



MPLAB[®] C18 to XC8 C Compiler Migration Guide

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.

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in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates.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 your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, dsPIC, FlashFlex, KEELQ, KEELQ logo, MPLAB, PIC, PICmicro, PICSTART, PIC³² logo, rfPIC, SST, SST Logo, SuperFlash and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Hampshire, HI-TECH C, Linear Active Thermistor, MTP, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.


Analog-for-the-Digital Age, Application Maestro, BodyCom, chipKIT, chipKIT logo, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, HI-TIDE, In-Circuit Serial Programming, ICSP, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, mTouch, Omniscent Code Generation, PICC, PICC-18, PICDEM, PICDEM.net, PICkit, PICtail, REAL ICE, rLAB, Select Mode, SQI, Serial Quad I/O, Total Endurance, TSHARC, UniWinDriver, WiperLock, ZENA and Z-Scale are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

GestIC and ULPP are registered trademarks of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2013, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.

ISBN: 978-1-62076-984-3

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
= ISO/TS 16949 =

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Table of Contents

Preface	5
Chapter 1. Migration Overview	
1.1 Introduction	10
1.2 Using C18 Compatibility Mode	12
1.3 Migrating Projects to MPLAB XC8	13
Chapter 2. Invoking the Compiler	
2.1 Introduction	15
2.2 General Compiler Usage	15
2.3 Driver Options	18
2.4 MPLINK Options	22
Chapter 3. Language Features	
3.1 Introduction	23
3.2 Operating Modes	24
3.3 Memory Models	24
3.4 Integer Promotions	24
3.5 Device-Specific Information	25
3.6 Data Types and Limits	27
3.7 Size Limitations	28
3.8 Storage Classes	28
3.9 Storage Qualifiers	29
3.10 Pointer Storage Qualifiers	30
3.11 Function Variants	31
3.12 Structures and Unions	32
3.13 Interrupts	32
3.14 Locating Objects	34
3.15 Function Reentrancy and Calling Conventions	36
3.16 The Runtime Startup Code	40
3.17 Register Usage	41
3.18 Preprocessing	42
3.19 C and Assembly	44
3.20 Linking	47
Glossary	49
Worldwide Sales and Service	72

MPLAB® C18 to XC8 C Compiler Migration Guide

NOTES:



MPLAB® C18 TO XC8 C COMPILER MIGRATION GUIDE

Preface

NOTICE TO CUSTOMERS

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

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

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

INTRODUCTION

This chapter contains general information that will be useful to know before using the MPLAB® C18 to XC8 C Compiler Migration Guide. Items discussed in this chapter include:

- Document Layout
- Conventions Used in this Guide
- Recommended Reading
- The Microchip Web Site
- Development Systems Customer Change Notification Service
- Customer Support
- Document Revision History

DOCUMENT LAYOUT

This document describes how to migrate C source code from the MPLAB C18 Compiler to the MPLAB XC8 C Compiler. This guide is organized as follows:

- **Chapter 1. Migration Overview**
- **Chapter 2. Invoking the Compiler**
- **Chapter 3. Language Features**
- **Glossary**

MPLAB® C18 to XC8 C Compiler Migration Guide

CONVENTIONS USED IN THIS GUIDE

This manual uses the following documentation conventions:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	<i>MPLAB® IDE User's Guide</i>
	Emphasized text	...is the <i>only</i> 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><i>File>Save</i></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	<i>file.o</i> , where <i>file</i> can be any valid filename
Square brackets []	Optional arguments	mcc18 [options] <i>file</i> [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) { ... }

RECOMMENDED READING

This user's guide describes migrating MPLAB C18 C projects to the XC8 C compiler. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Readme Files

For the latest information on using other tools, read the tool-specific Readme files in the Readmes subdirectory of the MPLAB IDE installation directory. The Readme files contain update information and known issues that may not be included in this migration guide.

MPLAB X IDE User's Guide (DS52027)

A guide to using the MPLAB X Integrated Development Environment (IDE).

MPLAB XC8 C Compiler User's Guide (DS50002053)

A guide to using the MPLAB XC8 C Compiler.

THE MICROCHIP WEB SITE

Microchip provides online support via our web site at www.microchip.com. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **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 consultant 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

MPLAB® C18 to XC8 C Compiler Migration Guide

DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

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

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

The Development Systems product group categories are:

- **Compilers** – The latest information on Microchip C compilers and other language tools. These include the MPLAB C18 and MPLAB C30 C compilers; MPASM™ and MPLAB ASM30 assemblers; MPLINK™ and MPLAB LINK30 object linkers; and MPLIB™ and MPLAB LIB30 object librarians.
- **Emulators** – The latest information on Microchip in-circuit emulators. This includes the MPLAB ICE 2000 and MPLAB ICE 4000.
- **In-Circuit Debuggers** – The latest information on the Microchip in-circuit debugger, MPLAB ICD 2.
- **MPLAB® IDE** – The latest information on Microchip MPLAB IDE, the Windows® Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB SIM simulator, MPLAB IDE Project Manager and general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the MPLAB PM3 and PRO MATE® II device programmers and the PICSTART® Plus and PICKit™ 1 development programmers.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: <http://support.microchip.com>

DOCUMENT REVISION HISTORY

Revision A (July 2013)

Initial release of this document.

Chapter 1. Migration Overview

1.1 INTRODUCTION

This documentation looks at the differences between the MPLAB® C Compiler for PIC18 MCUs (formerly, and referred to here as, MPLAB C18) and the MPLAB XC8 C Compiler, and how you might migrate C source code and compiler options that were tailored to MPLAB C18 over to MPLAB XC8.

This guide will be useful for those porting projects to the MPLAB XC8 compiler, as well as those programmers familiar with MPLAB C18 syntax and operation, who wish to come up to speed quickly with MPLAB XC8 for new projects.

This chapter introduces the following overviews.

- Using C18 Compatibility Mode
- Migrating Projects to MPLAB XC8

Before the release of XC8, MPLAB C18 had been the only PIC18 C compiler offered by Microchip. It is ANSI compliant and includes separate assembler and linker applications. Peripheral libraries are supplied to speed development. The MPLAB C18 compiler is being phased out, and no significant new development is planned for future releases.

The MPLAB XC8 C compiler has replaced the MPLAB C18. This replacement compiler is also ANSI compliant, includes a separate assembler and linker, and the same peripheral library is supplied.

For existing projects designed for MPLAB C18 and future code development, you have several options:

- Continue to use the deprecated MPLAB C18 compiler.
- Compile in C18 compatibility mode with the MPLAB XC8 compiler.¹
- Migrate project and code to native MPLAB XC8 settings and syntax.

As MPLAB C18 will not be further developed, it is recommended that you only continue to use this compiler for complete projects that are not being actively maintained. You might also use this compiler if you need to exactly replicate a previous build output.

The MPLAB XC8 compatibility mode will be useful if you are unable to use the MPLAB C18 compiler but cannot justify the time spent to migrate code over to the native MPLAB XC8 syntax.

Microchip recommends that, if possible, code be migrated to MPLAB XC8 syntax. This will guarantee support for projects into the future.

¹. See the note box in **Section 1.2 “Using C18 Compatibility Mode”** concerning the availability of this mode.

MPLAB® C18 to XC8 C Compiler Migration Guide

This guide is specifically targeted at code migration and looks at the specific source code and compiler option changes that will be required to build with MPLAB XC8. In each section you will find general discussion of the differences between the compilers; porting guidelines, which summarize the possible changes required; and tips for moving forward with MPLAB XC8 code development.

- Note:**
1. This guide assumes you are familiar with MPLAB C18 syntax and features. This guide is primarily aimed at introducing the equivalent or nearest MPLAB XC8 syntax or feature, but does not attempt to explain every aspect of these features. It is important that you check the MPLAB XC8 user's guide for full information about the features offered by this compiler.
 2. The MPLAB C18 compiler can compile functions as being reentrant. This functionality will be introduced in a future version of MPLAB XC8. Ensure that you are using a version of MPLAB XC8 that supports reentrancy, if your project requires this functionality.
 3. In addition to PIC18 devices, MPLAB XC8 can compile for all Microchip 8-bit devices, including devices in the baseline and mid-range families. This guide deals only with PIC18 projects. The features described in this guide may not be relevant for other devices.

1.2 USING C18 COMPATIBILITY MODE

If you do not want to migrate code to the native MPLAB XC8 format, you can use the MPLAB XC8 compiler in a C18 compatibility mode. In this mode, it will internally translate many of the non-standard extensions used by MPLAB C18 source code. This compatibility even extends to MPLAB C18 compiler options, by the use of a fake `mcc18` driver. MPLINK linker scripts can be read by MPLAB XC8 when it is operating in compatibility mode.

Note: The C18 compatibility mode is, at the time of this document preparation, only a beta feature. Full compatibility mode operation is planned for a future compiler release. Check the release notes of subsequent compilers to confirm the availability of this feature. Not all MPLAB C18 features are available in the beta implementation of this mode.

Microchip recommends that you use compatibility mode to quickly rebuild code using the MPLAB XC8 compiler or to experiment with this tool. You may also use compatibility mode as you gain familiarity with the MPLAB XC8 language and environment.

To use compatibility mode, you simply need to follow these steps:

- Open your existing MPLAB C18 project in MPLAB IDE¹.
- Do *not* change the project toolsuite; continue to use the MPLAB C18 toolsuite.
- Change the path for the MPLAB C18 driver application from being the `mcc18.exe` application installed by MPLAB C18 to the identically named fake application found in your MPLAB XC8 compiler's `bin` directory. Also change the path to the `mplink.exe` and `mplib.exe` applications in the same way. (There is no support for code built with the `mpasmwin.exe` application.)
- Rebuild your code in the usual way.

When you build the project, the IDE will think it is using MPLAB C18, but it will instead be using the fake applications provided by MPLAB XC8. When run, these applications will transcribe the MPLAB C18 options specified in your project to equivalent MPLAB XC8 options, run the MPLAB XC8 compiler, and ensure that the compiler is running in compatibility mode.

If you are using batch or make files, instead of MPLAB IDE, to build your MPLAB C18 projects, you can edit these files so that they execute the fake applications provided by MPLAB XC8.

Note: Do not attempt to explicitly run the MPLAB XC8 compiler (`xc8`) when compiling MPLAB C18 projects.
Do not manually set any option to indicate C18 compatibility mode.
Only use the fake applications provided by MPLAB XC8, invoked from the command line, batch or make files; or from MPLAB IDE.

¹At the time of writing, only MPLAB IDE version 8 can be used to compile legacy projects in C18 compatibility mode.

1.3 MIGRATING PROJECTS TO MPLAB XC8

Migrating MPLAB C18 code over to the native MPLAB XC8 syntax ensures that your project is always compatible with the latest tools offered by Microchip. The amount of effort required largely depends on how the code was written and which MPLAB C18 features were employed.

1.3.1 ANSI Compliant Code

Any code which is ANSI compliant can be migrated to MPLAB XC8 with a minimal effort.

