compiler has been given contradictory paths for the output directory via any of the `-O` or `--OUTDIR` options, for example:

```
--outdir=../../   -o../main.HEX
```

(1256) undefined symbol "*" treated as HEX constant (Assembler)
A token which could either be interpreted as a symbol or a hexadecimal value does not match any previously defined symbol and so will be interpreted as the latter. Use a leading zero to avoid the ambiguity, or use an alternate radix specifier such as `0x`. For example:

```
MOV  a, F7h  ; is this the symbol F7h, or the HEX number 0xF7?
```

(1257) local variable "*" is used but never given a value (Code Generator)
An auto variable has been defined and used in an expression, but it has not been assigned a value in the C code before its first use. Auto variables are not cleared on startup and their initial value is undefined. For example:

```
int main(void) {
    double src, out;
    out = sin(src);   /* oops -- what value was in src? */
```

(1258) possible stack overflow when calling function "*" (Code Generator)
The call tree analysis by the code generator indicates that the hardware stack can overflow. This should be treated as a guide only. Interrupts, the assembler optimizer and the program structure can affect the stack usage. The stack usage is based on the C program and does not include any call tree derived from assembly code.

(1259) can't optimize for both speed and space (Driver)
The driver has been given contradictory options of compile for speed and compile for space, for example:

```
--opt=speed,space
```

(1260) macro "*" redefined (Assembler)
More than one definition for a macro with the same name has been encountered, for example:

```
MACRO fin
    ret
ENDM
MACRO fin   ; oops -- was this meant to be a different macro?
    reti
ENDM
```

(1261) string constant required (Assembler)
A string argument is required with the `DS` or `DSU` directive, for example:

```
DS ONE     ; oops -- did you mean DS "ONE"?
```

(1262) object "*" lies outside available * space (Code Generator)
An absolute variable was positioned at a memory location which is not within the memory defined for the target device, for example:

```
int data __at(0x800);     /* oops -- is this the correct address? */
```

(1264) unsafe pointer conversion (Code Generator)
A pointer to one kind of structure has been converted to another kind of structure and the structures do not have a similar definition, for example:

```
struct ONE {
    unsigned a;
    long b;          /* ! */
```

```
} one;
struct TWO {
    unsigned a;
    unsigned b;      /* ! */
} two;
struct ONE * oneptr;
oneptr = & two;    /* oops -- was ONE meant to be same struct as TWO? */
```

(1267) fixup overflow referencing * into * bytes at 0x* (Linker)
See error message 1356 for more information.

(1268) fixup overflow storing 0x* in * bytes at * (Linker)
See error message 1356 for more information.

(1273) Omniscient Code Generation not available in Free mode (Driver)
This message advises that advanced features of the compiler are not be enabled in this Free mode compiler.

(1275) the qualifier "*" is only applicable to functions (Parser)
A qualifier which only makes sense when used in a function definition has been used with a variable definition.

```
interrupt int dacResult;  /* oops --
the interrupt qualifier can only be used with functions */
```

(1276) buffer overflow in DWARF location list (Cromwell)
A buffer associated with the ELF/DWARF debug file has overflowed. Contact Microchip Technical Support with details.

(1278) omitting "*" which does not have a location (Cromwell)
A variable has no storage location listed and will be omitted from the debug output. Contact Microchip Technical Support with details.

(1284) malformed mapfile while generating summary: CLASS expected but not found (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1285) malformed mapfile while generating summary: no name at position * (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1286) malformed mapfile while generating summary: no link address at position * (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1287) malformed mapfile while generating summary: no load address at position * (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1288) malformed mapfile while generating summary: no length at position * (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1289) line range limit exceeded, possibly affecting ability to debug code (Cromwell)
A C statement has produced assembly code output whose length exceeds a preset limit. This means that debug information produced by CROMWELL may not be accurate. This warning does not indicate any potential code failure.

(1290) buffer overflow in DWARF debugging information entry (Cromwell)
A buffer associated with the ELF/DWARF debug file has overflowed. Contact Microchip Technical Support with details.

(1291) bad ELF string table index (Cromwell)
An ELF file passed to CROMWELL is malformed and cannot be used.

(1292) malformed define in .SDB file * (Cromwell)
The named SDB file passed to CROMWELL is malformed and cannot be used.

(1293) couldn’t find type for “*” in DWARF debugging information entry (Cromwell)
The type of symbol could not be determined from the SDB file passed to CROMWELL. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1294) there is only one day left until this license expires (Driver)
The compiler is running as a demo and will be unable to run in PRO mode after the evaluation license has expired in less than one day’s time. After expiration, the compiler can be operated in Free mode indefinitely, but will produce a larger output binary.

(1295) there are * days left until this license will expire (Driver)
The compiler is running as a demo and will be unable to run in PRO mode after the evaluation license has expired in the indicated time. After expiration, the compiler can be operated in Free mode indefinitely, but will produce a larger output binary.

(1296) source file “*” conflicts with “*” (Driver)
The compiler has encountered more than one source file with the same base name. This can only be the case if the files are contained in different directories. As the compiler and IDEs based the names of intermediate files on the base names of source files, and intermediate files are always stored in the same location, this situation is illegal. Ensure the base name of all source files are unique.

(1297) option * not available in Free mode (Driver)
Some options are not available when the compiler operates in Free mode. The options disabled are typically related to how the compiler is executed, e.g., `--getoption` and `--setoption`, and do not control compiler features related to code generation.

(1298) use of * outside macros is illegal (Assembler)
Some assembler directives, e.g., `EXITM`, can only be used inside macro definitions.

(1299) non-standard modifier “*” - use “*” instead (Parser)
A printf placeholder modifier has been used which is non-standard. Use the indicated modifier instead. For example, the standard `hh` modifier should be used in preference to `b` to indicate that the value should be printed as a `char` type.

(1300) maximum number of program classes reached; some classes may be excluded from debugging information (Cromwell)
CROMWELL is passed a list of class names on the command line. If the number of class names passed in is too large, not all will be used and there is the possibility that debugging information will be inaccurate.

(1301) invalid ELF section header; skipping (Cromwell)
CROMWELL found an invalid section in an ELF section header. This section will be skipped.

(1302) could not find valid ELF output extension for this device (Cromwell)
The extension could not be for the target device family.

(1303) invalid variable location detected: * - * (Cromwell)
A symbol location could not be determined from the SDB file.

(1304) unknown register name: "*" (Cromwell)
The location for the indicated symbol in the SDB file was a register, but the register name was not recognized.

(1305) inconsistent storage class for variable: "*" (Cromwell)
The storage class for the indicated symbol in the SDB file was not recognized.

(1306) inconsistent size (* vs *) for variable: "*" (Cromwell)
The size of the symbol indicated in the SDB file does not match the size of its type.

(1307) psect * truncated to * bytes (Driver)
The psect representing either the stack or heap could not be made as large as requested and will be truncated to fit the available memory space.

(1308) missing/conflicting interrupts sub-option; defaulting to "*" (Driver)
The suboptions to the `--INTERRUPT` option are missing or malformed, for example:

```
--INTERRUPTS=single,multi
```

Oops, did you mean single-vector or multi-vector interrupts?

(1309) ignoring invalid runtime * sub-option (*) using default (Driver)
The indicated suboption to the `--RUNTIME` option is malformed, for example:

```
--RUNTIME=default,speed:0y1234
```

Oops, that should be 0x1234.

(1310) specified speed (*Hz) exceeds max operating frequency (*Hz); defaulting to *Hz (Driver)
The frequency specified to the perform suboption to `--RUNTIME` option is too large for the selected device.

```
--RUNTIME=default,speed:0xffffffff
```

Oops, that value is too large.

(1311) missing configuration setting for config word *; using default (Driver)
The configuration settings for the indicated word have not be supplied in the source code and a default value will be used.

(1312) conflicting runtime perform sub-option and configuration word settings; assuming *Hz (Driver)
The configuration settings and the value specified with the perform suboption of the `--RUNTIME` options conflict and a default frequency has been selected.

(1313) * sub-options ("*") ignored (Driver)
The argument to a suboption is not required and will be ignored.

```
--OUTPUT=intel:8
```

Oops, the :8 is not required

(1314) illegal action in memory allocation (Code Generator)
This is an internal error. Contact Microchip Technical Support with details.

(1315) undefined or empty class used to link psect * (Linker)
The linker was asked to place a psect within the range of addresses specified by a class, but the class was either never defined, or contains no memory ranges.

(1316) attribute "*" ignored (Parser)
An attribute has been encountered that is valid, but which is not implemented by the parser. It will be ignored by the parser and the attribute will have no effect. Contact Microchip Technical Support with details.

(1317) missing argument to attribute "*" (Parser)
An attribute has been encountered that requires an argument, but this is not present. Contact Microchip Technical Support with details.

(1318) invalid argument to attribute "*" (Parser)
An argument to an attribute has been encountered, but it is malformed. Contact Microchip Technical Support with details.

(1319) invalid type "*" for attribute "*" (Parser)
This indicated a bad option passed to the parser. Contact Microchip Technical Support with details.

(1320) attribute "*" already exists (Parser)
This indicated the same attribute option being passed to the parser more than once. Contact Microchip Technical Support with details.

(1321) bad attribute -T option "%s" (Parser)
The attribute option passed to the parser is malformed. Contact Microchip Technical Support with details.

(1322) unknown qualifier "%s" given to -T (Parser)
The qualifier specified in an attribute option is not known. Contact Microchip Technical Support with details.

(1323) attribute expected (Parser)
The `__attribute__` directive was used but did not specify an attribute type.

```
int rv (int a) __attribute__(())  /* oops -- what is the attribute? */
```

(1324) qualifier "*" ignored (Parser)
Some qualifiers are valid, but cannot be implemented on some compilers or target devices. This warning indicates that the qualifier will be ignored.

(1342) whitespace after "\" (Preprocessor)
Whitespace characters have been found between a backslash and newline characters and will be ignored.

(1343) hexfile data at address 0x* (0x*) overwritten with 0x* (Objtohex)
The indicated address is about to be overwritten by additional data. This would indicate more than one section of code contributing to the same address.

(1346) can't find 0x* words for psect "*" in segment "*" (largest unused contiguous range 0x%lX) (Linker)
See also message (491). The new form of message also indicates the largest free block that the linker could find. Unless there is a single space large enough to accommodate the psect, the linker will issue this message. Often when there is banking or paging involved the largest free space is much smaller than the total amount of space remaining,

(1347) can't find 0x* words (0x* withtotal) for psect "*" in segment "*" (largest unused contiguous range 0x%lX) (Linker)

See also message (593). The new form of message also indicates the largest free block that the linker could find. Unless there is a single space large enough to accommodate the psect, the linker will issue this message. Often when there is banking or paging involved the largest free space is much smaller than the total amount of space remaining,

(1348) enum tag "*" redefined (from *:*) (Parser)

More than one `enum` tag with the same name has been defined, The previous definition is indicated in the message.

```
enum VALS { ONE=1, TWO, THREE };
enum VALS { NINE=9, TEN };  /* oops -- is VALS the right tag name? */
```

(1349) initialization of absolute variable "*" in RAM is not supported (Code Generator)

An absolute variable, one defined using `__at(`*`address`*`)`, cannot be assigned an initial value when it is defined. Place an assignment to the variable at an appropriate location in your program.

```
int foobar __at(0x20) = 0x55;  /* oops --
 you cannot assign a value to an absolute variable */
```

(1350) pointer operands to "-" must reference the same array (Code Generator)

If two addresses are subtracted, the addresses must be of the same object to be ANSI compliant.

```
int * ip;
int fred, buf[20];
ip = &buf[0] - &fred;  /* oops --
 second operand must be an address of a "buf" element */
```

(1352) truncation of operand value (0x*) to * bits (Assembler)

The operand to an assembler instruction was too large and was truncated.

```
movlw  0x321  ; oops -- is this the right value?
```

(1354) ignoring configuration setting for unimplemented word * (Driver)

A Configuration Word setting was specified for a Word that does not exist on the target device.

```
__CONFIG(3, 0x1234);  /* config word 3 does not exist on an 18C801 */
```

(1355) in-line delay argument too large (Code Generator)

The in-line delay sequence `_delay` has been used, but the number of instruction cycles requested is too large. Use this routine multiple times to achieve the desired delay length.

```
#include <xc.h>
int main(void) {
    delay(0x400000); /* oops -- cannot delay by this number of cycles */
}
```

(1356) fixup overflow referencing * * (0x*) into * byte* at 0x*/0x* -> 0x* (* */0x*) (Linker)**

'Fixup' is the process conducted by the linker of replacing symbolic references to operands with an absolute value. This takes place after positioning the psects (program sections or blocks) into the available memory. 'Fixup overflow' is when a symbol's value is too large to fit within the assembler instruction. For example, if an assembler instruction has an 8-bit field to hold an address and the linker determines that the symbol used to represent this address has the value 0x110, then clearly this value cannot be encoded into the instruction.

Fixup errors are often caused by hand-written assembly code. Common mistakes that trigger these errors include failing to mask a full, banked data address in file register instructions, or failing to mask the destination address in jump or call instructions. If this error is triggered by assembly code generated from C source, then it is often that constructs like switch() statements have generated a block of assembly too large for jump instructions to span. Adjusting the default linker options can also causes such errors.

To identify these errors, follow these steps.

- Perform a debug build (in MPLAB X IDE select **Debug** > **Discrete Debugger Operation** > **Build for Debugging**; alternatively, on the command line use the `-D__DEBUG` option).
- Open the relevant assembler list file (ensure the MPLAB X IDE project properties has **XC8 Compiler** > **Preprocessing and Messaging** > **Generate the ASM listing file** enabled; alternatively, on the command line, use the `-Wa,-a` option).
- Find the instruction at the address quoted in the error message.

Consider the following error message.

```
main.c: 4: (1356)(linker) fixup overflow referencing psect bssBANK1 (0x100) into 1 byte at
0x7FF0/0x1 -> 0x7FF0 (main.obj 23/0x0)
```

The file being linked was `main.obj`. This tells you the assembly list file in which you should be looking is `main.lst`. The location of the instruction at fault is 0x7FF0. (You can also tell from this message that the instruction is expecting a 1 byte quantity—this size is rounded to the nearest byte—but the value was determined to be 0x100.)

In the assembly list file, search for the address specified in the error message.

```
61  007FF0  6F00      movwf   _foobar,b       ;#
```

and to confirm, look for the symbol referenced in the assembler instruction at this address in the symbol table at the bottom of the same file.

```
Symbol Table                    Tue Oct 28 11:06:37 2014
_foobar 0100
```

In this example, the hand-written PIC18 `movwf` instruction causing the problem takes an 8-bit offset into a bank of memory, but clearly the address 0x100 exceeds this size. The instruction should have been written as:

```
MOVWF        BANKMASK(_foo)
```

which masks out the top bits of the address containing the bank information.

If the assembler instruction that caused this error was generated by the compiler, in the assembler list file look back up the file from the instruction at fault to determine which C statement has generated this instruction. You will then need to examine the C code for possible errors.

(1357) fixup overflow storing 0x* in * byte* at 0x*/0x* -> 0x* (* */0x*) (Linker)**
See message (1356).

(1358) no space for * temps (*) (Code Generator)
The code generator was unable to find a space large enough to hold the temporary variables (scratch variables) for this program.

(1359) no space for * parameters (Code Generator)
The code generator was unable to find a space large enough to hold the parameter variables for a particular function.

(1360) no space for auto/param * (Code Generator)
The code generator was unable to find a space large enough to hold the auto variables for a particular function. Some parameters passed in registers can need to be allocated space in this auto area as well.

(1361) syntax error in configuration argument (Parser)
The argument to `#pragma config` was malformed.

```
#pragma config WDT     /* oops -- is WDT on or off? */
```

(1362) configuration setting *=* redefined (Code Generator)
The same `config` pragma setting have been issued more than once with different values.

```
#pragma config WDT=OFF
#pragma config WDT=ON     /* oops -- is WDT on or off? */
```

(1363) unknown configuration setting (* = *) used (Driver)
The configuration value and setting is not known for the target device. The use of an unknown configuration register number may also trigger this message.

```
#pragma config WDR=ON     /* oops -- did you mean WDT? */
#pragma config CONFIG1L=0x46 /* oops -- no 1L register on a 18F4520 */
```

(1364) can’t open configuration registers data file * (Driver)
The file containing value configuration settings could not be found.

(1365) missing argument to pragma “varlocate” (Parser)
The argument to `#pragma varlocate` was malformed.

```
#pragma varlocate    /* oops -- what do you want to locate & where? */
```

(1366) syntax error in pragma “varlocate” (Parser)
The argument to `#pragma varlocate` was malformed.

```
#pragma varlocate fred     /* oops -- which bank for fred? */
```

(1367) end of file in _asm (Parser)
An end-of-file marker was encountered inside a `_asm _endasm` block.

(1368) assembler message: * (Assembler)
Displayed is an assembler advisory message produced by the `MESSG` directive contained in the assembler source.

(1369) can’t open proc file * (Driver)
The proc file for the selected device could not be opened.

(1370) peripheral library support is not available for the * (Driver)
The peripheral library is not available for the selected device.

(1371) float type can’t be bigger than double type; double has been changed to * bits (Driver)
Use of the `-fshort-double` options has result in the size of the `double` type being smaller than that of the `float` type. This is not permitted by the C Standard. The `double` type size has been increased to be that indicated.

(1372) interrupt level cannot be greater than * (Code Generator)
The specific `interrupt_level` is too high for the device selected.

```
#pragma interrupt_level 4
// oops - there aren't that many interrupts on this device
```

(1374) the compiler feature “*” is no longer supported; * (Driver)
The feature indicated is no longer supported by the compiler.

(1375) multiple interrupt functions (* and *) defined for device with only one interrupt vector (Code Generator)
The named functions have both been qualified interrupt, but the target device only supports one interrupt vector and hence one interrupt function.

```
interrupt void isr_lo(void) {
// ...
}
interrupt void isr_hi(void) {   // oops, cannot define two ISRs
// ...
}
```

(1376) initial value (*) too large for bitfield width (*) (Code Generator)
A structure with bit-fields has been defined an initialized with values. The value indicated it too large to fit in the corresponding bit-field width.

```
struct {
unsigned flag :1;
unsigned mode :3;
} foobar = { 1, 100 };   // oops, 100 is too large for a 3 bit object
```

(1377) no suitable strategy for this switch (Code Generator)
The compiler was unable to determine the switch strategy to use to encode a C `switch` statement based on the code and your selection using the `#pragma switch` directive. You can need to choose a different strategy.

(1378) syntax error in pragma "*" (Parser)
The arguments to the indicated pragma are not valid.

```
#pragma addrqual ingore  // oops -- did you mean ignore?
```

(1379) no suitable strategy for this switch (Code Generator)
The compiler encodes `switch()` statements according to one of a number of strategies. The specific number and values of the case values, and the switch expression, as well as the `switch` pragma determine the strategy chosen. This error indicates that no strategy was available to encode the `switch()` statement. Contact Microchip support with program details.

(1380) unable to use switch strategy "*" (Code Generator)
The compiler encodes `switch()` statements according to one of a number of strategies. The specific number and values of the case values, and the switch expression, as well as the `switch` pragma, determine the strategy chosen. This error indicates that the strategy that was requested cannot be used to encode the `switch()` statement. Contact Microchip support with program details.

(1381) invalid case label range (Parser)
The values supplied for the case range are not correct. They must form an ascending range and be integer constants.

```
case 0 ... -2:  // oops -- do you mean -2 ... 0  ?
```

(1385) * "*" is deprecated (declared at *:*) (Parser)
Code is using a variable or function that was marked as being deprecated using an attribute.

```
char __attribute__((deprecated)) foobar;
foobar = 9;     // oops -- this variable is near end-of-life
```

(1386) unable to determine the semantics of the configuration setting "*" for register "*" (Parser, Code Generator)

The numerical value supplied to a configuration bit setting has no direct association setting specified in the data sheet. The compiler will attempt to honor your request, but check your device data sheet.

```
#pragma config OSC=11
// oops -- there is no direct association for that value on an 18F2520
// either use OSC=3 or OSC=RC
```

(1387) in-line delay argument must be constant (Code Generator)

The `_delay` in-line function can only take a constant expression as its argument.

```
int delay_val = 99;
_delay(delay_val);   // oops, argument must be a constant expression
```

(1388) configuration setting/register of "*" with 0x* will be truncated by 0x* (Parser, Code Generator)

A Configuration bit has been programmed with a value that is either too large for the setting, or is not one of the prescribed values.

```
#pragma config WDTPS=138  // oops -- do you mean 128?
```

(1389) attempt to reprogram configuration * "*" with * (is *) (Parser, Code Generator)

A Configuration bit that was already programmed has been programmed again with a conflicting setting to the original.

```
#pragma config WDT=ON
#pragma config WDT=OFF  // oops -- watchdog on or off?
```

(1390) identifier specifies insignificant characters beyond maximum identifier length (Parser)

An identifier that has been used is so long that it exceeds the set identifier length. This can mean that long identifiers cannot be correctly identified and the code will fail. The maximum identifier length can be adjusted using the -N option.

```
int theValueOfThePortAfterTheModeBitsHaveBeenSet;
// oops, make your symbol shorter or increase the maximum
// identifier length
```

(1391) constant object size of * exceeds the maximum of * for this chip (Code Generator)

The const object defined is too large for the target device.

```
const int array[200] = { ... };  // oops -- not on a Baseline part!
```

(1392) function "*" is called indirectly from both mainline and interrupt code (Code Generator)

A function has been called by main-line (non-interrupt) and interrupt code. If this warning is issued, it highlights that such code currently violates a compiler limitation for the selected device.