Note that certain aspects of the ANSI Standard permit code to behave in an implementation-defined way. The behavior of implementation-defined code is clearly listed in both compilers' user's guides. You should ensure that your source code does not make any assumptions as to how code with implementation-defined behavior will execute.

The following issues are situations in which there are differences in the implementation-defined behavior between code that is compiled with MPLAB C18 and code compiled with MPLAB XC8.

The type of a plain `char`.

This is fully discussed in **Section 3.6 “Data Types and Limits”**.

The sign of the remainder of integer divisions.

This is the same as the sign of the dividend for MPLAB XC8, and the sign of the quotient for MPLAB C18.

Sign extension of right shifts of negative signed values.

Sign extension occurs when using MPLAB XC8; it does not for MPLAB C18.

Rounding of floating-point values.

MPLAB C18 always rounds floating-point results to the nearest value. MPLAB XC8 also rounds results to the nearest value, except when a floating-point number is converted to a narrower floating-point number. In that case, the rounding is to the smaller floating-point value.

Support for signed bit-fields.

Structure and bit-field differences are fully described in **Section 3.12 “Structures and Unions”**.

Note that the MPLAB C18 compiler does not apply integer promotion by default. If the option to enable this has not been selected in your project, then the results of integer expressions in which the type of any operand is smaller than an `int` may be different. This is discussed further in **Section 3.4 “Integer Promotions”**.

1.3.2 Non-Standard Extensions

Both the MPLAB C18 and XC8 compilers have non-standard extensions to assist with programming in an embedded environment. Since the syntax and operation of these extensions are different, they will require the most amount of attention to migrate. Non-standard extensions are discussed throughout **Chapter 3. “Language Features”**.

All the latest MPLAB XC compilers (XC8, XC16 and XC32) provide a common C interface (CCI), which allows for a higher degree of portability between these compilers. It standardizes the syntax of some of the non-standard extensions and further refines implementation-defined behavior.

You might consider using the alternate syntax provided by the CCI when you compile in MPLAB XC8. Note that CCI is a compiler mode that must be enabled. A chapter in the *MPLAB XC8 C Compiler User’s Guide* (DS50002053) describes this interface, and the mapping from the native MPLAB XC8 syntax and the CCI syntax.

1.3.3 Options and Linker Scripts

In addition to the source code itself, a project may utilize compiler options and/or linker scripts to achieve the desired program operation. These also need to be transcribed to equivalent MPLAB XC8 code or options, as they are not accepted outside of C18 compatibility mode.

Compiler options are described in **Section Chapter 2. “Invoking the Compiler”**, which also lists additional, related MPLAB XC8 options.

Linker scripts have no direct counterpart in MPLAB XC8. Instead, linker options, which can be controlled from the compiler driver, are used. Linker options work in conjunction with assembler directives that define sections (or psects) in assembly code. This is described in **Section 3.20 “Linking”**.

1.3.4 Assembly Code

Assembly code is inherently non-portable, even between assemblers for the same device. Assembly code migration is beyond the scope of this document, but **Section 3.19 “C and Assembly”** gives brief suggestions as to the changes that might be necessary to ensure that assembly code builds and operates as expected.

1.3.5 Object and Library Files

The object and library files used by MPLAB C18 are not compatible with MPLAB XC8. MPLAB XC8 comes with its own set of C standard libraries, which are linked in by default; so, similar library functions are available (see also **Section 3.11 “Function Variants”**).

It will be necessary to obtain the source code for any object files that are used, and include this into the MPLAB XC8 project. Indeed, it is not recommended that object files (even those in MPLAB XC8 format), are used as an input file for this compiler. Use C source files or p-code library files whenever possible.

If no equivalent library is available for MPLAB XC8, you will need to obtain the source code for these routines, as well. Note that the peripheral library supplied with MPLAB C18 is included with MPLAB XC8. This library is linked in by default (see the `--RUNTIME` option in the MPLAB XC8 user’s guide).

MPLAB® C18 to XC8 C Compiler Migration Guide

NOTES:

Chapter 2. Invoking the Compiler

2.1 INTRODUCTION

This chapter looks at the changes that will be made to command-line compiler options when migrating from MPLAB C18 to MPLAB XC8.

It explains how MPLAB XC8 should be executed and the differences between this compiler application and its MPLAB C18 counterpart.

The following topics are discussed.

- General Compiler Usage
- Driver Options
- MPLINK Options

2.2 GENERAL COMPILER USAGE

The MPLAB C18 and MPLAB XC8 C compilers differ slightly in how they are run on the command-line.

If you normally use MPLAB X IDE to perform builds, many of the following translations are handled by the IDE when you move to the new compiler toolsuite.

2.2.1 Compiler Applications

Both compilers employ a command-line driver that is used to invoke the appropriate applications when compiling. However, MPLAB C18 requires that the linker (`mplink`) is run explicitly, whereas the MPLAB XC8 driver can run the linker implicitly. It is recommended that you do not attempt to run the linker any other way. The names of the MPLAB C18 applications that are run when performing compilation are given in Table 2-1, along with the MPLAB XC8 equivalent.

TABLE 2-1: APPLICATION NAMES

Task	MPLAB C18 application	MPLAB XC8 application
C compilation (driver)	<code>mcc18</code>	<code>xc8</code>
Linking	<code>mplink</code>	<code>xc8</code>
Library creation	<code>mplib</code>	<code>xc8</code>

Not only can the `xc8` driver invoke all the compilation steps, it can usually do so in one command. For example, to compile and link a trivial source file using MPLAB C18, you might use the commands:

```
mcc18.exe -p=18F4410 test.c
mplink.exe /p18F4410 /u_CRUNTIME test.o
```

This can be performed using just one command with MPLAB XC8:

```
xc8 --chip=18f4410 test.c
```

If you are using MPLAB X IDE, it is aware of how both compilers operate and always uses the appropriate command lines.

2.2.2 Compilation Sequence

The MPLAB XC8 compiler uses a novel code generation technology called Omniscient Code Generation (OCG). Unlike conventional compilers, the OCG compiler brings together all parts of the C source code (including library code) at the code generation stage of compilation, not at the link stage.

This different compilation sequence means that the temporary, intermediate file used by the compiler is not an object file, but is the file produced by the parser application. These are known as p-code files.

Although object files are still used by MPLAB XC8, they should not be used as the intermediate file for C source code. You will almost certainly trigger an error if you attempt to generate these from C code. Object files should only be generated from assembly source code. Similarly, libraries produced from C source should use the p-code library format (`.lpp` files), rather than the convention object code libraries (`.lib`).

If you wish to perform multi-step compilation of your source files, you must use the `--PASS1` option to stop compilation after parsing, leaving the intermediate file. The source file used in the above example could be compiled using an intermediate file as follows.

```
xc8 --chip=18f4410 --pass1 test.c
xc8 --chip=18f4410 test.p1
```

Note that the driver (`xc8`) is used for both compilation steps. This multi-step compilation is what MPLAB X IDE uses to compile C source files when the MPLAB XC8 toolsuite is selected.

2.2.3 Command-line Format

The layout of the command-line arguments to the driver are similar. The drivers use special options to control the compilation process, and one or more file names indicate the files that are to be compiled.

The relative position of the files and options on the command line is not important for either compiler — you may issue options before or after the names of files. In terms of the generated code, file positions may affect the memory addresses allocated to variables and functions.

Some operating systems use case sensitive file systems, so always use the correct case when specifying the names of files.

The general form of compilation under MPLAB XC8 is usually:

```
xc8 --chip=device [options] files
```

The options are not case sensitive; however, some option arguments may specify case-sensitive information. A summary of the equivalent MPLAB XC8 compiler options are shown in **Section 2.3 “Driver Options”**.

The default behavior of the MPLAB C18 compiler is to produce a COFF file output (`.o` file extension). Modern MPLAB XC8 compilers (v1.20 and after) can produce an ELF/DWARF output (`.elf` extension) if required, but the default output format is also the COFF file. ELF files allow for better debugging and should be used whenever possible.

Note that the options to `mplink` differ in format to those used by the MPLAB C18 driver, `mcc18`. This application has its own set of command line options, described in the MPLINK User's Guide and discussed in **Section 2.4 “MPLINK Options”**.

2.2.4 Compiler Usage Porting Guidelines

The following guidelines only apply if you are not using MPLAB X IDE to build projects, i.e., you are using make or batch files.

- Change the name of the compiler driver, linker and librarian to `xc8`.
- If intermediate files are to be produced from C source code, use the `--PASS1` option to generate these p-code files. The final 'link' step will read files with a `.p1` extension.

2.3 DRIVER OPTIONS

This section looks at the actual compiler options and maps the MPLAB C18 options to the nearest MPLAB XC8 equivalent. A summary of all MPLAB C18 options is provided, followed by a general discussion of the more important options.

2.3.1 Option Summary

The columns in Table 2-2 list all of the command-line options for the MPLAB C18 compiler alongside any equivalent or similar options that are used in MPLAB XC8.

While some of the MPLAB XC8 options are a direct replacement, others simply work in a similar way. Check the *MPLAB XC8 C Compiler User's Guide* (DS500002053) for full information on the options shown.

TABLE 2-2: SUMMARY OF EQUIVALENT MPLAB XC8 DRIVER OPTIONS

MPLAB C18 Option	Nearest MPLAB XC8 Equivalent
-?, --help	--HELP
-I=path	-I=path
-fo=name	-O
-fe=name	-E
-z, --inc=name	n/a
-k	n/a
-ls	n/a (large software stack is always supported)
-ms	n/a
-ml	n/a
-O, -O+	--OPT=all
-O-	--OPT=none
-Oi+	n/a (integer promotion always enabled)
-Oi-	n/a (integer promotion always enabled)
-Om+	n/a (string merging always enabled)
-Om-	n/a (string merging always enabled)
-On+	n/a (optimization always enabled)
-On-	n/a (optimization always enabled)
-Ou+	n/a (optimization always enabled)
-Ou-	n/a (optimization always enabled)
-Os+	n/a (optimization always enabled)
-Os-	n/a (optimization always enabled)
-Ot+	n/a (optimization always enabled)
-Ot-	n/a (optimization always enabled)
-Ob+	n/a (optimization always enabled)
-Ob-	n/a (optimization always enabled)
-sca	n/a (auto locals always enabled)
-scs	n/a (auto locals always enabled)
-sco	n/a (auto locals always enabled)
-Od+	n/a (optimization always enabled)
-Od-	n/a (optimization always enabled)
-Opa+	--OPT=+space
-Opa-	--OPT=+speed
-pa=repeat count	n/a (optimization always enabled)
-Op+	n/a (optimization always enabled)

TABLE 2-2: SUMMARY OF EQUIVALENT MPLAB XC8 DRIVER OPTIONS

MPLAB C18 Option	Nearest MPLAB XC8 Equivalent
-Op-	n/a (optimization always enabled)
-Or+	n/a (optimization always enabled)
-Or-	n/a (optimization always enabled)
-Oa+	n/a (optimization always enabled)
-Oa-	n/a (optimization always enabled)
-Ow+	n/a (optimization always enabled)
-Ow-	n/a (optimization always enabled)
-p=processor	--CHIP=processor
--extended	n/a (standard instruction set always used)
--no-extended	n/a (standard instruction set always used)
-Dmacro[=text]	-Dmacro[=text]
-w={ 1 2 3 }	--WARN=-9 to 9
-nw=n	--MSGDISABLE=list
-verbose	-V
--help-message-list	n/a
--help-message-all	n/a
--help-message=n	n/a
--help-config	n/a
-v	--VER

If you are using MPLAB IDE to compile, the MPLAB XC8 compiler toolsuite in MPLAB IDE will allow graphical access to the MPLAB XC8 compiler options. Note that the MPLAB XC8 user's guide maps the command-line options to the corresponding widgets for both the MPLAB IDE v8 Build options and the MPLAB X IDE Project Properties dialogs.