(1393) possible hardware stack overflow detected; estimated stack depth: * (Code Generator)

The compiler has detected that the call graph for a program could be using more stack space that allocated on the target device. If this is the case, the code can fail. The compiler can only make assumption regarding the stack usage, when interrupts are involved and these lead to a worst-case estimate of stack usage. Confirm the function call nesting if this warning is issued.

(1394) attempting to create memory range (* - *) larger than page size * (Driver)
The compiler driver has detected that the memory settings include a program memory “page” that is larger than the page size for the device. This would mostly likely be the case if the --ROM option is used to change the default memory settings. Consult your device data sheet to determine the page size of the device you are using and to ensure that any contiguous memory range you specify using the --ROM option has a boundary that corresponds to the device page boundaries.

```
--ROM=100-1fff
```

The above might need to be paged. If the page size is 800h, the above could specified as

```
--ROM=100-7ff,800-fff,1000-17ff,1800-1fff
```

(1395) notable code sequence candidate suitable for compiler validation suite detected (*) (Code Generator)
The compiler has in-built checks that can determine if combinations of internal code templates have been encountered. Where unique combinations are uncovered when compiling code, this message is issued. This message is not an error or warning and its presence does not indicate possible code failure, but if you are willing to participate, the code you are compiling can be sent to Support to assist with the compiler testing process.

(1396) “*” positioned in the * memory region (0x* - 0x*) reserved by the compiler (Code Generator)
Some memory regions are reserved for use by the compiler. These regions are not normally used to allocate variables defined in your code. However, by making variables absolute, it is possible to place variables in these regions and avoid errors that would normally be issued by the linker. Absolute variables can be placed at any location, even on top of other objects. This warning from the code generator indicates that an absolute has been detected that will be located at memory that the compiler will be reserving. You must locate the absolute variable at a different location. This message will commonly be issued when placing variables in the common memory space.

```
char shared __at(0x7);   // oops, this memory is required by the compiler
```

(1397) unable to implement non-stack call to “*”; possible hardware stack overflow (Code Generator)
The compiler must encode a C function call without using a call assembly instruction and the hardware stack (i.e., use a lookup table), but is unable to. A call instruction might be required if the function is called indirectly via a pointer, but if the hardware stack is already full, an additional call will cause a stack overflow.

(1401) eeprom qualified variables can’t be accessed from both interrupt and mainline code (Code Generator)
All `eeprom` variables are accessed via routines that are not reentrant. Code might fail if an attempt is made to access `eeprom`-qualified variables from interrupt and main-line code. Avoid accessing `eeprom` variables in interrupt functions.

(1402) a pointer to eeprom can’t also point to other data types (Code Generator)
A pointer cannot have targets in both the EEPROM space and ordinary data space.

(1403) pragma “*” ignored (Parser)
The pragma you have specified has no effect and will be ignored by the compiler. This message can only be issued in C18 compatibility mode.

```
#pragma varlocate "mySection" fred  // oops -- not accepted
```

(1404) unsupported: * (Parser)
The unsupported `__attribute__` has been used to indicate that some code feature is not supported.

The message printed will indicate the feature that is not supported and which should be avoided.

(1405) storage class specifier "*" ignored (Parser)
The storage class you have specified is not required and will be ignored by the compiler. This message can only be issued in C18 compatibility mode.

```
int procInput(auto int inValue)  // oops -- no need for auto
{ ...
```

(1406) auto eeprom variables are not supported (Code Generator)
Variables qualified as `eeprom` cannot be auto. You can define `static` local objects qualified as `eeprom`, if required.

```
int main(void) {
eeprom int mode;  // oops -- make this static or global
```

(1407) bit eeprom variables are not supported (Code Generator)
Variables qualified as `eeprom` cannot have type `bit`.

```
eeprom bit myEEbit;    // oops -- you cannot define bits in EEPROM
```

(1408) ignoring initialization of far variables (Code Generator)
Variables qualified as `far` cannot be assigned an initial value. Assign the value later in the code.

```
far int chan = 0x1234; // oops -- you cannot assign a value here
```

(1409) warning number used with pragma "warning" is invalid (Parser)
The message number used with the `warning` pragma is below zero or larger than the highest message number available.

```
#pragma warning disable 1316 13350   // oops -- possibly number 1335?
```

(1410) can't assign the result of an invalid function pointer (Code Generator)
The compiler will allow some functions to be called via a constant cast to be a function pointer, but not all. The address specified is not valid for this device.

```
foobar += ((int (*)(int))0x0)(77);
// oops -- you cannot call a function with a NULL pointer
```

(1411) Additional ROM range out of bounds (Driver)
Program memory specified with the `-mrom` option is outside of the on-chip, or external, memory range supported by this device.

```
-mrom=default,+2000-2ffff
```

Oops -- memory too high, should that be 2fff?

(1412) missing argument to pragma "warning disable" (Parser)
Following the `#pragma warning` disable should be a comma-separated list of message numbers to disable.

```
#pragma warning disable  // oops -- what messages are to be disabled?
```

Try something like the following.

```
#pragma warning disable 1362
```

(1413) pointer comparisons involving address of "*", positioned at address 0x0, may be invalid (Code Generator)
An absolute object placed at address 0 has had its address taken. By definition, this is a `NULL` pointer and code which checks for `NULL` (i.e., checks to see if the address is valid) can fail.

```
int foobar __at(0x00);
int  * ip;
int main(void)
{
    ip = &foobar;  // oops -- 0 is not a valid address
```

(1414) option * is defunct and has no effect (Driver)
The option used is now longer supported. It will be ignored.

```
xc8 --chip=18f452 --cp=24 main.c
```

Oops -- the `--cp` option is no longer required.

(1415) argument to "merge" psect flag must be 0 or 1 (Assembler)
This psect flag must be assigned a 0 or 1.

```
PSECT myTxt,class=CODE,merge=true  ; oops -- I think you mean merge=1
```

(1416) psect flag "merge" redefined (Assembler)
A psect with a name seen before specifies a different `merge` flag value to that previously seen.

```
psect mytext,class=CODE,merge=1
; and later
psect mytext,class=CODE,merge=0
; Oops, can mytext be merged or not?
```

(1417) argument to "split" psect flag must be 0 or 1 (Assembler)
This psect flag must be assigned a 0 or 1.

```
psect mytext,class=CODE,split=5
```

Oops, the `split` flag argument must be 0 or 1.

(1418) Attempt to read "control" qualified object which is Write-Only (Code Generator)
An attempt was made to read a write-only register.

```
state = OPTION;  // oops -- you cannot read this register
```

(1419) using the configuration file *; you can override this with the environment variable XC_XML (Driver)
This is the compiler configuration file that is selected during compiler setup. This can be changed via the `XC_XML` environment variable. This file is used to determine where the compiler has been installed. See message 1205.

(1420) ignoring suboption "*" (Driver)
The suboption you have specified is not valid in this implementation and will be ignored.

```
--RUNTIME=default,+ramtest
```

oops -- what is ramtest?

(1421) the qualifier __xdata is not supported by this architecture (Parser)
The qualifier you have specified is not valid in this implementation and will be ignored.

```
__xdata int coeff[2];  // that has no meaning for this target
```

(1422) the qualifier __ydata is not supported by this architecture (Parser)
The qualifier you have specified is not valid in this implementation and will be ignored.

```
__ydata int coeff[2];  // that has no meaning for this target
```

(1423) case ranges are not supported (Driver)
The use of GCC-style numerical ranges in case values does not conform to the CCI Standard. Use individual case labels and values to conform.

```
switch(input) {
case 0 ... 5:    // oops -- ranges of values are not supported
    low();
```

(1424) short long integer types are not supported (Parser)
The use of the `short long` type does not conform to the CCI Standard. Use the corresponding long type instead.

```
short long typeMod;    // oops -- not a valid type for CCI
```

(1425) __pack qualifier only applies to structures and structure members (Parser)
The qualifier you have specified only makes sense when used with structures or structure members. It will be ignored.

```
__pack int c;  // oops -- there aren't inter-member spaces to pack in an int
```

(1426) 24-bit floating point types are not supported; * have been changed to 32-bits (Driver)
Floating-point types must be 32-bits wide to conform to the CCI Standard. These types will be compiled as 32-bit wide quantities.

```
-fshort-double=24
```

oops -- you cannot set this `double` size

(1427) machine-dependent path specified in name of included file; use -I instead (Preprocessor)
To conform to the CCI Standard, header file specifications must not contain directory separators.

```
#inlcude <inc\lcd.h>    // oops -- do not indicate directories here
```

Remove the path information and use the `-I` option to indicate this, for example:

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

and issue the `-Ilcd` option.

(1428) “*” is not supported; this feature will be ignored (Driver)
The specified option is not supported and will have no effect on compilation.

```
xc8-cc -mcpu=18f4520 --html main.c
```

Oops, `--html` is not a valid option.

(1429) attribute “*” is not understood by the compiler; this attribute will be ignored (Parser)
The indicated attribute you have used is not valid with this implementation. It will be ignored.

```
int x __attribute__ ((deprecate)) = 0;
```

oops -- did you mean `deprecated`?

(1430) section redefined from "*" to "*" (Parser)
You have attempted to place an object in more than one section.

```
int __section("foo") __section("bar") myvar;  // oops -- which section should it be in?
```

(1431) the __section specifier is applicable only to variable and function definitions at file-scope (Parser)
You cannot attempt to locate local objects using the `__section()` specifier.

```
int main(void) {
int __section("myData") counter;  // oops -- you cannot specify a section for autos
```

(1432) "*" is not a valid section name (Parser)
The section name specified with `__section()` is not a valid section name. The section name must conform to normal C identifier rules.

```
int __section("28data") counter;  // oops -- name cannot start with digits
```

(1433) function "*" could not be inlined (Assembler)
The specified function could not be made in-line. The function will called in the usual way.

```
int inline getData(int port)   // sorry -- no luck inlining this
{   //...
```

(1434) missing name after pragma "intrinsic" (Parser)
The `intrinsic` pragma needs a function name. This pragma is not needed in most situations. If you mean to in-line a function, see the `inline` keyword or pragma.

```
#pragma intrinsic    // oops -- what function is intrinsically called?
```

(1435) variable "*" is incompatible with other objects in section "*" (Code Generator)
You cannot place variables that have differing startup initializations into the same psect. That is, variables that are cleared at startup and variables that are assigned an initial non-zero value must be in different psects. Similarly, bit objects cannot be mixed with byte objects, like char or int.

```
int __section("myData") input;   // okay
int __section("myData") output;  // okay
int __section("myData") lvl = 0x12;  // oops -- not with uninitialized
bit __section("myData") mode;    // oops again -- no bits with bytes
// each different object to their own new section
```

(1436) "*" is not a valid nibble; use hexadecimal digits only (Parser)
When using `__IDLOC()`, the argument must only consist of hexadecimal digits with no radix specifiers or other characters. Any character which is not a hexadecimal digit will be programmed as a 0 in the corresponding location.

```
__IDLOC(0x51);  // oops -- you cannot use the 0x radix modifier
```

(1437) CMF error * (Cromwell, Linker)
The CMF file being read by Cromwell or the linker is invalid. Unless you have modified or manually generated this file, this is an internal error. Contact Microchip Technical Support with details.