TIPS: There are many other MPLAB XC8 options than just those shown in the table. These options have no MPLAB C18 equivalent. Check the *MPLAB XC8 C Compiler User's Guide* (DS500002053) for more information.

2.3.2 Basic Compiler Options

The name of the target device should always be specified on the command-line when using MPLAB XC8. If this information is missing when using MPLAB C18, the compiler uses a default device. When using MPLAB XC8, a target device must always be specified, and an error will be generated if this is missing.

MPLAB XC8 does not support the PIC18 extended instruction set. If you have used the --extended option in your MPLAB C18 project, you will need to ensure that you do not have hand-written assembly code that uses this instruction set.

The MPLAB C18 option -FO, which permits renaming of the output file, performs a similar task to the MPLAB XC8 option -O, although this latter option also allows for specification of an output directory, as well as file renaming.

TIPS: Further fundamental MPLAB XC8 options include --EMI, --CCI and --MODE.

2.3.3 Optimization Options

There is far less control over compiler optimizations when using the MPLAB XC8 compiler compared to when using MPLAB C18.

Many of the MPLAB XC8 optimizations at the C level are performed as part of the fundamental code processing and cannot be disabled. Since these optimizations are performed before conversion to assembly, they have little effect on debugging, and so there is little need to disable them. Assembly-level optimizations, which can adversely impact debugging, can, however, be completely disabled.

A single MPLAB XC8 option `--OPT`, and its suboptions, enable or disable the available optimizations. As with MPLAB C18, all compiler optimizations are enabled by default when using MPLAB XC8.

2.3.4 Preprocessor Commands

As with MPLAB C18, preprocessor macros can be defined on the command line as well as in code, using the `#define` directive. In both compilers, this option is `-D`.

TIPS: The MPLAB XC8 compiler has an additional option that undefines macros, `-U`.

The paths searched for include files by MPLAB C18 are specified by an environment variable `MCC_INCLUDE` and the `-I` option. The MPLAB XC8 compiler also has an equivalent `-I` option, but no environment variables are used to specify the paths. Add paths specified by the `MCC_INCLUDE` environment variable using the MPLAB XC8 option `-I`, instead.

TIPS: Other MPLAB XC8 options related to preprocessing include `-P`, `--PRE` and `--SCANDEP`.

2.3.5 Diagnostics

Messages are issued by compilers to indicate invalid or suspicious code. A complete list of all messages is shown in the Error and Warning Messages Appendix of the *MPLAB XC8 C Compiler User's Guide* (DS500002053).

The MPLAB XC8 compiler uses a centralized messaging system that is common to all compiler applications. Message have several categories, which are similar to those in MPLAB C18, but with the addition of fatal error messages. The following message categories are used by MPLAB XC8.

Advisory Messages –

to convey information regarding a situation the compiler has encountered, or some action the compiler is about to take.

Warning Messages –

to indicate source code or some other situation that can be compiled, but is unusual and may lead to a runtime failure of the code.

Error Messages –

to indicate source code that is illegal, or that compilation of this code cannot take place.

Fatal Error Messages –

to indicate a situation that prevents compilation from proceeding and which stops the compilation process immediately.

As with MPLAB C18, MPLAB XC8 has driver options that can be used to suppress warning messages by their level. The MPLAB XC8 warnings are assigned a level ranging from -9 to +9, as opposed to the 1, 2 and 3 levels assigned to MPLAB C18 messages. The higher the level, the more important the warning.

The levels assigned to messages can be obtained from the message description file (`en_msg.txt`) that is located in the `DAT` directory of the MPLAB XC8 compiler.

The MPLAB XC8 option `--WARN` suppresses warning messages below the indicated threshold level. The default level is 0. This option performs a similar task to the MPLAB C18 option `-w`.

The MPLAB XC8 compiler also allows messages to be disabled via their identification message number. The message's number is usually printed with the message text. They can also be found in the Error and Warning Messages Appendix of the MPLAB XC8 user's guide. The MPLAB XC8 option `--MSGDISABLE` performs the same task as the `-NW` option supplied with MPLAB C18. The `--MSGDISABLE` option, however, can take a comma-separated list of message numbers, not just a single number, as with MPLAB C18.

TIPS: The MPLAB XC8 compiler goes further, in that it also allows the control of messages via the use of pragmas in the source code. See `#pragma warning` in the MPLAB XC8 user's guide. The pragma allows control of messages issued for one or more lines of code. Other MPLAB XC8 options that are related to messaging include: `--MSGFORMAT`, `--ERRFORMAT`, `--WARNFORMAT`, `--ERRORS`, and `--LANG`.

2.3.6 Driver Options Porting Guidelines

The following guidelines will help you select the compiler options that are correct for your project.

- If you are using an IDE, build your MPLAB C18 project and take note of the driver options being used in the Build or Output window.
- Find the equivalent MPLAB XC8 options in the summary listed in this guide.
- Use the MPLAB XC8 user's guide to find the IDE controls that correspond to these options, if required.
- Browse the other MPLAB XC8 options available and select those appropriate.

MPLAB® C18 to XC8 C Compiler Migration Guide

2.4 MPLINK OPTIONS

As indicated in **Section 2.3 “Driver Options”**, the MPLAB XC8 linker (`hlink`) is not run explicitly. The most commonly used linker options have counterparts in the MPLAB XC8 driver, `xc8`. The MPLINK options and the equivalent MPLAB XC8 driver options are shown in Table 2-3.

TABLE 2-3:

MPLINK Option	Nearest MPLAB XC8 Equivalent
<code>/a hexformat</code>	<code>--RUNTIME=download</code>
<code>/d</code>	Do not use <code>--ASMLIST</code>
<code>/g</code>	n/a
<code>/h, /?</code>	<code>--HELP</code>
<code>/i</code>	<code>--ASMLIST</code>
<code>/k pathlist</code>	n/a
<code>/l pathlist</code>	n/a
<code>/m filename</code>	<code>-M</code>
<code>/n length</code>	n/a
<code>/o filename</code>	<code>-O</code>
<code>/q</code>	<code>-Q</code>
<code>/u sym[=value]</code>	n/a
<code>/v</code>	<code>--WARN=-9</code>
<code>/w</code>	n/a
<code>/x</code>	n/a
<code>/z symbol=value</code>	<code>-L-Usymbol</code> (as an undefined symbol)

The generation of the assembly list file and output files, such as the COD and HEX files, are independently controlled using MPLAB XC8, and are attributes of different internal applications. Thus, there is no direct counterpart to the MPLAB C18 options such as `/i`, `/w` and `/x`. However you may control list file generation with the MPLAB XC8 option `--ASMIST`, and the output file type with `--OUTPUT`.

Chapter 3. Language Features

3.1 INTRODUCTION

This chapter compares source code differences between the MPLAB C18 and MPLAB XC8 compilers. It presents important issues to consider when migrating C source code to MPLAB XC8. If you are familiar with MPLAB C18, this information will highlight equivalent features that you can use in the MPLAB XC8 C compiler.

Information has been organized into the following sections.

- Operating Modes
- Memory Models
- Integer Promotions
- Device-Specific Information
- Data Types and Limits
- Size Limitations
- Storage Classes
- Storage Qualifiers
- Pointer Storage Qualifiers
- Function Variants
- Structures and Unions
- Interrupts
- Locating Objects
- Function Reentrancy and Calling Conventions.
- The Runtime Startup Code
- Register Usage
- Preprocessing
- C and Assembly
- Linking

3.2 OPERATING MODES

MPLAB C18 has two operating modes: extended and non-extended. These modes relate to the instruction set utilized on the PIC18 device. Only those PIC18 devices with the extended instruction set can use the extended compiler mode. All devices can use the non-extended mode, which uses the standard PIC18 instruction set. The choice of this mode affects how programs are compiled and what features are available in the source code.

There are no corresponding modes with MPLAB XC8. The compiler does *not* support the PIC18 extended instruction set; code is always compiled for the standard PIC18 instruction set.

3.2.1 Operating Mode Porting Guidelines

Use the following guidelines when porting code to MPLAB XC8.

- Code compiled for MPLAB XC8 will use the non-extended instruction set.
- Ensure configuration bit settings disable the extended instruction set.
- Ensure hand-written assembly code assumes the non-extended instruction set is in use.

3.3 MEMORY MODELS

The MPLAB XC8 compiler does not use memory models, and the MPLAB C18 options used to specify this can be ignored.

The memory models used by MPLAB C18 only affect the size of pointer variables. The size of each pointer variable allocated by MPLAB XC8 is determined independently and automatically, based on the addresses that are assigned to that pointer in the entire program.

3.4 INTEGER PROMOTIONS

By default, MPLAB C18 will perform arithmetic operations using the size of the largest operand, even if both operands are smaller than an `int`. This behavior does not conform to the ANSI Standard but can be changed using the `-Oi` driver option.

The MPLAB XC8 compiler cannot mimic this non-standard behavior. The results obtained from this compiler will always be consistent with the smaller data types (i.e. `short`, `char` and structure bit-fields) being promoted to either `signed int` or `unsigned int`. This means that the result of expressions using the smaller types can be different to those obtained from the same code compiled with MPLAB C18.

Integer promotion (also called integral promotion) is discussed (with examples) in the MPLAB XC8 user's guide, and in any good book on the C programming language.

<p>Note: Do not be tempted to believe that changing from a smaller to a larger type will not change the numerical result of an expression. Look especially for cases where a smaller unsigned type would be promoted to <code>signed int</code>. A change in operand signedness can lead to a very different expression results.</p>

3.4.1 Integer Promotion Porting Guidelines

Any MPLAB C18 code compiled *without* the `-Oi` option and which uses operands smaller than an `int` must be reviewed. Check to ensure that expressions involving these smaller types will yield the same result when each type is promoted to the 16-bit `int` types used with MPLAB XC8.

3.5 DEVICE-SPECIFIC INFORMATION

3.5.1 Header Files

MPLAB XC8 uses the header file `<xc.h>` for all device specific information and it is likely you will need to include this header in all your C source modules.

Separate assembly modules should include the header `<xc.inc>` to access SFRs by name.

3.5.2 Special Function Registers

The names defined by MPLAB XC8 for SFR registers and bits within those registers should be similar to the names used by MPLAB C18, so use of these identifiers should port with little effort. Any mismatch will be reported as an undefined symbol.¹

In MPLAB XC8 code, you can access an SFR register as a whole, for example using the symbol `PORTA`. You can access bits within that register using a structure with bit-field members representing each bit, for example `PORTAbits.RA0`.

TIPS: MPLAB XC8 also allows you to use predefined `bit` variables, such as `RA0`, `RA1`, etc., instead of the bit-fields; however, these are less portable.

For separate assembly modules, include the file `<xc.inc>` when compiling with MPLAB XC8. You can use the `#include` directive to do this, provided you enable the `-P` option to allow preprocessing of assembly source files. Alternatively, you can use the `INCLUDE` assembler directive. The names of the assembly symbols that represent the SFRs and the bits within them will be the same as their C counterparts.

3.5.3 Configuration Bits

Configuration bits are set using the `config` pragma in both MPLAB C18 and XC8 compilers. The syntax is identical, and in most instances code written for MPLAB C18 should work as expected when using MPLAB XC8 with no modification. The names of configuration settings and values should be identical, but name changes have occurred in the past. If you are using an old version of MPLAB C18, some symbol names might need to be brought up to date. Check the MPLAB XC8 user's guide for more information on this pragma.

TIPS: You can open the `pic_chipinfo.html` or the `pic18_chipinfo.html` files in the `docs` directory of the MPLAB XC8 compiler to see the settings-value names used by the pragma and examples for all supported devices. The same pragma can be used to set the ID location bits.

¹An accurate means of determining the MPLAB XC8 SFR symbol names for the device you are using is to open the preprocessed file (`.pre` extension) of any module in your project which includes the `<xc.h>` header.

This preprocessed file will contain the C definitions for most device-specific information.

3.5.4 Built-in Routines and Macros

MPLAB XC8 defines several preprocessor macros that can assist with device-specific code. Check the MPLAB XC8 user's guide for more information on the macros and in-built routines listed in Table 3-1.

TABLE 3-1: EQUIVALENT MPLAB XC8 MACROS AND IN-BUILT FUNCTIONS

MPLAB C18	MPLAB XC8 Equivalent
Nop()	NOP()
ClrWdt()	CLRWDT()
Sleep()	SLEEP()
Reset()	RESET()
Rlcf(), Rlncf(), Rrcf(), Rrncf()	Use regular rotate expression ¹
Swap()	Use regular swap expression 2

Note 1: The compiler looks for expressions such as `c = (c << 1) | (c >> 7);` and encodes these using a rotate instruction.

2: An expression such as `(c >> 4) | (c << 4)` can swap nibbles in a value.

TIPS: MPLAB XC8 also implements `_delay()`, `_delaywdt()`, `__delay_us()` etc., in-built routines associated with delay loops; interrupt enable macros, such as `ei()` and `di()`; and `__EEPROM_DATA()` for preloading data to EEPROM.

3.5.5 Device-specific Porting Guidelines

When compiling for MPLAB XC8, you should ensure the following are done.

- Include the file `<xc.h>` in all C modules that need access to device-specific information.
- Remove inclusion of MPLAB C18 headers, such as `<p18cxxx.h>`, `<p18c452.h>` or any other device-dependent header file in the code.
- Include the file `<xc.inc>` for separate assembly modules.
- Check for undefined symbol errors. Correct SFR names, settings and value names used with configuration macros, preprocessor macros and in-built function names, using the documentation and techniques described above.

3.6 DATA TYPES AND LIMITS

The integer data types defined by MPLAB C18 are identical in size to those defined by the MPLAB XC8 compiler.

The type of a plain `char` is `signed char` on MPLAB C18 (unless the `-k` option is used), but `unsigned char` on MPLAB XC8. It is always good practice to explicitly state the signedness of `char` types when you define them.

Both compilers allow the use of a non-standard `short long int` type, and this type is equal in size in both implementations.

Both the `float` and `double` types defined by MPLAB C18 are 32-bit wide and use the IEEE-754 format. The size of the `float` and `double` types when using MPLAB XC8 are configurable. Each type can be independently set to a 32-bit IEEE-754 format, as used by MPLAB C18, but a 24-bit truncated form of this format is used by default. Set the MPLAB XC8 options `--DOUBLE` and `--FLOAT` to 32 for exact compatibility, if required. See the MPLAB XC8 user's guide for more information on these options and floating-point types.

Values stored by MPLAB XC8 use a little endian format, identical to that used by MPLAB C18.

The order of allocation of bits in bit-fields are identical for both compilers: from least to most significant in order of allocation.

TIPS: The XC8 compiler allows use of a non-standard type, `bit`, which can hold a single-bit wide integer (as opposed to boolean) values.

3.6.1 Data Type Porting Guidelines

Ensure the following guidelines are enacted.

- Explicitly state whether plain `char` variables defined in MPLAB C18 code should be `signed` or `unsigned`.
- If floating-point objects used in MPLAB C18 must be 32-bits wide, use the `--FLOAT` and/or `--DOUBLE` options to increase the type size in MPLAB XC8.

3.7 SIZE LIMITATIONS

The maximum size of individual functions and data objects could vary between MPLAB C18 and MPLAB XC8. The limits imposed by MPLAB XC8 are given below.

3.7.1 Function Size Limits

The size of assembly code generated for a function is limited only by the available program memory on the target device.

Note: Note that for each C function, the size, location, and the actual instructions themselves will be different in the assembly generated by MPLAB XC8.
--

3.7.2 Data Size Limits

For `auto` and parameter objects associated with a function that is using the *compiled stack*, the following limitations apply.

- The size of an *individual* `auto` or parameter object (e.g. an array or structure) can not exceed the size of a PIC18 data bank, 100h bytes.
- The total size of *all* parameters for a function must not exceed the size of a PIC18 data bank. (This does not apply to `auto` objects.)
- The total size of *all* `auto` and parameter objects for *all* functions in the program is limited only by the available data memory.

For `auto` and parameter objects associated with a function that is using the *software stack*, the following limitations apply.

- The size of an *individual* `auto` or parameter object (e.g. an array or structure) can not exceed the size of a PIC18 data bank, 100h bytes.
- The total size of *all* `auto` and parameter objects for a function must not exceed the size of a PIC18 data bank.
- The total size of *all* `auto` and parameter objects for *all* functions in the program is limited only by the available data memory.

For all non-stack based objects (e.g., global and static objects), their sizes are limited only by the available data memory.

3.8 STORAGE CLASSES

The MPLAB C18 compiler allocates space for local variables based on their storage class specifier: `auto`, `static` and the non-standard class, `overlay`. All `auto` objects are allocated space on one of two data stacks; `static` objects are allocated permanent static storage; and `overlay` objects are positioned in the compiled stack, if the compiler is operating in non-extended mode. The `overlay` specifier has no effect when the compiler is operating in extended mode.

Allocation to a compiled stack is a static allocation, but one that can share memory with other local objects that are never concurrently active.

The above specifiers directly relate to whether a function will be reentrant. Refer to **Section 3.15 “Function Reentrancy and Calling Conventions.”** for more information and porting guidelines relating to these class specifiers.

3.9 STORAGE QUALIFIERS

The MPLAB C18 compiler defines several non-standard qualifiers, as well as the standard `const` and `volatile` qualifiers. Qualifiers are used with basic objects, as well as with pointer target types, which are discussed separately in

Section 3.10.1 “Pointer Storage Qualifier Porting Guidelines”.

A programmer using MPLAB C18 has independent control over the placement of objects to data memory or program memory using the `rom` and `ram` qualifiers. The standard read-only indicator, `const`, can be used in conjunction with these memory specifiers.

When using the MPLAB XC8 compiler, the `const` qualifier has a dual action of making global objects read-only, plus allocating them to program memory. Thus, the MPLAB XC8 `const` qualifier performs the same action as `const` and `rom` used together in MPLAB C18. It is not possible to define a global, read-only variable that is located in data memory with the MPLAB XC8 compiler, but `const auto` objects are allocated to RAM.

The MPLAB C18 `far` qualifier indicates that an object is not `near`, i.e., it located in banked memory. Objects, by default, are `far`. The MPLAB XC8 compiler also, by default, places objects that are not qualified anywhere in the banked memory, but there is no qualifier that explicitly states this.

The MPLAB XC8 `far` qualifier has a different meaning to its namesake in MPLAB C18. An object that is qualified `far` will be located in program memory by the MPLAB XC8 compiler, but will be placed in extended memory that the compiler assumes is writable.

The `near` and `far` MPLAB XC8 qualifiers are both controlled by the `--ADDRQUAL` option. See the MPLAB XC8 user's guide for more information on this option and the qualifiers.

3.9.1 Storage Qualifier Porting Guidelines

Use the following guidelines, in the order they are listed, to convert qualifiers to the nearest MPLAB XC8 equivalent.

- Replace all instances of `const` and `rom` qualifiers, used together, with `const`.
- Remove all instances of the `ram` qualifier.
- Remove all instances of the `far` qualifier.

Note that there are no direct equivalents in MPLAB XC8 for the following MPLAB C18 qualifier sequences. Consider the suggested alternatives.

- Only the `const` qualifier used with global objects: Consider removing the `const` qualifier, resulting in the object being writable, or retaining the qualifier and having it read-only but in program memory. (No change is required for `const auto` objects.)
- The `rom` qualifier (used without `const`): Consider replacing this with `far` if you want the object to be writable and in external memory, or replace with `const` if it is never written.
- Both `near` and `rom`, used together: Consider replacing these with `const`, which will place the object in program memory, but at any address. If the target device has more than 64k of program memory, consider making the object absolute if it is being linked above the 64k boundary and this is an issue; although, most `const` objects are allocated below this address.

Other qualifier sequences do not need modification.

3.10 POINTER STORAGE QUALIFIERS

When defining pointers and using the MPLAB C18 compiler, the target type must use the `near`, `far` and `rom` qualifiers, where appropriate. These change the size of the pointer and access method used to dereference the target. So, for example, if a pointer is to hold the address of an object in program memory that is located below the 64k boundary, it should be defined as follows.

```
rom near char * npsp;
```

A pointer to a `rom`-qualified target cannot point to a target that is located in RAM, and a (default) pointer to RAM cannot point to an object in program memory. The qualifiers used with the pointer definition must match those used with the objects whose addresses are assigned to the pointer.

Qualifiers are *not* required with pointer target types when using MPLAB XC8, and they are actually ignored. The compiler tracks all address assignments to pointers, so a pointer's targets are always known. As the memory space and assigned bank of all objects is known by the compiler, the best size and pointer deference method can be chosen automatically.