(1438) pragma "*" options ignored (Parser)
You have used unsupported options with a pragma. The options will be ignored.

```
#pragma inline=forced  // oops -- no options allowed with this pragma
```

(1439) message: * (Parser)
This is a programmer generated message; there is a pragma directive causing this advisory to be printed. This is only printed when using IAR C extensions.

```
#pragma message "this is a message from your programmer"
```

(1440) big-endian storage is not supported by this compiler (Parser)
You have specified the `__big_endian` IAR extension for a variable. The big-endian storage format is not supported by this compiler. Remove the specification and ensure that other code does not rely on this endianism.

```
__big_endian int  volume;  // oops -- this won't be big endian
```

(1441) use __at() instead of '@' and ensure the address is applicable (Parser)
You have used the `@` address specifier when using the IAR C extensions. Any address specified is unlikely to be correct on a new architecture. Review the address in conjunction with your device data sheet. To prevent this warning from appearing again, use the reviewed address with the `__at()` specifier instead.

(1442) type used in definition is incomplete (Parser)
When defining objects, the type must be complete. If you attempt to define an object using an incomplete type, this message is issued.

```
typedef struct foo foo_t;
foo_t x;   // oops -- you cannot use foo_t until it is fully defined
struct foo {
int i;
};
```

(1443) unknown --EXT sub-option “*” (Driver)
The suboption to the `--EXT` option is not valid.

```
xc8 --chip=18f8585 x.c --ext=arm --ext=cci
```

Oops -- valid choices are `iar`, `cci` and `xc8`

(1444) respecified C extension from “*” to “*” (Driver)
The `--EXT` option has been used more than once, with conflicting arguments. The last use of the option will dictate the C extensions accepted by the compiler.

```
xc8 --chip=18f8585 x.c --ext=iar --ext=cci
```

Oops -- which C extension do you mean?

(1445) #advisory: * (Preprocessor)
This is a programmer generated message; there is a directive causing this advisory to be printed.

```
#advisory "please listen to this good advice"
```

(1446) #info: * (Preprocessor)
This is a programmer generated message; there is a directive causing this advisory to be printed. It is identical to `#advisory` messages (1445).

```
#info "the following is for your information only"
```

(1447) extra -L option (-L*) ignored (Preprocessor)
This error relates to a duplicate `-L` option being passed to the preprocessor. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

(1448) no dependency file type specified with -L option (Preprocessor)
This error relates to a malformed `-L` option being passed to the preprocessor. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

(1449) unknown dependency file type (*) (Preprocessor)
This error relates to a unknown dependency file format being passed to the preprocessor. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

(1450) invalid --*-spaces argument (*) (Cromwell)
The option passed to Cromwell does not relate to a valid memory space. The space arguments must be a valid number that represents the space.

```
--data-spaces=a
```

Oops — a is not a valid data space number.

(1451) no * spaces have been defined (Cromwell)
Cromwell must be passed information that indicates the type for each numbered memory space. This is down via the `--code-spaces` and `--data-spaces` options. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

(1452) one or more spaces are defined as data and code (Cromwell)
The options passed to Cromwell indicate memory space is both in the code and data space. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

```
--code-space=1,2  --data-space=1
```

Oops — is space 1 code or data?

(1453) stack size specified for non-existent * interrupt (Driver)
The `-mstack` option has been used to specify the maximum sizes for each stack. A size has been used for each interrupt, but the compiler cannot see the corresponding interrupt function definition, which means the stack space can never be used. Ensure that you create the interrupt function for each interrupt the device supports.

```
-mstack=reentrant:20:20:auto
```

Oops, you have asked for two interrupt stacks, but the compiler cannot see both interrupt function definitions.

(1454) stack size specified (*) is greater than available (*) (Driver)
The `-mstack` option has been used to specify the maximum sizes for each stack, but the total amount of memory requested exceeds the amount of memory available.

```
-mstack=software:1000:1000:20000
```

Oops, that is too much stack space for a small device.

(1455) unrecognized stack size "*" in "*" (Driver)
The `-mstack` option has been used to specify the maximum sizes for each stack, but one or more of the sizes are not a valid value. Use only decimal values in this option, or the token auto, for a default size.

```
-mstack=software:30:all:default
```

Oops, only use decimal numbers or `auto`.

(1456) too many stack size specifiers (Driver)
Too many software stack maximum sizes have been specified in the `-mstack` option. The maximum stack sizes are optional. If used, specify one size for each interrupt and one for main-line code.

```
-mstack=reentrant:20:20:auto
```

Oops, too many sizes for a device with only one interrupt.

(1457) local variable “*” cannot be made absolute (Code Generator)
You cannot specify the address of any local variable, whether it be an auto, parameter, or static local object.

```
int pushState(int a) {
int cnt __at(0x100); // oops -- you cannot specify an address ...
```

(1458) Omniscient Code Generation not available in Standard mode (Driver)
This message warns you that not all optimizations are enabled in the Standard operating mode.

(1459) peripheral library support is missing for the * (Driver)
The peripheral libraries do not have code present for the device you have selected. Disable the option that links in the peripheral library.

(1460) function-level profiling is not available for the selected chip (Driver)
Function profiling is only available for PIC18 or enhanced mid-range devices. If you are not using such a device, do not attempt to use function profiling.

(1461) insufficient h/w stack to profile function “*” (Code Generator)
Function profiling requires a level of hardware stack. The entire stack has been used by this program so not all functions can be profiled. The indicated function will not have profiling code embedded into it, and it will not contribute to the profiling information displayed by MPLAB X IDE.

(1462) reentrant data stack model option conflicts with stack management option and will be ignored (Code Generator)
The managed stack option allows conversion of function calls that would exceed the hardware stack depth to calls that will use a lookup table. This option cannot be enabled if the reentrant function model is also enabled. If you attempt to use both the managed stack and reentrant function model options, this message will be generated. Code will be compiled with the stack management option disabled. Either disable the reentrant function model or the managed stack option.

(1463) reentrant data stack model not supported on this device; using compiled stack for data (Code Generator)
The target device does not support reentrant functions. The program will be compiled so that stack-based data is placed on a compiled stack.

(1464) number of arguments passed to function “*” does not match function's prototype (Code Generator)
A function was called with arguments, but the definition of the function had an empty parameter list (as opposed to a parameter list of void).

```
int test();    // oops--this should define the parameters
...
test(12, input);
```

(1465) the stack frame size for function “*” (* bytes) has exceeded the maximum allowable (* bytes) (Code Generator)
The compiler has been able to determine that the software stack requirements for the named function’s auto, parameter, and temporary variables exceed the maximum allowable. The limits are 31 for enhanced mid-range devices and 127 for PIC18 devices. Reduce the size or number of these variables. Consider `static` local objects instead of auto objects.

```
reentrant int addOffset(int offset) {
int report[400];    // oops--this will never fit on the software stack
```

(1466) registers * unavailable for code generation of this expression (Code Generator)
The compiler has been unable to generate code for this statement. This is essentially a “can’t generate code” error message (message 712), but the reason for this inability to compile relates to there not being enough registers available. See message 712 for suggested workarounds.

(1467) pointer used for writes includes read-only target “*” (Code Generator)
A pointer to a non-`const`-qualified type is being used to write a value, but the compiler knows that this pointer has targets (the first of which is indicated) that have been qualified `const`. This could lead to code failure or other error messages being generated.

```
void keepTotal(char * cp) {
    *cp += total;
}
char c;
const char name[] = "blender";
keepTotal(&c);
keepTotal(&name[2]);  // oops--will write a read-only object
```

(1468) unknown ELF/DWARF specification (*) in --output option (Driver)
The ELF suboption uses flags that are unknown.

```
—output=elf:3
```

Oops, there is no `elf` flag of 3.

This `elf` suboption and its flags are usually issued by the MPLAB X IDE plugin. Contact Microchip Technical Support with details of the compiler and IDE if this error is issued.

(1469) function specifier “reentrant/software” used with “*” ignored (Code Generator)
The `reentrant` (or `software`) specifier was used with a function (indicated) that cannot be encoded to use the software stack. The specifier will be ignored and the function will use the compiled stack.

```
reentrant int main(void) // oops--main cannot be reentrant
{ ...
```

(1470) trigraph sequence “??*” replaced (Preprocessor)
The preprocessor has replaced a trigraph sequence in the source code. Ensure you intended to use a trigraph sequence.

```
char label[] = “What??!”;  // you do know that’s a trigraph
// sequence, right?
```

(1471) indirect function call via a NULL pointer ignored (Code Generator)
The compiler has detected a function pointer with no valid target other than `NULL`. That pointer has been used to call a function. The call will not be made.

```
int (*fp)(int, int);
result = fp(8,10);  // oops--this pointer has not been initialized
```

(1472) --CODEOFFSET option ignored: * (Driver)
The compiler is ignoring an invocation of the `-mcodeoffset` option. The printed description will indicate whether the option is being ignored because the compiler has seen this option previously or the compilation mode does not support its use.

(1474) read-only target “*” may be indirectly written via pointer (Code Generator)
This is the same as message 1467, but for situations where an error is required. The compiler has encountered a pointer that is used to write, and one or more of the pointer’s targets are read-only.

```
const char c = ‘x’;
char * cp = &c; // will produce warning 359 about address assignment
```

```
*cp = 0x44;     // oops--you ignored the warning above, now you are
// actually going to write using the pointer?
```

(1478) initial value for “*” differs to that in *:* (Code Generator)
The named object has been defined more than once and its initial values do not agree. Remember that uninitialized objects of static storage duration are implicitly initialized with the value zero (for all object elements or members, where appropriate).

```
char myArray[5] = { 0 };
// elsewhere
char myArray[5] = {0,2,4,6,8};  // oops--previously initialized
// with zeros, now with different values
```

(1479) EEPROM data not supported by this device (Parser)
The `eeprom` qualifier was used but there is no EEPROM on the target device. Any instances of this qualifier will be ignored.

```
eeprom int serialNo;          // oops--no EEPROM on this device
```

(1480) initial value(s) not supplied in braces; zero assumed (Code Generator)
The assignment operator was used to indicate that the object was to be initialized, but no values were found in the braces. The object will be initialized with the value(s) 0.

```
int xy_map[3][3] = { };   // oops--did you mean to supply values?
```

(1481) call from non-reentrant function, “*”, to “*” might corrupt parameters
(Code Generator)

If several functions can be called indirectly by the same function pointer, they are called ‘buddy’ functions, and the parameters to buddy functions are aligned in memory. This allows the parameters to be loaded without knowing exactly which function was called by the pointer (as is often the case). However, this means that the buddy functions cannot directly or indirectly call each other.

```
// fpa can call any of these, so they are all buddies
int (*fpa[])(int) = { one, two, three };
int one(int x) {
    return three(x+1);  // oops--one() cannot call buddy three()
}
```

(1482) absolute object * overlaps * (Linker)
The reservation for an absolute object has been found to overlap with the memory reserved by another absolute object.

```
unsigned char nfo[6] __at(0x80);
unsigned char nfo2[6] __at(0x7b);  //oops--this overlaps nfo
```

(1483) __pack qualifier ignored (Parser)
The `__pack` qualifier has no affect on auto or `static` local structures and has been ignored.

```
int setInput(void) {
    __pack struct {       //oops--this will not be packed
        unsigned x, y;
    } inputData;
...
```

(1484) the branch errata option is turned on and a BRW instruction was detected (Assembler)

The use of this instruction may cause code failure with the selected device. Check the published errata for your device to see if this restriction is applicable for your device revision. If so, remove this instruction from hand-written assembly code.

```
btfsc status,2
brw  next      ;oops--this instruction cannot be safely used
call  update
```

(1485) * mode is not available with the current license and other modes are not permitted by the NOFALLBACK option (Driver)

This compiler's license does not allow the requested compiler operating mode. Since the `--nofallback` option is enabled, the compiler has produced this error and will not fall back to a lower operating mode. If you believe that you are entitled to use the compiler in the requested mode, this error indicates that your compiler might not be activated correctly.

(1486) size of pointer cannot be determined during preprocessing. Using default size * (Preprocessor)

The preprocessor cannot determine the size of pointer type. Do not use the `sizeof` operator in expressions that need to be evaluated by the preprocessor.

```
#if sizeof(int *) == 3   // oops - you can't take the size of a pointer type
#define MAX 40
#endif
```

(1488) the stack frame size for function "*" may have exceeded the maximum allowable (* bytes) (Code Generator)

This message is emitted in the situation where the indicated function's software-stack data has exceeded the theoretical maximum allowable size. Data outside this stack space will only be accessible by some instructions that could attempt to access it. In some situations the excess data can be retrieved, your code will work as expected, and you can ignore this warning. This is likely if the function calls a reentrant function that returns a large object, like a structure, on the stack. At other times, instructions that are unable to access this data will, in addition to this warning, trigger an error message at the assembly stage of the build process, and you will need to look at reducing the amount of stack data defined by the function.

(1489) unterminated IF directive at end of psect * (Assembler)

The assembler has reached the end of the named psect and not seen the terminating `ENDIF` directive associated with the last `IF` or `ELSIF` directive previously encountered.

```
psect mytext,class=CODE,reloc=2
movlw 20h
IF TEST_ONLY
    movlw 00h
    movwf _mode,c  ; oops--where does the IF end?
psect nexttext,class=CODE,reloc=2
```

(1490) ENDIF not inside an IF directive (Assembler)

The assembler has encountered an `ENDIF` directive that does not have any corresponding `IF` or `ELSIF` directive.

```
psect mytext,class=CODE,reloc=2
movlw 20h
IF TEST_ONLY
    movlw 00h
ENDIF
ENDIF    ; oops--what does this terminate?
```

(1491) runtime sub-option “*” is not available for this device (Driver)
A specified suboption to the `--RUNTIME` option is not available for the selected device.

```
xc8 --CHIP=MCP19114 --RUNTIME=+osccal main.c
```

Oops, the `osccal` suboption is not available for this device.

(1492) using updated 32-bit floating-point libraries; improved accuracy might increase code size (Code Generator)
This advisory message ensures you are aware of the changes in 32-bit floating-point library code operation that might lead to an increase in code size.

(1493) updated 32-bit floating-point routines might trigger “can't find space” messages appearing after updating to this release; consider using the smaller 24-bit floating-point types (Linker)
This advisory message ensures you are aware of the changes in 32-bit floating-point library code operation which might lead to the Can’t Find Space error message that has been issued.

(1494) invalid argument to normalize32 (Assembler)
The `NORMALIZE32` operator has been used on an operand that is not a literal constant.

```
NORMALIZE(_foobar)  ; oops--that must be a literal constant operand
```

(1495) ADDFSR/SUBFSR instruction argument must be 0-3 (Assembler)
The operand to this instruction must be a literal constant and in the range 0 to 3, inclusive.

```
addfsr 1, 6  ; oops--the offset must be between 0 to 3
```

(1496) arithmetic on pointer to void yields Undefined Behavior (Code Generator)
Performing operations on pointers requires the size of the pointed-to object, which is not known in the case of generic (`void *`) pointers.

```
void * vp;
vp++;   // oops—how can this be incremented without knowing what it points to?
```

(1497) more than one *interrupt function defined (Code Generator)
Only one interrupt function of the same priority can be defined.

```
void interrupt lo_isr(void) {  // oops — was this meant to be a low_priority interrupt?
  …
}
void interrupt hi_isr(void) {
  …
}
```

(1498) pointer (*) in expression may have no targets (Code Generator)
A pointer that contains `NULL` has been dereferenced. Assign the pointer a valid address before doing so.

```
char * cp, c;
c = *cp;    // oops —what is cp pointing to?
```

(1499) only decimal floating-point constants can be suffixed “f” or “F”
The floating-point constant suffix has been used with an integer value.

```
float myFloat = 100f*3.2;  // oops — is ‘100f’ mean to be a hex or floating-point value?
```

10.4 Messages 1500 Thru 1999

(1500) invalid token in #if expression (Preprocessor)
There is a malformed preprocessor expression.

```
#define LABEL
#define TEST 0
#if (LABEL == TEST) // oops--LABEL has no replacement text
```

(1504) the PIC18 extended instruction set was enabled but is not supported by this compiler (Parser)
The MPLAB XC8 compiler does not support generation of code using the PIC18 extended instruction set. The extended instruction set configuration bit must always be disabled.

```
#pragma config XINST=ON  // oops--this must be disabled at all times
```

(1505) interrupts not supported by this device (Code Generator)
You have attempted to define an interrupt function for a device that does not support interrupts.

```
void interrupt myIsr(void)  // oops--nothing will trigger this
{ ... }
```

(1506) multiple interrupt functions (* and *) defined at interrupt level * (Code Generator)
More than one interrupt function has been defined for the same priority.

```
void interrupt low_priority
isr(void)
{ ... }
void interrupt low_priority    // oops--you can have two ISRs
loisr(void)                // with the same priority
{ ... }
```

(1507) asmopt state popped when there was no pushed state (Assembler)
The state of the assembler optimizers was popped in assembly code but there was no corresponding push.

```
movlw  20h
movwf  LATB
opt asmopt_pop; oops--there was never a state pushed
```

(1508) specifier "__ram" ignored (Parser)
This pointer-target specifier cannot be used with an ordinary variable and it will be ignored. Confirm that this definition was not meant to indicate a pointer type.

```
__ram int ip;   // oops -- was this meant to be a pointer?
```

(1509) specifier "__rom" ignored (Parser)
This pointer-target specifier cannot be used with an ordinary variable and it will be ignored. Confirm that this definition was not meant to indicate a pointer type.

```
const __rom int cip;   // oops -- was this meant to be a pointer?
```

(1510) non-reentrant function "*" appears in multiple call graphs and has been duplicated by the compiler (Code Generator)
This message indicates that the generated output for a function has been duplicated since it has been called from both main-line and interrupt code. It does not indicate a potential code failure. If you do not want function output duplicated, consider using the hybrid stack model (if possible), or restructure your source code.

(1511) stable/invariant mode optimizations no longer implemented; option will be ignored (Driver)
This option is no longer available and has been ignored.

(1512) stable/invariant mode optimizations no longer implemented; specifier will be ignored (Code Generator)
This specifier is no longer available and has been ignored.

(1513) target "*" of pointer "*" not in the memory space specified by * (Code Generator)
The pointer assigned an address by this statement was defined using a pointer-target specifier. This assignment might be assigning addresses to the pointer that conflict with that memory space specifier.

```
__rom int * ip;
int foobar;
ip = &foobar;  // oops -- foobar is in data memory, not program memory
```

(1514) "__ram" and "__rom" specifiers are mutually exclusive (Parser)
Use of both the `__ram` and `__rom` pointer-target specifiers with the same pointer does not make sense. If a pointer should be able to represent targets in any memory space, do not use either of these specifiers.

```
// oops -- you can't limit ip to only point to objects in ram and
// also only point to objects in rom
__ram __rom int * ip;
```

(1515) disabling OCG optimizations for this device is not permitted (Driver)
Due to memory limits, projects targeting some devices cannot be built. Ensure that the OCG category of optimization is enabled.

(1516) compiler does not support 64-bit integers on the target architecture
Due to memory restrictions, the current device cannot support 64-bit integers and a smaller integer will be used instead. Choose a type smaller than `long long` to suppress this warning.

```
long long int result;  // oops - this will not be 64-bits wide
```

(1517) peripheral library support only available for C90 (Driver)
The legacy peripheral library was build for the C90 standard and cannot reliably be used for other C standards.

(1518) * function call made with an incomplete prototype (*) (Code Generator)
A function has been called with the compiler not having seen a complete prototype for that function. Check for an empty parameter list in the declaration.

```
void foo();  // oops -- how will this call be encoded?
int main(void)
{
    foo();
}
```

1519 note-psects will ignore optimisation-related psect flags (Assembler)
Psects using the `note` psect flag cannot be optimized and any additional psect flags which request optimization will be ignored. Psects using the note flag typically contain debug information not related to your project code.

1520 malformed mapfile while generating summary: no space at position * (Driver)
While printing the memory summary after compilation, the psect information read in from the map file was malformed.

1521 internal error encountered creating DWARF information; contact Microchip support with details (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

1522 RAM access bit operand not specified, assuming * (Assembler)
The assembly instruction is missing the RAM access operand and the assembler has made the stated assumption as to the instruction's destination location. Always use the RAM access bit to ensure the intended operation of your code is clearly stated.

```
    movwf input   ; oops - use for example movwf input,b to indicate banked access, etc
```

1523 debug_source state popped when there was no pushed state (Assembler)
The state of the `debug_source` setting was popped but no previous value had been pushed.

```
DEBUG_SOURCE asm
    MY_UNLOCK_MACRO
DEBUG_SOURCE pop   ; oops - there was no prior push of the debug source state
```

1600 "*" argument : * (Hexmate)
There is an error in an argument to a Hexmate option. The message indicates the offending argument and the problem.

1601 "*" argument : * (Hexmate)
There is a warning in an argument to a Hexmate option. The message indicates the offending argument and the potential problem.

1602 contents of the hex-data (*) do not conform with the chosen output type (*) (Hexmate)
There is data to be written to the HEX file that is not valid for the particular HEX file format chosen, for example, data might be at an address too large for the supported format. Consider a different output format or check the source of the offending data.