Note: This means that if you never assign an address to a pointer, or only ever assign a `NULL` pointer, the MPLAB XC8 compiler will be aware of this fact, as well. In such a case, the compiler could radically optimize the size of the pointer and how it is dereferenced.

A generic MPLAB XC8 pointer can access any object (of the correct type) no matter which memory space it resides in. The same pointer can be used to access both data and program memory targets.

Use the `const` and `volatile` qualifier when defining pointers to indicate their usual meaning (read-only and do-not-optimize-access, respectively), but all other non-standard qualifiers should not be used and have no effect. So, for example, the above pointer example would be defined as follows when using MPLAB XC8:

```
char * npsp;
```

MPLAB C18 pointers will be either 16- or 24-bits wide, based on the qualifiers used when the pointer is defined. Pointers defined when using MPLAB XC8 will be either 8-, 16-, or 24-bits-wide, based on the addresses assigned to them in the code. Their size is fixed for the duration of the program, but might change from one build to the next as source code is developed. Your source code should not be making assumptions as to the size of a pointer.

3.10.1 Pointer Storage Qualifier Porting Guidelines

Porting pointer variables is relatively straight forward. Ensure you understand the difference between a pointer's *target type* qualifiers and the *pointer* qualifiers in a definition.

- Remove all instances of `near`, `far` or `rom` from all pointer target types.
- Review any pointer qualifiers, as described in **Section 3.9.1 “Storage Qualifier Porting Guidelines”**.

3.11 FUNCTION VARIANTS

A consequence of the qualifiers required with MPLAB C18 pointer definitions (described in **Section 3.10 “Pointer Storage Qualifiers”**) is that a function can need several variants to work with data in different memory spaces. For example, the `strcpy()` function, which accepts two pointer parameters, has four variants to handle the possible source and destination combinations. Their prototypes are shown below.

```
char      *strcpy (auto char *s1, auto const char *s2);
char      *strcpypgm2ram (auto char *s1, auto const rom char *s2);
rom char  *strcpyram2pgm (auto rom char *s1, auto const char *s2);
rom char  *strcpypgm2pgm (auto rom char *s1, auto const rom char *s2);
```

Since the MPLAB XC8 compiler's tracking of pointer assignment also applies to library functions, only one variant of this function is needed and has the Standard ANSI prototype:

```
char *strcpy (char *s1, const char *s2);
```

The size of the pointer parameters and the encoding of this function will be based on the actual addresses passed to the function in your program.

When migrating code to MPLAB XC8, you only need ever call the base version of the function variants. So for example, calls to `strcpypgm2ram()` should be changed to calls to `strcpy()`.

If you are writing a function that accepts a pointer argument, you only need write one version of that function. You can pass to this function the address of any object (of the correct type) regardless of which memory space it resides in.

3.11.1 Function Variants Porting Guidelines

Porting code which uses multiple variants of a routine amounts to simplifying the code.

Replace calls made to variants of a routine to calls to the base routine itself. So, for example, calls to `strcpypgm2ram()` should be changed to calls to `strcpy()`.

3.12 STRUCTURES AND UNIONS

Unnamed (anonymous) structures and unions are supported by both MPLAB C18 and XC8. Use of this feature in MPLAB C18 will freely migrate to MPLAB XC8 without need of alteration. Such features are not compliant with the ANSI Standard so you should try to avoid these, if possible.

The MPLAB C18 compiler allows signed bit-fields. These are currently not supported by MPLAB XC8.

3.12.1 Structure Porting Guidelines

If your MPLAB C18 code defines signed bit-fields, the following change could be necessary.

Change instances of `signed bit-fields` to `unsigned`, or change them to `signed char` structure members.

3.13 INTERRUPTS

The definitions of interrupt vectors and service routines when using MPLAB XC8 is different to those applicable for MPLAB C18. Encoding of the service routines also differs between the two compilers.

The MPLAB C18 compiler expects two functions to be written for each interrupt priority: one to define the interrupt service routine (ISR), and the other to be linked at the interrupt vector and which will transfer control to the ISR. This is shown in the following MPLAB C18 example.

```
#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
    _asm GOTO low_isr _endasm
}

#pragma code /* return to the default code section */
#pragma interruptlow low_isr
void low_isr (void)
{
    /* service routine body goes here */
}
```

The MPLAB XC8 compiler expects only one function to be defined for each interrupt. The compiler will automatically complete the code associated with the interrupt vector once it has seen the interrupt service routine. The ISR is created when using the `interrupt` specifier with a function. The above example would be written as the following when using MPLAB XC8.

```
void interrupt low_priority low_isr (void)
{
    /* service routine body goes here */
}
```

For the high priority (default) interrupt, omit the `low_priority` keyword when defining the function; or you can use the `high_priority` keyword instead, for example

```
void interrupt high_isr (void)
{
    /* service routine body goes here */
}
```

There are virtually no restrictions as to the code you can include in the body of an interrupt function when using MPLAB XC8, but small, simple routines are better when real-time performance is required.

The MPLAB XC8 compiler can deduce all registers used by the interrupt function and any functions (including library functions) it calls; however, in-line assembly code is *not* scanned for register usage. If any assembly routines are called from an interrupt function, the `pragma regsused` can be used to specify registers used by that assembly routine. This might force extra registers to be saved in the context switch. There is no mechanism by which you can restrict the set of registers normally saved by the interrupt routine.

3.13.1 Interrupt porting Guidelines

The following guidelines should be followed for each interrupt priority.

- Remove the function which is linked to the interrupt vector. Such functions are surrounded by `#pragma code` directives. Remove these directives as well.
- Remove the `interrupt` or `interruptlow` pragmas associated with the ISR.
- Add the keyword `interrupt` to the ISR prototype.
- Add the keyword `low_priority` to the ISR prototype (along with the `interrupt` specifier) if this function handles the low priority interrupt.
- Confirm if additional registers used by in-line assembly or separate assembly routines need saving, and use the `#pragma regsused` directive for the routine that contains the assembly code. Alternatively, you can manually save the registers in the assembly code.

3.14 LOCATING OBJECTS

The MPLAB C18 compiler uses pragmas to locate objects in specific memory locations. Although there is an MPLAB XC8 equivalent of this pragma, it is usually best to consider the motivation for placing the objects at non-default locations and choosing one of the alternative strategies available with MPLAB XC8. These are discussed in the following sections.

3.14.1 Variables and Functions

The MPLAB C18 `#pragma sectiontype` is used to allocate variables and code to an alternate section with the name specified. This is usually done so that this section can be explicitly linked in memory at a specific address or in an address range.

If the aim is purely to locate a variable or function at a specific address, then the easiest way of performing this in MPLAB XC8 is to make the variable or function absolute, using the `@ address` construct.

For a variable defined in MPLAB C18 as follows:

```
#pragma udata myUdata=0x100
int foobar;
```

you can allocate it the same address in MPLAB XC8 using the following definition for an absolute variable:

```
int foobar @ 0x100;
```

For a function defined in MPLAB C18 using the similar pragma:

```
#pragma code myCode=0x2000
int calcOffset(int radius) { ...
```

The same effect can be applied using the following definition of an absolute function in MPLAB XC8.

```
int calcOffset(int radius) @ 0x2000 { ...
```

If the original intention behind using the `sectiontype` pragma was only to ensure that a variable is allocated to a specific bank, then the MPLAB XC8 `bankx` keywords are the easiest way to do this; however, these keywords only allow placement in the first four banks of any device. For example

```
bank1 int foobar;
```

will allocate `foobar` to bank 1 data memory. For the qualifier to work as expected, you must set the `--ADDRQUAL` option to `request`. The default compiler operation is to ignore bank qualifiers.

If the intention is to specifically allocate the variable or function to a new section, the MPLAB XC8 `__section()` specifier can be used to indicate an alternate section for an object. The above MPLAB C18 example which places `foobar` in a new section could be replaced with MPLAB XC8 code similar to:

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

It is not possible to specify an address for the section in the code. You must use an option to locate the section at a specific address, or in an address range. You might, for example, use the MPLAB XC8 driver option:

```
-L-pmyUdata=0100h
```

to place the `myUdata` section at address 0x100. The leading `-L` is stripped from this option and the remainder (the `-p linker` option) is passed directly to the linker. You do not need to run the linker explicitly to adjust the linker options.

You can also link a section anywhere in a pre-existing or user-defined address range (known as a linker class). For example, the following options define a new linker class (using the `-A linker` option), then link the section anywhere in this class.

```
-L-AMYSpace=50h-0ffh,100h-1ffh  
-L-pmyUdata=MYSpace
```

See the MPLAB XC8 user's guide for full information on sections and linker options.

Note: The MPLAB XC8 user's guide refers to sections as "psect". This term is short for "program section", but is an identical concept to sections used by MPLAB C18 and other compilers.

3.14.1.1 LOCATING VARIABLES AND FUNCTIONS PORTING GUIDELINES

- Replace all variables defined using the `sectiontype` pragma with either absolute variables, variables using the bank qualifiers or place them in a user-defined psect using the `__section()` specifier.
- Replace all functions defined using the `sectiontype` pragma with either absolute functions or place them in a user-defined psect using the `__section()` specifier.
- If required, add options to link any user-defined sections at the required address or in an address range.
- Confirm the placement of these objects using the MPLAB XC8 map file.

3.14.2 Indicating Object Locations

When using MPLAB C18 it is necessary to manually indicate the location of variables defined in other modules to ensure optimal code generation. This is done using the `varlocate` pragma.

Such information is not necessary when using MPLAB XC8. The compiler is aware of the bank in which all C objects are placed, even if they use the `__section()` specifier or the object is absolute.

3.14.2.1 OBJECT LOCATION PORTING GUIDELINES

Remove all `varlocate` pragmas from your source.

3.14.3 Temporary Data Location

MPLAB C18 uses the `tmpdata` pragma to specify the location of temporary data. The location of temporary data cannot be explicitly changed when using MPLAB XC8. The temporary variables used by a function are grouped with that function's `auto` variables and are stored in either the compiled stack or software stack, based on whether the function is reentrantly encoded.

Provided there is no code that makes assumptions about the location of temporary data, removing these pragmas should not affect code operation.

3.14.3.1 TEMPORARY DATA LOCATION PORTING GUIDELINES

- Remove all instances of the `#pragma tmpdata` directive.
- Ensure that no code makes assumptions about the location of temporary data.

3.15 FUNCTION REENTRANCY AND CALLING CONVENTIONS.

Note: The MPLAB XC8 reentrancy features that are described in this document will be introduced in future versions of this compiler.

Ensure that the compiler version you are using supports reentrancy, if your project requires this feature. Only the compiled stack is utilized by MPLAB XC8 compilers that do not support reentrancy, and there are no options or specifiers to control this behavior.

Both MPLAB C18 and XC8 allow functions to be called reentrantly; however, the program stack models used by both compilers are different

Reentrancy issues ultimately relate to the compiler's placement of `auto` and parameter variables. For a function to be reentrant, its `auto` and parameter objects need to be allocated memory that is unique for each instance of the function. If a function is not called reentrantly in a program, then there is no requirement for these variables to be allocated unique memory locations and they can each be statically assigned an address.