1604 Storage Area Flash has been enabled by the configuration bits; ensure this memory is not used for program code (Assember)
A configuration bit setting has enabled the storage area flash memory on this device. This memory should not be used by any other part of the program, so it will need to be reserved using an option (e.g. `-mreserve`). Check your device data sheet for more information.

```
#pragma config SAFEN=ON  ;make sure ordinary program code does not use this space
```

1605 Block Table Read Protection has been enabled by the configuration bits; this may affect variable initialization and reading constants in program memory (Assembler)
A configuration bit setting has enabled the block table read protection feature. This can affect any code that reads from program memory, resulting in code failing. Check your device data sheet for more information.

```
#pragma config EBTRB=ON  ;remember this might affect reading program memory
```

10.5 Messages 2000 Thru 2499

(2000) * attribute/specifier has a misplaced keyword (*) (Parser)
An attribute token has been used in a context where it was not expected.

```
// oops -- ’base’ is a token which has specific meaning
void __interrupt(irq(base)) isr(void)
```

(2001) * attribute/specifier has a misplaced parenthesis (Parser)
The parentheses used in this attribute construct are not correctly formed. Check to ensure that you do not have extra brackets and that they are in the correct position.

```
void __interrupt(irq((TMR0)) isr(void) // oops -- one too many ’(’s
```

(2002) __interrupt attribute/specifier has conflicting priority-levels (Parser)
More than one priority has been assigned to an interrupt function definition.

```
//oops -- is it meant to be low or high priority?

void __interrupt(irq(TMR0), high_priority, low_priority) tc0Int(void)
```

(2003) * attribute/specifier has a duplicate keyword (*) (Parser)
The same token has been used more than once in this attribute. Check to ensure that one of these was not meant to be something else.

```
//oops -- using high_priority twice has no special meaning

void __interrupt(irq(TMR0), high_priority, high_priority) tc0Int(void)
```

(2004) __interrupt attribute/specifier has an empty "irq" list (Parser)
The `irq()` argument to the `__interrupt()` specifier takes a comma-separated list of interrupt vector numbers or symbols. At least one value or symbol must be present to link this function to the interrupt source.

```
//oops -- irq() does not indicate the interrupt source

void __interrupt(irq(),high_priority) tc0Int(void)
```

(2005) __interrupt attribute/specifier has an empty "base" list (Parser)
The `base()` argument to the `__interrupt()` specifier is optional, but when used it must take a comma-separated list of interrupt vector table addresses. At least one address must be present to position the vector table. If you do not specify the base address with an ISR, its vector will be located in an interrupt vector table located at an address equal to the reset value of the IVTBASE register.

```
//oops -- base() was used but did not indicate a vector table address

void __interrupt(irq(TMR0), base()) tc0Int(void)
```

(2006) __interrupt attribute/specifier has a duplicate "irq" (*) (Parser)
An `irq()` argument to the `__interrupt()` specifier has been used more than once.

```
//oops -- is one of those sources wrong?

void __interrupt(irq(TMR0,TMR0)) tc0Int(void)
```

(2007) __interrupt attribute/specifier has a duplicate "base" (*) (Parser)
The same `base()` argument to the `__interrupt()` specifier has been used more than once.

```
//oops -- is one of those base addresses wrong?

void __interrupt(irq(TMR0), base(0x100,0x100)) tc0Int(void)
```

(2008) unknown "irq" (*) in __interrupt attribute/specifier (Parser)
The interrupt symbol or number used with the `irq()` argument to the `__interrupt()` specifier does not correspond with an interrupt source on this device.

```
//oops -- what interrupt source is TODO?

void __interrupt(irq(TODO),high_priority) tc0Int(void)
```

(2009) * attribute/specifier has a misplaced number (*) (Parser)
A numerical value appears in an attribute where it is not expected.

```
//oops -- this specifier requires specific argument, not a number
void __interrupt(0) isr(void)
```

(2010) __interrupt attribute/specifier contains a misplaced interrupt source name (*) (Parser)

An interrupt source name can only be used as an argument to `irq()`.

```
//oops -- base() needs a vector table address

void __interrupt(irq(TMR0), base(TMR0)) tc0Int(void)
```

(2011) __interrupt attribute/specifier has a base (*) not supported by this device (Parser)

The address specified with the `base()` argument to the `__interrupt()` specifier is not valid for the target device. It cannot, for example, be lower than the reset value of the IVTBASE register.

```
//oops -- the base() address is too low

void __interrupt(irq(TMR0), base(0x00)) tc0Int(void)
```

(2012) * attribute/specifier is only applicable to functions (Parser)

The `__interrupt()` specifier has been used with something that is not a function.

```
// oops -- foobar is an int, not an ISR

__interrupt(irq(TMR0)) int foobar;
```

(2013) argument "*" used by "*" attribute/specifier not supported by this device (Parser)

The argument of the indicated specifier is not valid for the target device.

```
// oops -- base() can't be used with a device that does not

// support vectored interrupts

void __interrupt(base(0x100)) myMidrangeISR(void)
```

(2014) interrupt vector table @ 0x* already has a default ISR "*" (Code Generator)

You can indicate only one default interrupt function for any vector location not specified in a vector table. If you have specified this twice, check to make sure that you have specified the correct `base()` address for each default.

```
void __interrupt(irq(default), base(0x100)) tc0Int(void) { ...

void __interrupt(irq(default), base(0x100)) tc1Int(void) { ...
// oops -- did you mean to use different different base() addresses?
```

(2015) interrupt vector table @ 0x* already has an ISR (*) to service IRQ * (*) (Parser or Code Generator)

You have specified more than one interrupt function to handle a particular interrupt source in the same vector table.

```
void __interrupt(irq(TMR0), base(0x100)) tc0Int(void) { ...

void __interrupt(irq(TMR0), base(0x100)) tc1Int(void) { ...
// oops -- did you mean to use different different base() addresses?
```

(2016) interrupt function "*" does not service any interrupt sources (Code Generator)

You have defined an interrupt function but did not indicate which interrupt source this function should service. Use the `irq()` argument to indicate the source or sources.

```
//oops -- what interrupt does this service?

void __interrupt(low_priority, base(0x100)) tc0Int(void)
```

(2017) config programming has disabled multi-vectors, “irq” in __interrupt attribute/specifier is ignored (Code Generator)
An interrupt function has used the `irq()` argument to specify an interrupt source, but the vector table has been disabled via the configuration bits. Either re-enable vectored interrupts or use the priority keyword in the `__interrupt()` specifier to indicate the interrupt source.

```
#pragma config MVECEN=0

void __interrupt(irq(TMR0), base(0x100)) tc0Int(void)

// oops -- you cannot disable the vector table then allocate interrupt

// functions a vector source using irq()
```

(2018) interrupt vector table @ 0x* has multiple functions (* and *) defined at interrupt level * (Code Generator)
The program for a device operating in legacy mode has specified a vector table that contains more than one function at the same interrupt priority-level in the same table. In this mode, there can be at most one interrupt function for each priority level in each vector table.

```
#pragma config MVECEN=0

void __interrupt(high_priority) tc0Int(void) {...

void __interrupt(high_priority) tc1Int(void) {...
```

(2019) * interrupt vector in table @ 0x* is unassigned, will be programmed with a * (Code Generator)
In a program for a device operating in legacy mode, an interrupt vector in the indicated vector table has not been programmed with an address. The compiler will program this vector with an address as specified by the `-mundefints` option.

(2020) IRQ * (*) in vector table @ 0x* is unassigned, will be programmed with the address of a * (Code Generator)
The interrupt vector in the indicated vector table has not been programmed with an address. The compiler will program this vector with an address as specified by the `-mundefints` option.

(2021) invalid runtime “*” sub-option argument (*) (Driver)
The argument to a sub-option specified with the `--RUNTIME` option is not valid.

```
--RUNTIME=default,+ivt:reset
```

Oops, the `ivt` suboption requires a numeric address as its argument.

(2022) runtime sub-option “ivt” specifies a base address (0x*) not supported by this device (Driver)
The address specified with the `ivt` sub-option is not valid for the selected target device. It cannot, for example, be lower than the reset value of the IVTBASE register.

(2023) IVT @ 0x* will be selected at startup (Code Generator)
The source code defines more than one IVT and no address was specified with the `ivt` sub-option to the `--RUNTIME` option to indicate which table should be selected at startup. The IVT with the lowest address will be selected by the compiler. It is recommended that you always specify the table address when using this option.

(2024) runtime sub-option “ivt” specifies an interrupt table (@ 0x*) that has not been defined (Driver)
The ivt sub-option to the `--RUNTIME` option was used to specify a IVT address, but this address has not been specified in the source code with any ISR. Check that the address in the option is correct, or check that the `base()` arguments to the `__interrupt()` specifier are specified and are correct.

```
--RUNTIME=+ivt:0x100
```

Oops -- is this the right address? Nothing in the source code uses this base address.

(2025) qualifier * on local variable “*” is not allowed and has been ignored (Parser)
Some qualifiers are not permitted with auto or local static variables. This message indicates that the indicated qualifier has been ignored with the named variable.

```
near int foobar; // oops -- auto variables cannot use near
```

(2026) variables qualified “*” are not supported for this device (Parser)
Some variable qualifiers are not permitted with some devices.

```
eeprom int serialNo; // oops -- can’t use eeprom with PIC18 devices
```

(2027) initialization of absolute variable “*” in * is not supported (Code Generator)
The variable indicated cannot be specified as absolute in the memory space.

```
eeprom char foobar __at(0x40) = 99; // oops - absolute can’t be eeprom
```

(2028) external declaration for identifier “*” doesn't indicate storage location (Code Generator)
The declaration for an external object (e.g., one defined in assembly code) has no storage specifiers to indicate the memory space in which it might reside. Code produced by the compiler which accesses it might fail. Use `const` or a bank specifier as required.

```
extern int tapCounter;  // oops - how does the compiler access this?
```

(2029) a function pointer cannot be used to hold the address of data (Parser)
A function pointer must only hold the addresses of function, not variables or objects.

```
int (*fp)(int);
int foobar;
fp = &foobar;  // oops - a variable’s address cannot be assigned
```

(2030) a data pointer cannot be used to hold the address of a function (Parser)
A data pointer (even a generic `void *` pointer) cannot be used to hold the address of a function.

```
void *gp;
int myFunc(int);
gp = foobar;  // oops - a function’s address cannot be assigned
```

(2033) recursively called function might clobber a static register it has allocated in expression (Code Generator)
The compiler has encountered a situation where a register is used by an expression that is defined in a function that is called recursively and that expression is part of a larger expression that requires this same function to be called. The register might be overwritten and the code may fail.

```
unsigned long fib_rec(unsigned long n)
{
    // the temporary result of the LHS call to fib_rec() might
    // store the result in a temp that is clobbered during the RHS
    // call to the same function
    return ((n > 1) ? (fib_rec(n-1) + fib_rec(n-2)) : n);
}
```

(2034) 24-bit floating-point types are not CCI compliant; use 32-bit setting for compliance (Parser)
The CCI does not permit the use of 24-bit floating point types. If you require compliance, use the `-no-short-float` and `-no-short-double` options, which will ensure the IEEE standard 32-bit floating-point type is used for `float` and `double` types.