What Microchip calls a *software stack* can provide storage for `auto` and parameter variables associated with reentrant functions. Alternatively, static allocation of these objects is made to what is known as a *compiled stack*. Accessing a compiled stack is typically more efficient than accessing a software stack, but this storage does not allow reentrancy.¹

Both MPLAB C18 and XC8 employ both a software stack and compiled stack to store `auto` and parameter variables, but the default behaviors of the compilers are different. So too are the controls you have over the placement of variables in the stacks.

3.15.1 Default Function Reentrancy Behavior

By default, MPLAB C18 encodes all functions in a reentrant manner, i.e., they use the software stack. If the target device supports the extended instruction set, then extended mode is enabled. Accessing the software stack in this mode is more efficient than in non-extended mode.

By default, MPLAB XC8 encodes in a reentrant manner only those functions that have been called reentrantly in the program. The `auto` and parameter variables for these functions will, thus, use the software stack. Functions which have not been called reentrantly are encoded using the compiled stack and will be non-reentrant. This means that most functions will be efficiently encoded to use the compiled stack, but reentrancy is supported when required. This is called a hybrid stack model, since both stacks can be used by the one program. Only the standard instruction set is used at all times.

The MPLAB XC8 compiler has the ability to detect which functions are called reentrantly from the complete call graph that is built during compilation, and this information is used by the compiler when using the hybrid model. A function is considered to have been called reentrantly if it is part of a loop in the call graph, or it appears in multiple call graphs, e.g. in the main-line code and an interrupt call graph. Functions called indirectly via a pointer are considered by the compiler.

You can find out how a function has been encoded, i.e., which stack has been used, but looking at the function information shown in the assembly list file or in the map file. The MPLAB XC8 user's guide has complete information on the layout of these files.

¹The MPLAB XC8 compiler can duplicate functions that must use the compiled stack and that are called from more than one call graph. This makes these functions *appear* to be reentrant. This technique cannot be used for recursively-called functions. If you force a function to use the compiled stack (discussed later) and it is called from both main-line and interrupt code, then duplication will take place.

If your project uses MPLAB C18's default behavior regarding function reentrancy, no action is typically needed when porting to MPLAB XC8.

If you have used any of the language keywords or compiler options to change the default MPLAB C18 compiler behavior, these features and porting them are described in the following sections.

3.15.2 Function Reentrancy Controls

Since MPLAB C18 allocates all `auto` and parameter variables on the software stack by default, the only options available with this compiler are those that force these variables to be allocated to the compiled stack.

The `overlay` specifier will force `auto` variables to be allocated to the compiled stack. The `static` specifier performs the same action for parameter variables. (Specifying an otherwise `auto` variable as `static` has the usual effect stipulated by the ANSI standard: it will have permanent program duration and its memory allocated in a similar fashion to global variables.)

The MPLAB C18 option, `-sco`, can be used to force *all* `auto` and parameter variables in the entire program to be overlaid, hence allocated to the compiled stack; or the option `-scs` will change *all* `auto` and parameter variables to be `static`.

By contrast, MPLAB XC8 only allows control of how each *function* allocates all of its `auto` and parameter variables. There are no controls to change the allocation of a single variable. However, other options can be used to change the program wide behavior.

There could be instances where you want to force a function to use the software stack for its `auto` and parameter variables, even though it is not reentrantly called in the C program, for example, if it was called from assembly code in such a way as to require this. For these functions, use the MPLAB XC8 `reentrant` specifier.

If you require that a function must use the compiled stack, you can use the `nonreentrant` specifier to indicate this. If such a function is then called from multiple call graphs in the program (and so needs to be reentrant), the function output will be duplicated. (See MPLAB XC8 user's guide for more information on function duplication.) An error will result if a function specified `nonreentrant` is called recursively in your source code.

The MPLAB XC8 option `--STACK=compiled` forces *all* functions to be encoded using the compiled stack, regardless of how they are called in the program; the option `--STACK=reentrant` forces all functions to use the software stack. The `--STACK=hybrid` option explicitly states the default behavior, which is to allocate a stack for each function based on how it is called in the program. If the `--STACK=compiled` option is used and a function (which does not use the `reentrant` specifier, see below) is reentrantly called from multiple call graphs in the program, the function output will be duplicated. An error will result if a function is called recursively in the program and this option is used.

The MPLAB XC8 keywords described above have precedence over the `--STACK` option settings where there is conflict. Thus, you might specify all functions to be non-reentrant, using the `--STACK=compiled` option, but allow one function reentrancy by using the `reentrant` specifier where you define that function.

3.15.3 Reentrancy Porting Guidelines

When porting code from MPLAB C18, it is recommended that you use the hybrid (default) model in MPLAB XC8 and only use the specifiers and options if required. Since reentrantly called functions are automatically detected, you do not need to indicate this information to the compiler. Porting should mainly consist of removing the explicit controls used with MPLAB C18.

Use the following guidelines when porting from MPLAB C18 to XC8.

- Use the MPLAB XC8's default hybrid-stack model.
- Remove all instances of the specifier `overlay` used with `auto` variables.
- Remove all instances of the `static` specifier used with parameters.
- Use the `static` specifier with local variables in accordance with the usual ANSI Standard meaning (if you require these variables to have a permanent duration).
- If you were using the C18 option `-scs`; instead, explicitly state the `static` storage class when defining each local object (remembering that parameters can not be specified `static`).
- If you were using the C18 option `-sco`, consider using the MPLAB XC8 option `--STACK=compiled`.
- Review if any function must always use the software or compiled stack and use the `reentrant` and/or `nonreentrant` specifiers accordingly.

If you find you are using the `nonreentrant` specifier with many, or all, functions, consider swapping to the compiled stack (`nonreentrant`) model using the `--STACK=compiled` option. Similarly, if you are using the `reentrant` specifier repeatedly, consider using the `reentrant` model using `--STACK=reentrant`.

3.15.4 Calling Conventions

The function calling convention used by each compiler is different and also varies based on the stack model being used.

Neither compiler has direct controls over the calling convention. The change in behavior that is seen when moving to MPLAB XC8 is unlikely to be of consequence, unless you have hand-written assembly code that is accessing stack-based objects. Such code will need review. Although this is beyond the scope of this document, the following is general information that will assist with this process.

In MPLAB XC8, when a function uses the compiled stack, `auto` and parameter variables have static addresses that can be accessed using global symbols of the form `functionName@variableName`. So, for example, an `auto` variable called `input` defined in `main()` can be accessed using the symbol `main@input`. Use the assembler `GLOBAL` directive to link with this symbol's definition.

It is recommended that you do not access objects stored on the software stack. When a function is using the software stack, the compiler will dedicate FSR1 as the stack pointer register. No frame pointer is used, and hence, no symbol can be defined to represent an object's offset in the stack. The contents of the stack pointer can vary during the execution of a function, so no fixed offset can reliably be used to reference stack-based objects.

When a function using the software stack is called, the calling function places (pushes) any parameters on the stack in an order that is the opposite order to that in which they are defined in the source code. So if a function that has the following prototype

```
int aReentrantFunction(int a, int b);
```

is called, the argument for `b` is pushed onto the stack, then the argument for `a`. After the function call is made, the called function will reserve space on the stack for any `auto` or temporary variables that it defines.

The stack grows upward in memory, toward higher addresses. That is, a push increments the stack pointer, a pop decrements it.

Before the called function returns, the stack pointer is adjusted (lower) so that any `auto` or local variables are freed.

3.16 THE RUNTIME STARTUP CODE

With MPLAB C18, there are several precompiled runtime startup modules available. These are linked in to your project, based on your selected options. Initialized `static` objects are not cleared unless a special runtime startup module is linked in. To conform to the ANSI Standard, these objects must be cleared.

With MPLAB XC8, there are no precompiled runtime startup modules. Assembly code that performs this task is generated by the compiler after examination of the entire C program. The compiler generates only the code required, resulting in an optimal startup sequence. This code is automatically included into your project.

There is an option to control how this code is generated; however, it is unlikely its use will be warranted. Using this option, you can select to omit code that clears or initializes variables etc., but note that this can cause code failure. The option that controls this is `--RUNTIME`. See the MPLAB XC8 user's guide for the many suboptions available with this option. All `static` variables are initialized, unless the relevant `--RUNTIME` option is used.

If you wish to add custom code that is executed immediately after reset, then use the MPLAB XC8 compiler's powerup routine feature. The assembly code in this routine is executed before the runtime startup code. You do not need to precompile this routine; simply include the source file into your project, and rebuild. This is the equivalent to the MPLAB C18 `entry()` function in the runtime startup routines.

3.16.1 Runtime Startup Module porting Guidelines

- Remove all startup modules from your project (MPLAB C18 object files are not compatible with MPLAB XC8).
- Confirm that initializing `static` objects will not adversely affect your project's behavior.
- Move any code from the `entry()` function to the powerup routine.

3.17 REGISTER USAGE

The registers used by the MPLAB XC8 code generator are almost identical to those used by MPLAB C18. These registers, along with their primary usage, are indicated in Table 3-2. (Note that the registers can have other uses, for example `PROD`, `FSR0`, `FSR2`, and `TBLPTR` registers can be used by MPLAB XC8 to temporarily hold values by reentrant functions.)

The MPLAB XC8 compiler assumes these registers are not modified by in-line assembly code. In-line assembly code may need to save and restore them if they must be modified.

TABLE 3-2: REGISTERS USED BY THE COMPILER

Compiler-Managed Resource	Primary Use(s) in C18	Primary Use(s) in XC8
PC	Execution control	Execution control
WREG	Intermediate calculations	Intermediate calculations
STATUS	Calculation results	Calculation results
BSR	Bank selection	Bank selection
PROD	Multiplication results, return values, intermediate calculations	Multiplication results
<code>section.tmpdata</code>	Intermediate calculations	n/a
FSR0	Pointers to RAM	Pointers to RAM
FSR1	Stack pointer	Stack pointer
FSR2	Frame pointer	Pointers to RAM
TBLPTR	Accessing values in program memory	Accessing values in program memory
TABLAT	Accessing values in program memory	Accessing values in program memory
PCLATH	Function pointer invocation	Function pointer invocation
PCLATU	Function pointer invocation	Function pointer invocation
<code>section MATH_DATA</code>	Arguments, return values and temporary locations for math library functions	n/a

MPLAB XC8 ensures that these registers will be saved and restored by an ISR if they are used by C code in that function, or any of the functions called by the ISR. Note that in-line assembly code is not scanned for register usage.

3.17.1 Register Usage Porting Guidelines

- Avoid using any compiler-managed register in your project.
- Remove any references to sections not defined by MPLAB XC8, e.g `section.tmpdata`.
- Ensure any compiler-managed register clobbered by in-line assembly code is saved and restored by the assembly code.