(2035) use of sizeof() in preprocessor expressions is deprecated; use __SIZEOF_*__ macro to avoid this warning (Preprocessor)

The use of `sizeof()` in expressions that must be evaluated by the preprocess are no longer supported. Preprocessor macros defined by the compiler, such as `__SIZEOF_INT__`, can be used instead. This does not affect the C operator `sizeof()` which can be used in the usual way.

```
#if (sizeof(int) > 2)   // oops -- use (__SIZEOF_INT__ > 2) instead
```

(2036) use of @ is not compliant with CCI, use __at() instead (Parser)

The CCI does not permit the definition of absolute functions and objects that use the `@ address` construct. Instead, place `__at(address)` after the identifier in the definition.

```
int foobar @ 0x100;  // oops -- use __at(0x100) instead
```

(2037) short long integer types are not compliant with CCI (Parser)

The CCI does not permit use of the 3-byte `short long` type. Instead consider an equivalent `long int` type.

```
short long input;  // oops -- consider input to be long when using CCI
```

(2038) use of short long integer types is deprecated; use __int24 or __uint24 to avoid this warning (Parser)

The `short long` type specifiers has been replaced with the more portable `__int24` (replacing `short long`) and `__uint24` (replacing `unsigned short long`) types.

```
short long input; // oops -- use __int24 as the type for input
```

(2039) __int24 integer type is not compliant with CCI (Parser)

The CCI does not permit use of the 3-byte `__int24` type. Instead use the `long int` type.

```
__int24 input;  // oops -- use a long type when using CCI
```

(2040) __uint24 integer type is not compliant with CCI (Parser)

The CCI does not permit use of the 3-byte `__uint24` type. Instead use the `unsigned long int` type.

```
__uint24 input;  // oops -- use an unsigned long type when using CCI
```

(2041) missing argument after “*” (Driver)

The specified option requires an argument, but none was detected on the command line.

```
xc8-cc -mcpu=18f4520 -Wl,-Map main.c
```

Oops, the `-Map` option requires a map filename, e.g. `-Wl,-Map=proj.map`.

(2042) no target device specified; use the -mcpu option to specify a target device (Driver)

The driver was invoked without selecting what chip to build for. Running the driver with the `-mprint-devices` option will display a list of all chips that could be selected to build for.

```
xc8-cc main.c
```

Oops, use the `-mcpu` option to specify the device to build for.

(2043) target device was not recognized (Driver)

The top-level driver was not able to identify the family of device specified with the `-mcpu` option.

```
xc8-cc -mcpu=pic io.c
```

Oops, the device name must be exactly one of those shown by `-mprint-devices`.

(2044) unrecognized option "*" (Driver)
The option specified was not recognized by the top-level driver. The option in question will be passed further down the compiler tool chain, but this may cause errors or unexpected behavior.

(2045) could not find executable "*" (Driver)
The top-level driver was unable to locate the specified compiler tool in the usual locations. Ensure you have not moved files or directories inside the compiler install directory.

(2046) identifier length must be between * and *; using default length * (Driver)
The number of characters specified as the largest significant identifier length is illegal and the default length of 255 has been used.

```
-N=16
```

Oops, the identifier length must be between 32 and 255.

(2047) 24-bit floating point types are not supported when compiling in C99 (Driver)
The float and double types must be 32-bits wide when compiling for the C99 Standard. If you need 24-bit floating-point types, then you might be able to select the C90 compliant libraries (using the -mc90lib option) or you must compile for C90.

```
xc8-cc -mcpu=18f4520 -fshort-double main.c
```

Oops, you cannot use 24-bit `double` types with C99.

(2048) C language extension "*" is not supported and will be ignored (Driver)
The indicated language extension is not supported.

```
xc8-cc -mcpu=16f1937 -mext=iar main.c
```

Oops, that language extension is not supported.

(2049) C99 compliant libraries are currently not available for baseline or mid-range devices, or for enhanced mid-range devices using a reentrant stack; using C90 libraries (Driver)
At present, C99-compliant libraries are not available for all devices. The C90-compliant libraries can be used with these device while still building your source code to the C99 standard. Alternatively, you may choose to build to the C90 standard.

(2050) use of the -mcci option is deprecated; use -mext=cci to avoid this warning (Driver)
Always use the `-mext=cci` option to select the Common C Interface.

(2051) The current license does not permit the selected optimization level* (Driver)
This compiler's license does not allow the requested compiler operating mode.

```
xc8-cc -mcpu=18f4520 -Os main.c
```

Oops, you cannot select level 's' optimizations if this compiler is unlicensed.

(2052) The current license does not permit the selected optimization level and other levels are not permitted by the NOFALLBACK option (Driver)
This compiler's license does not allow the requested compiler operating mode. Since the `--nofallback` option is enabled, the compiler has produced this error and will not fall back to a lower optimization level. If you believe that you are entitled to use the requested optimizations, this error might indicate that your compiler is not be activated correctly.

(2053) function “*” is never called (Code Generator)
The specified inline function has never been called and will not generate code. This message differs to (520) in that the function specified is marked as inline. You may choose to disable this message for all inline functions, but allow message (520) to be issued for all other unused functions.

(2054) the language standard “*” is not supported; using C99 (Driver)
The language standard specified by the `-std` option is not supported by the compiler. The compiler will use the C99 standard instead.

```
xc8-cc -mcpu=12f510 -std=c11 main.c
```

Oops, you cannot select the C11 standard.

(2056) use of the -fmode option is deprecated; use -O to control optimizations and avoid this warning (Driver)
The compiler no longer uses both the mode and optimization selection to fully specify which optimizations are performed. All optimizations are now controllable via the optimization level, which is selectable using the compiler’s `-O` option. Unlicensed compilers, however, cannot use all levels.

(2057) The XC8 compiler installation appears to be corrupted. Please reinstall and try again (Driver)
The compiler has detected that something about the installation is not valid. This is most like due to compiler applications being deleted or moved.

(2058) function "*" cannot be inlined with code coverage enabled (Code Generator)
With the code coverage feature enabled, functions cannot be inlined. This advisary is just reminding you that the indicated function will not be inlined while code coverage is still in effect.

(2059) conflicting * register values found in Start Segment Address record (3) (Hexmate)
Hexmate will pass through any type 3 records in the Hex files being processed, but if there is any conflict in the values specified for the CS or IP registers in these records, it will flag this error.

(2060) CRC polynomial unspecified or set to 0 (Hexmate)
If you are calculating a CRC hash value using Hexmate and the polynomial value is zero, this warning will be triggered to indicate that you will be getting a trivial hash result. Typically this will occur if you have forgotten to set the polynomial value in the checksum option.

(2061) word width required when specifying reserve byte order hash (Hexmate)
If you are calculating a CRC reading data words in the Hex file in reverse order, you must specify a word width in bytes with Hexmate's `r` suboption to `-CK`. If you are using the compiler driver, this is specified using the `revword` suboption to `-mchecksum`.

(2062) word width must be * when specifying reserve byte order hash (Hexmate)
If you are calculating a CRC reading data words in the Hex file in reverse order, the word width can only be one of the values indicated in the message. This value is specified with Hexmate's `r` suboption to `-CK`. If you are using the compiler driver, this is specified using the `revword` suboption to `-mchecksum`.

(2063) * address must be a multiple of the word width when performing a reverse byte order hash (Hexmate)
If you are calculating a CRC reading data words in the Hex file in reverse order, the starting and ending addresses must be multiples of the word width specified in Hexmate's `-CK` option or the compiler's `-mchecksum` option.

(2064) PIC18 extended instruction cannot be used when the standard instruction set is selected (Assembler)
PIC18 assembly projects must be set up to use one of the standard or extended instructions sets. This message will be displayed if you have used an extended instruction without having enabled the extended instruction set using the `-misa` option.

(2065) offset out of range (Assembler)
The file register address for this extended PIC18 instruction is out of range.

```
clrf [200]  ; oops
; for extended instructions, the file operand must be less than, for example, 0x60
```

(2066) MESSG directive: * (Assembler)
This is the output message of the `MESSG` assembler directive.

(2067) ERROR directive: * (Assembler)
This is the output of the `ERROR` assembler directive.

(2068) use of the opt control "*" is deprecated; use the corresponding directive (Assembler)
The assembler controls of the form `OPT` *`CONTROL`* should no longer be used. Equivalent directives are available and can be formed by removing the `OPT` token from the control. Instead of using `OPT TITLE "My great project"`, for example, use `TITLE "My great project"`.

(2069) use of the radix option of the list control is deprecated; use the radix directive (Assembler)
The `LIST` assembler control previously allowed the input source to be specified using the `r` argument to the `LIST` option. This should no longer be used. Use the `RADIX` directive instead. Instead of using `OPT LIST r=hex`, for example, use `RADIX hex`.

(2070) device specified by the PROCESSOR directive conflicts with that set by the -mcpu option (Assembler)
The `-mcpu` driver option sets the target device being built for. The `PROCESSOR` directive may be used, if required, to ensure that an assembly source file is only ever built for the specified device. If there is a mismatch in the device specified by the option and the directive, this message will be displayed.

(2071) could not find record containing hash starting address 0x* (Hexmate)
Hexmate was asked to calculate a hash from data starting at an address that did not appear in the HEX file.

(2072) only SHA256 is currently supported (set width control to 256) (Hexmate)
The width suboption can only be set to 256 or -256 when selecting a SHA hash algorithm. Alternatively, the width suboption can be omitted entirely.

(2073) when compiling for C90, specifying an output file for dependencies is not supported and will be ignored (Driver)
Only the Clang front end can create a file containing dependencies.

```
xc8-cc -mcpu=18f4520 -MF depfile -std=c90 main.c
```

Oops, you cannot use the `-MF` option with `-std=c90`.

(2074) word size for byte skip with hash calculation must be * (Hexmate)
The argument to the `s` suboption of `-CK`, which indicates the size of the word in which bytes will be skipped or the purposes of calculating a hash value is not permitted.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s1
```

Oops, the argument to `s` must be larger than 1.

(2075) word size required when requesting byte skip with hash calculation (Hexmate)

An argument to the `s` suboption of `-CK` is required. It represents the word width in which bytes will be skipped for the purposes of calculating a hash value.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s.2
```

Oops, a number is required after the `s`, for example `s4.2`.

(2076) number of bytes for byte skip with hash calculation must be * (Hexmate)
The number of bytes to skip within each word is illegal.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s4.4
```

Oops, the number of bytes to skip must be less than 4, the skip word width, for example `s4.2`.

(2077) number of bytes required when requesting byte skip with hash calculation (Hexmate)
An argument following the `.` in the `s` suboption of `-CK` is required. It represents the number of bytes to skip in each word for the purposes of calculating a hash value.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s4.
```

Oops, a number is required after the `.` in the `s` argument, for example `s4.2`.

(2078) the number of hash bytes to which the trailing code is appended (*) must be no greater than the hash width (*) (Hexmate)
A trailing code has been requested to follow the specified number of bytes of the hash value, but this number is larger than that the entire hash.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021t00.4
```

Oops, if the hash value is only 2 bytes long, asking for a trailing code to be appended after every 4 bytes makes no sense. Instead try `t00.1`, for example, to append the code to each byte.

(2079) the hash width (*) must be a multiple of the number of hash bytes appended with a trailing code (*) (Hexmate)
A trailing code has been requested to follow the specified number of bytes of the hash value, but this number is not a multiple of the hash width.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-4p1021t00.3
```

Oops, if the hash value is 4 bytes long, asking for a trailing code to be appended after every 3 bytes makes no sense. Instead try `t00.2`, for example, to append the code to every two bytes of the hash.

(2080) WARN: * (Assembler)
This is a programmer-generated warning; there is an assembly directive causing a deliberate warning. Check the source code to determine if the warning should be investigated further. Consider removing the directive if it is no longer pertinent.

```
WARN "I have not yet confirmed that this is the correct threshold"
movlw 22
movwf threshold,c
```

(2081) the device architecture "*" does not match previously encountered object files "*" (Linker)
The linker was passed objects files that were built for different target devices. This is only likely to occur if you are using precompiled object files or libraries built from assembler code. Ensure all modules are built for the same device.

(2082) can't create temporary file (Driver)
The driver's attempt to create a temporary file failed. This could be due to a number of reasons, including disk space or permissions for the directory in which the system will create temporary files.

(2083) the current license does not permit the "*" feature (Driver)
A feature has been used that is not permitted by the compiler license. A PRO compiler license is required for some compiler features. Ensure that your compiler is installed correctly and has not expired if you believe that you have the appropriate license.

```
xc8-cc -mcpu=16f1937 -mchp-stack-usage main.c perip.c
```

Oops, the stack guidance feature, for example, requires a PRO compiler license to operate.

(2084) absolute object "*" has an address that lies within memory utilized by the compiler and will likely lead to code failure (Code Generator)
An absolute object (one declared using `__at()`) has been placed at an address that must be used by the compiler. The object will likely be corrupted by compiler generated code at runtime and lead to program failure.

```
int __at(0x0) myVar;  // oops - not a good address
```

(2085) the BANKISEL directive has no effect with the currently selected device and will be ignored (Assembler)
The `BANKISEL` assembler directive is required only for Baseline and Mid-range devices. It will be ignored for other devices, but check your device data sheet to ensure your code that performs indirect access of objects is valid for the selected device.

```
  BANKISEL myVar  ;oops - this device does not use separate indirect access bank bits
  movwf INDF
```

(2086) memory for a heap has been reserved in response to the detection of calls to malloc/calloc/realloc/free function(s) (Driver)
The compiler has detected calls to the standard dynamic memory allocation functions when the `auto` size value was set with the `-mheap` option. A compiler-determined amount of memory has been reserved for the heap.

(2087) memory for the * software stack has been reserved in response to the detection of functions built for reentrancy (Driver)
The compiler has detected functions built with the reentrant (software stack) model when the `auto` size value was set with the `-mstack` option. A compiler-determined amount of memory has been reserved for the indicated stack.

11. Implementation-Defined Behavior

This section indicates the compiler's choice of behavior where the C standard indicates that the behavior is implementation defined.

11.1 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)).

11.2 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: function()`
	`linenumber: source line`
	`^ (ID) message (warning)`
	or
	`filename: linenumber: (ID) message (warning)`
	where *filename* is the name of the file that contains the code (or empty if no particular file is relevant); *linenumber* is the line number of the code (or 0 if no line number is relevant); *ID* is a unique number that identifies the message; and *message* is the diagnostic message itself.

"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.

11.3 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)."	
	Multi-byte characters are not supported in source files.
"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)."	
	A soft reset implemented by a branch to the reset vector location.
"An alternative manner in which the `main` function may be defined (5.1.2.2.1)."	
	`void main (void);`

...........continued

ISO Standard	Implementation
"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.

11.4 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.

11.5 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)."	

...........continued

ISO Standard	Implementation
	The value of the `char` object is the 8-bit binary representation of the character in the source character set. That is, no translation is done.
“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, `unsigned char` is functionally equivalent to plain `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.

11.6 Integers

ISO Standard	Implementation
“Any extended integer types that exist in the implementation (C99 6.2.5).”	
	The `__bit` keyword designates a single-bit integer type. 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).”	

...........continued	
ISO Standard	**Implementation**
	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.

11.7 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)."	

...........continued

ISO Standard	Implementation
	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).”	
	The exception is not raised.

11.8 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).”	
	When converting an integer to a pointer variable, if the pointer variable throughout the entire program is only assigned the addresses of objects in data memory or is only assigned the addresses of objects in program memory, the integer address is copied without modification into the pointer variable. If a pointer variable throughout the entire program is assigned addresses of objects in data memory and also addresses of objects in program memory, then the MSb of the integer value will be set if it is explicitly cast to a pointer to const type; otherwise the MSb is not set. The remaining bits of the integer are assigned to the pointer variable without modification. When converting a pointer to an integer, the value held by the pointer is assigned to the integer without modification. The usual integer truncation applies if the integer is larger than the size of the pointer.
“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.

11.9 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 specifier has no effect.
“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.

11.10 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. Signed integer bit-fields are not supported.
"Allowable bit-field types other than `_Bool`, `signed int`, and `unsigned int` (C99 6.7.2.1)."	
	The `signed` and `unsigned char` type is allowed.
"Whether a bit-field can straddle a storage unit boundary (C90 6.5.2.1, C99 6.7.2.1)."	
	A bit-field cannot straddle a storage unit. Any bit-field that would straddle a storage unit will be moved to the LSb position in a new 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)."	
	The type chosen to represent an enumerated type depends on the enumerated values. A signed type is chosen if any value is negative; unsigned otherwise. If a `char` type is sufficient to hold the range of values, then this type is chosen; otherwise, an `int` type is chosen. Enumerated values must fit within an `int` type and will be truncated if this is not the case.

11.11 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.

11.12 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.

...........continued

ISO Standard	Implementation
"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`*`>/pic/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.

11.13 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`*`)`
"The representation of floating-point exception flags stored by the `fegetexceptflag` function (7.6.2.2)."	
	Unimplemented.
"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)."	

...........continued

ISO Standard	Implementation
	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)."	
	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.

...........continued

ISO Standard	Implementation
"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)."	
	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)."	

...........continued

ISO Standard	Implementation
	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)."	
	These functions are not implemented.
"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.

11.14 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)."	
	The type of the result is equivalent to unsigned int.

12. Document Revision History

Revision A (March 2018)

- Initial release of this document, adapted from the MPLAB® XC8 C Compiler User's Guide, DS 50002053.

Revision B (March 2019)

- Clarified the operation of the `-mcodeoffset` option when using devices with vectored interrupts
- Added information on the new code coverage feature
- Updated descriptions of the compiler operating modes and the optimization level control, `-O`
- Added descriptions and screen captures of the MPLAB X IDE project property dialogs corresponding to the compiler command-line options
- Updated information pertaining to `long long` type support
- Corrected typos and errors in example code sequences
- Clarified usage of the `__at()` construct
- Miscellaneous corrections and improvements

Revision C (February 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
- Added information on new `movff`-related errata
- Updated information reflecting new behavior of assembler branch instruction transformations
- Documented changes relating to Hexmate's find (and replace) command syntax
- Detailed changes that allow object and p-code modules to co-exist inside library archives
- Added SHA256 suboption to memory summary option
- Indicated that the `printf`-family of functions can now be replaced by a user-defined version
- Added new assembler `CONFIG` directive
- Detailed changes relating to assembler controls, which now are treated like directives
- Expanded information on use of the device family pack (DFP) option
- Miscellaneous corrections and improvements

Revision D (February 2021)

- Added information on new stack guidance feature
- Clarified positioning of `const`-qualifed objects
- Added information on the new Hexmate hash value options
- Expanded the description of some existing Hexmate features
- Added information relating to use of `printf()` in library examples
- Described the updated `-x` driver option
- Added `-mcmacros` and `-mclink` driver options
- Updated screen captures of MPLAB X IDE project properties dialogs
- Clarified use of case ranges inside switch statements
- Added additional information relating to interrupts for PIC18 IVT devices operating in legacy mode
- Expanded description of status register preservation feature
- Clarified use of the assembler `DABS` and `DLABS` directive
- Added new assembler `WARN` and `DEBUG_SOURCE` directives

Revision E (August 2021)

- Removed standard C library functions; these are now described in a separate *Microchip Universal Standard Library Reference Guide* document
- Added `-fcacheconst` option and description of new constant caching feature
- Added `-mheap` option and description of the heap and dynamic memory allocation feature
- Updated information relating to the operation of the `-mstack` option.
- Updated `-mdebugger` debugger option to reflect support for generation 4 debugger tools
- Updated screen captures of latest MPLAB X IDE Project Properties dialog relating to Analysis options
- Added information on new `BANKISEL` directive
- Added information on previously undocumented `FILE` and `LINE` assembler directives
- Added description of string support for the `DB`, `DW`, and `DDW` assembler directives
- Added updated information on Hexmate's handling of INHX16 format HEX files
- Added new `O` suboption to Hexmate's `-CK` option which requests the final hash result be XORed with a the specified value

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

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

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with

Trademarks

ISBN: 978-1-5224-8838-5

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

Australia - Sydney
Tel: 61-2-9868-6733

China - Beijing
Tel: 86-10-8569-7000

China - Chengdu
Tel: 86-28-8665-5511

China - Chongqing
Tel: 86-23-8980-9588

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115

China - Hong Kong SAR
Tel: 852-2943-5100

China - Nanjing
Tel: 86-25-8473-2460

China - Qingdao
Tel: 86-532-8502-7355

China - Shanghai
Tel: 86-21-3326-8000

China - Shenyang
Tel: 86-24-2334-2829

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

China - Wuhan
Tel: 86-27-5980-5300

China - Xian
Tel: 86-29-8833-7252

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444

India - New Delhi
Tel: 91-11-4160-8631

India - Pune
Tel: 91-20-4121-0141

Japan - Osaka
Tel: 81-6-6152-7160

Japan - Tokyo
Tel: 81-3-6880- 3770

Korea - Daegu
Tel: 82-53-744-4301

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

Malaysia - Penang
Tel: 60-4-227-8870

Philippines - Manila
Tel: 63-2-634-9065

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600

Thailand - Bangkok
Tel: 66-2-694-1351

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

Finland - Espoo
Tel: 358-9-4520-820

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-72400

Germany - Karlsruhe
Tel: 49-721-625370

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra'anana
Tel: 972-9-744-7705

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Padova
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Norway - Trondheim
Tel: 47-72884388

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820