3.18 PREPROCESSING

There are slight differences in preprocessing and predefined macros between the MPLAB C18 and XC8 compilers.

3.18.1 Predefined Macro Names

Preprocessor macros can be defined using `#define` in your source code. Both compilers allow you to define these macros on the command line, and both use the `-D` option to do this. However, several macros are predefined by the compiler itself. You will not see definitions for these, but you can use them in your source code.

The predefined macros defined by MPLAB C18 are shown in Table 3-3 along with the MPLAB XC8 equivalent. These macros are only defined if the corresponding condition is true.

TABLE 3-3: PREDEFINED PREPROCESSOR MACROS

MPLAB C18 Macro	Condition When Set	MPLAB XC8 Equivalent(s)
<code>__18CXX</code>	The MPLAB C18 compiler is in use.	<code>__XC</code> , <code>__XC8</code>
<code>__PROCESSOR</code>	Code is being compiled for the indicated device, e.g., <code>__18F452</code>	<code>__PROCESSOR</code>
<code>__SMALL__</code>	The <code>-ms</code> command-line option has been used.	n/a
<code>__LARGE__</code>	The <code>-ml</code> command-line option has been used.	n/a
<code>__TRADITIONAL18__</code>	The Non-Extended mode is being used.	<code>__TRADITIONAL18</code>
<code>__EXTENDED18__</code>	The Extended mode is being used	n/a

Use the `__18CXX` macro and either the `__XC` or `__XC8` macros to allow code to compile under both compilers during development. For example:

```
#ifdef __18CXX
    signed int input; // to ensure no integral promotion issues
#elif defined (__XC8)
    signed char input;
#elif
#error "What exactly are we using to compile this code?"
#endif
```

TIPS: Note that there are many more predefined preprocessor macros created by MPLAB XC8. These are shown in the *MPLAB XC8 C Compiler User's Guide* (DS500002053).

3.18.1.1 PREDEFINED MACRO PORTING GUIDELINES

To migrate code, follow these steps.

- Remove instances of the `__SMALL__` and `__LARGE__` macros, as these will always evaluate as false. Review code that is conditional on these being defined.
- Code conditionally compiled using `__TRADITIONAL18__` and `__EXTENDED18__` macros will work as expected, but the latter will never be defined.
- Use the `__18CXX` and `__XC` or `__XC8` macros to indicate which compiler is in use, and to assist with porting.

3.18.2 Preprocessor Macros

Unlike MPLAB C18, the MPLAB XC8 compiler does not currently support preprocessor macros with variable argument lists. These will need to be rewritten as functions or split into several macros to handle the different argument combinations.

Both compilers perform macro expansion of pragma arguments. MPLAB XC8 allows for quoted arguments to the `config` pragma to prevent unintended substitutions being made.

3.18.2.1 PREPROCESSOR MACRO PORTING GUIDELINES

Replace all definitions of preprocessor macros with variable argument lists with equivalent functions, or create several macros with different argument combinations.

3.19 C AND ASSEMBLY

Like MPLAB C18, the MPLAB XC8 compiler allows for hand-written assembly code to be written in separate modules or in-line with C code. A complete migration process for the assembly code instructions is beyond the scope of this C migration guide. However, the following sections summarize the differences and the main changes that are most likely required.

3.19.1 Assembler Applications

MPLAB C18 actually uses two assemblers: one (an internal assembler) to process assembly specified in-line with C code; another (MPASM) to process separate modules containing hand-written assembly code. Although the assembly language accepted by MPASM is fully featured, the internal assembler is not. It does not accept assembly directives and can only be used to specify actual instructions and code.

By contrast, assembly code, whether it be in-line with C code, or in a separate module, is always processed by the same MPLAB XC8 assembler application (`aspc18`). You can use MPLAB XC8 assembler directives in both situations, but note the remarks concerning directives in the following section.

3.19.2 Assembly Language Differences

MPLAB XC8 does *not* support the PIC18 extended instruction set. Any MPLAB C18 assembly code which has been written for this instruction set will fail on MPLAB XC8. The PIC18 extended instruction set defines new instructions not present in the standard instruction set. But — most importantly — the behavior of the *standard* instructions, when using the extended instruction set, change. Thus, even code that only uses the standard instructions (but executes with the extended instruction set selected) will mostly likely fail. The amount of work to port to the standard instruction set can be prohibitive if there are large amounts of assembly in your project.

Changes can be required to the assembly instructions themselves when porting to MPLAB XC8. The differences in instruction syntax are listed in the Macro Assembler chapter of the MPLAB XC8 user's guide, but the most common changes are indicated in the following paragraphs.

The most common assembly code change when porting to MPLAB XC8 is the operand used for the instruction destination, which is `,w` or `,f` when using MPLAB XC8, not the `,0` and `,1` specified by MPLAB C18. Note also that `,f` is also used with the `RETFIE` instruction to indicate a fast return.

The MPLAB C18 pseudo-instruction `MOVFw operand` has not been implemented in the MPLAB XC8 assembler. Either replace instances of this with the instruction `MOVF operand,w`, or implement an MPLAB XC8 assembler macro (see `MACRO` and `ENDM` in the assembler section of the MPLAB XC8 user's guide) to define the instruction. You can also define a preprocessor macro to do the same job.

Global symbols defined in C source use a leading underscore character in the assembly domain used by MPLAB XC8, regardless of whether the assembly is in-line with C code or in a separate module. So, for example, the C variable `input` can be accessed using the symbol `_input` from assembly code, after first using the `GLOBAL` directive. Non-global symbols use special assembler representations, described in **Section 3.15.4 “Calling Conventions”**.

TIPS: MPLAB XC8 supplies several pseudo instructions, such as `LJMP` and `FCALL`, which are not implemented in MPLAB C18. These take care of page selection, but you can still use regular instructions to perform page selection and the calls and gotos in your code if you prefer.

The assembler directives used by both assemblers differ. A full list of directives and controls, and their functions, are given in the Macro Assembler chapter of the MPLAB XC8 user's guide. The most common MPLAB XC8 directives needed are the `GLOBAL` directive, which allows a symbol to be linked global with other symbols having the same name, and the `PSECT` directive (akin to the MPLAB C18 `CODE`, `IDATA` and `UDATA` directives, for example) which defines a section (or psect as they are referred to in MPLAB XC8 nomenclature).

Since assembly code must conform to the operation of the assembly code generated by the C compiler, some restructuring of the assembly will almost certainly be required. The assembly symbols assigned to C objects and their location in memory can be different. All assembly code for MPLAB XC8 must be contained within a psect (see the `PSECT` directive in the MPLAB XC8 user's guide) and the flags used with these psects must be appropriate for the information they hold. The most common flags that are necessary for correct operation are `reloc`, `delta`, and `space`, all of which are described in the assembly section of the *MPLAB XC8 C Compiler User's Guide* (DS500002053).

Writing code in assembly should be a last resort, as it is far less portable than C code. Ensure your task is not possible in C code before including assembly code in your project. If assembly code is unavoidable, the Macro Assembler chapter of the MPLAB XC8 user's guide has information on writing in assembly. Also consider looking at the assembly output of the compiler using the assembly list file to see the assembly code that the compiler generates. An assembly list file is automatically produced if you are using MPLAB X IDE. See the `--ASMLIST` option in the MPLAB XC8 user's guide if you are building outside of the IDE.

3.19.3 In-Line Assembly

The commands to specify in-line assembly differ. MPLAB C18 uses the tokens `_asm` and `_endasm` to indicate the start and end, respectively, of the assembly code block. These can be changed to `#asm` and `#endasm`, respectively, in MPLAB XC8. Alternatively, use the preferred syntax of `asm("instruction");` for each assembly instruction. For example:

```
_asm
    movlw 20
    movwf 33h
_endasm
```

could be changed to:

```
asm("movlw 20");
asm("movwf 33h");
```

or you can use:

```
#asm
    movlw 20
    movwf 33h
#endasm
```

The in-lined assembly code itself will almost certainly require modification. See the information in **Section 3.19.2 “Assembly Language Differences”** for basic information on the differences and note the following points which are specific to in-line assembly code.

The assembly-domain symbol associated with a variable defined in C code will always consist of the C identifier with a leading underscore character. This is not the case with MPLAB C18 when writing in-line assembly code.

MPLAB® C18 to XC8 C Compiler Migration Guide

The MPLAB XC8 compiler never truncates the value of a symbol unless there is an operator to perform this action. Most of the PIC instructions that take a file address operand require an offset into the currently selected bank. This means you must mask out the bank information from the address. Failure to do so can result in fixup errors from the linker. This masking is not required when using MPLAB C18.

The following example shows a C variable, `myCVar`, being correctly accessed in MPLAB XC8 assembly code using the `BANKSEL` pseudo instruction and `BANKMASK` macro. This code makes no assumptions regarding where (specifically, which bank) the variable is located.

```
GLOBAL _myCVar
movlw 66
BANKSEL (_myCVar)
movwf BANKMASK(_myCVar)
```

You can choose to write code to select banks and mask addresses using conventional instructions and operators, but note that those used above are more portable when moving to devices with different memory architectures.

3.20 LINKING

The MPLAB C18 compiler (specifically, the MPLINK application) uses linker scripts to indicate how sections should be placed in memory. MPLAB C18 linker scripts are not compatible with the MPLAB XC8 compiler. Indeed, MPLAB XC8 does not use any type of file to specify linker settings.

A set of linker options is generated by the MPLAB XC8 driver with each build. These options are based on the device you select, as well as the other driver options you use in your project.

These generated linker options control how memory is defined and where psects (sections) are allocated in that memory. These linker options work in conjunction with flags specified with the psect definition in the assembly code. The flags indicate special linking requirements of that psect.

The default linker options generated by the MPLAB XC8 compiler are suitable for the vast majority of projects. If the application has special requirements, additional linker options can be added, or the default linker options can be modified, using the `-L-` driver option. This driver option allows direct control over the linker options without you having to run the linker directly. (Do not confuse this option with the `-L` driver option, or the `-L` linker option).

If you have not modified the default linker script in your MPLAB C18 project, then the default linker options in MPLAB XC8 should work for the migrated project. Note that these are different compilers, producing different output, and that the location of objects and code will change.

If you have modified your MPLAB C18 project so as to specifically locate objects or sections, then you will need to review your code and MPLAB XC8 project settings. See **Section 3.14 “Locating Objects”** for information on making objects absolute, or to place them in user-defined sections with MPLAB XC8.

Simple modifications made to the MPLAB C18 linker scripts can be transcribed to an equivalent MPLAB XC8 linker option. As described above, use the `-L-` driver option to pass these to the linker, as shown using the following general forms.

The MPLAB C18 `SECTION` directive (found in a linker script), which has a general form:

```
SECTION NAME=sectionName ROM=memoryName
```

would be implemented using the MPLAB XC8 driver option:

```
-L-psectionName=linkerClass
```

assuming that `sectionName` has been defined in C code via the `__section()` directive, or in assembly code using the `PSECT` directive. If `linkerClass` is not an existing, compiler-generated linker class, it can be defined with the following driver option:

```
-L-AlinkerClass=start-end[,start-end]
```

If you wish the section to be placed at a specific address (rather than anywhere in an address range), you can alternatively use:

```
-L-psectionName=address
```

See the Linker chapter in the MPLAB XC8 user's guide for more information on the linker options which you can control using the `-L-` driver option. This chapter also has more information on the linking process and on psects in general.

MPLAB® C18 to XC8 C Compiler Migration Guide

NOTES:

Glossary

A

Absolute Section

A GCC compiler section with a fixed (absolute) address that cannot be changed by the linker.

Absolute Variable/Function

A variable or function placed at an absolute address using the OCG compiler's @ *address* syntax.

Access Memory

PIC18 Only – Special registers on PIC18 devices that allow access regardless of the setting of the Bank Select Register (BSR).

Access Entry Points

Access entry points provide a way to transfer control across segments to a function which may not be defined at link time. They support the separate linking of boot and secure application segments.

Address

Value that identifies a location in memory.

Alphabetic Character

Alphabetic characters are those characters that are letters of the arabic alphabet (a, b, ..., z, A, B, ..., Z).

Alphanumeric

Alphanumeric characters are comprised of alphabetic characters and decimal digits (0,1, ..., 9).

ANDed Breakpoints

Set up an ANDed condition for breaking, i.e., breakpoint 1 AND breakpoint 2 must occur at the same time before a program halt. This can only be accomplished if a data breakpoint and a program memory breakpoint occur at the same time.

Anonymous Structure

16-bit C Compiler – An unnamed structure.

PIC18 C Compiler – An unnamed structure that is a member of a C union. The members of an anonymous structure may be accessed as if they were members of the enclosing union. For example, in the following code, *hi* and *lo* are members of an anonymous structure inside the union *caster*.

```
union castaway {
    int intval;
    struct {
        char lo; //accessible as caster.lo
        char hi; //accessible as caster.hi
    };
} caster;
```

ANSI

American National Standards Institute is an organization responsible for formulating and approving standards in the United States.

Application

A set of software and hardware that may be controlled by a PIC microcontroller.

Archive/Archiver

An archive/library is a collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the archiver/librarian to combine the object files into one archive/library file. An archive/library can be linked with object modules and other archives/libraries to create executable code.

ASCII

American Standard Code for Information Interchange is a character set encoding that uses 7 binary digits to represent each character. It includes upper and lower case letters, digits, symbols and control characters.

Assembly/Assembler

Assembly is a programming language that describes binary machine code in a symbolic form. An assembler is a language tool that translates assembly language source code into machine code.

Assigned Section

A GCC compiler section which has been assigned to a target memory block in the linker command file.

Asynchronously

Multiple events that do not occur at the same time. This is generally used to refer to interrupts that may occur at any time during processor execution.

Asynchronous Stimulus

Data generated to simulate external inputs to a simulator device.

Attribute

GCC characteristics of variables or functions in a C program which are used to describe machine-specific properties.

Attribute, Section

GCC characteristics of sections, such as “executable”, “readonly”, or “data” that can be specified as flags in the assembler `.section` directive.

B

Binary

The base two numbering system that uses the digits 0-1. The rightmost digit counts ones, the next counts multiples of 2, then $2^2 = 4$, etc.

Breakpoint

Hardware Breakpoint: An event whose execution will cause a halt.

Software Breakpoint: An address where execution of the firmware will halt. Usually achieved by a special break instruction.

Build

Compile and link all the source files for an application.

C

C/C++

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C++ is the object-oriented version of C.

Calibration Memory

A special function register or registers used to hold values for calibration of a PIC microcontroller on-board RC oscillator or other device peripherals.

Central Processing Unit

The part of a device that is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction. When necessary, it works in conjunction with the arithmetic logic unit (ALU) to complete the execution of the instruction. It controls the program memory address bus, the data memory address bus, and accesses to the stack.

Clean

Clean removes all intermediary project files, such as object, hex and debug files, for the active project. These files are recreated from other files when a project is built.

COFF

Common Object File Format. An object file of this format contains machine code, debugging and other information.

Command Line Interface

A means of communication between a program and its user based solely on textual input and output.

Compiled Stack

A region of memory managed by the compiler in which variables are statically allocated space. It replaces a software or hardware stack when such mechanisms cannot be efficiently implemented on the target device.

Compiler

A program that translates a source file written in a high-level language into machine code.

Conditional Assembly

Assembly language code that is included or omitted based on the assembly-time value of a specified expression.

Conditional Compilation

The act of compiling a program fragment only if a certain constant expression, specified by a preprocessor directive, is true.

Configuration Bits

Special-purpose bits programmed to set PIC microcontroller modes of operation. A Configuration bit may or may not be preprogrammed.

Control Directives

Directives in assembly language code that cause code to be included or omitted based on the assembly-time value of a specified expression.

CPU

See Central Processing Unit.

Cross Reference File

A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

D

Data Directives

Data directives are those that control the assembler's allocation of program or data memory and provide a way to refer to data items symbolically; that is, by meaningful names.

Data Memory

On Microchip MCU and DSC devices, data memory (RAM) is comprised of General Purpose Registers (GPRs) and Special Function Registers (SFRs). Some devices also have EEPROM data memory.

Data Monitor and Control Interface (DMCI)

The Data Monitor and Control Interface, or DMCI, is a tool in MPLAB X IDE. The interface provides dynamic input control of application variables in projects. Application-generated data can be viewed graphically using any of 4 dynamically-assignable graph windows.

Debug/Debugger

See ICE/ICD.

Debugging Information

Compiler and assembler options that, when selected, provide varying degrees of information used to debug application code. See compiler or assembler documentation for details on selecting debug options.

Deprecated Features

Features that are still supported for legacy reasons, but will eventually be phased out and no longer used.

Device Programmer

A tool used to program electrically programmable semiconductor devices such as microcontrollers.

Digital Signal Controller

A digital signal controller (DSC) is a microcontroller device with digital signal processing capability, i.e., Microchip dsPIC DSC devices.

Digital Signal Processing\Digital Signal Processor

Digital signal processing (DSP) is the computer manipulation of digital signals, commonly analog signals (sound or image) which have been converted to digital form (sampled). A digital signal processor is a microprocessor that is designed for use in digital signal processing.

Directives

Statements in source code that provide control of the language tool's operation.

Download

Download is the process of sending data from a host to another device, such as an emulator, programmer or target board.

DWARF

Debug With Arbitrary Record Format. DWARF is a debug information format for ELF files.

E

EEPROM

Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

ELF

Executable and Linking Format. An object file of this format contains machine code. Debugging and other information is specified in with DWARF. ELF/DWARF provide better debugging of optimized code than COFF.

Emulation/Emulator

See ICE/ICD.

Endianness

The ordering of bytes in a multi-byte object.

Environment

MPLAB PM3 – A folder containing files on how to program a device. This folder can be transferred to a SD/MMC card.

Epilogue

A portion of compiler-generated code that is responsible for deallocating stack space, restoring registers and performing any other machine-specific requirement specified in the runtime model. This code executes after any user code for a given function, immediately prior to the function return.

EPROM

Erasable Programmable Read Only Memory. A programmable read-only memory that can be erased usually by exposure to ultraviolet radiation.

Error/Error File

An error reports a problem that makes it impossible to continue processing your program. When possible, an error identifies the source file name and line number where the problem is apparent. An error file contains error messages and diagnostics generated by a language tool.

Event

A description of a bus cycle which may include address, data, pass count, external input, cycle type (fetch, R/W), and time stamp. Events are used to describe triggers, breakpoints and interrupts.

Executable Code

Software that is ready to be loaded for execution.

Export

Send data out of the MPLAB IDE in a standardized format.

Expressions

Combinations of constants and/or symbols separated by arithmetic or logical operators.

Extended Microcontroller Mode

In extended microcontroller mode, on-chip program memory as well as external memory is available. Execution automatically switches to external if the program memory address is greater than the internal memory space of the PIC18 device.

Extended Mode (PIC18 MCUs)

In Extended mode, the compiler will utilize the extended instructions (i.e., `ADDFSR`, `ADDULNK`, `CALLW`, `MOVSF`, `MOVSS`, `PUSHL`, `SUBFSR` and `SUBULNK`) and the indexed with literal offset addressing.

External Label

A label that has external linkage.

External Linkage

A function or variable has external linkage if it can be referenced from outside the module in which it is defined.

External Symbol

A symbol for an identifier which has external linkage. This may be a reference or a definition.

External Symbol Resolution

A process performed by the linker in which external symbol definitions from all input modules are collected in an attempt to resolve all external symbol references. Any external symbol references which do not have a corresponding definition cause a linker error to be reported.

External Input Line

An external input signal logic probe line (TRIGIN) for setting an event based upon external signals.

External RAM

Off-chip Read/Write memory.

F

Fatal Error

An error that will halt compilation immediately. No further messages will be produced.

File Registers

On-chip data memory, including General Purpose Registers (GPRs) and Special Function Registers (SFRs).

Filter

Determine by selection what data is included/excluded in a trace display or data file.

Fixup

The process of replacing object file symbolic references with absolute addresses after relocation by the linker.

Flash

A type of EEPROM where data is written or erased in blocks instead of bytes.

FNOP

Forced No Operation. A forced NOP cycle is the second cycle of a two-cycle instruction. Since the PIC microcontroller architecture is pipelined, it prefetches the next instruction in the physical address space while it is executing the current instruction. However, if the current instruction changes the program counter, this prefetched instruction is explicitly ignored, causing a forced NOP cycle.

Frame Pointer

A pointer that references the location on the stack that separates the stack-based arguments from the stack-based local variables. Provides a convenient base from which to access local variables and other values for the current function.

Free-Standing

An implementation that accepts any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (ANSI '89 standard clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` and `<stdint.h>`.

G

GPR

General Purpose Register. The portion of device data memory (RAM) available for general use.

H

Halt

A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

Heap

An area of memory used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order determined at runtime.

Hex Code\Hex File

Hex code is executable instructions stored in a hexadecimal format code. Hex code is contained in a hex file.

Hexadecimal

The base 16 numbering system that uses the digits 0-9 plus the letters A-F (or a-f). The digits A-F represent hexadecimal digits with values of (decimal) 10 to 15. The rightmost digit counts ones, the next counts multiples of 16, then $16^2 = 256$, etc.

High Level Language

A language for writing programs that is further removed from the processor than assembly.

I

ICE/ICD

In-Circuit Emulator/In-Circuit Debugger: A hardware tool that debugs and programs a target device. An emulator has more features than a debugger, such as trace.

In-Circuit Emulation/In-Circuit Debug: The act of emulating or debugging with an in-circuit emulator or debugger.

-ICE/-ICD: A device (MCU or DSC) with on-board in-circuit emulation or debug circuitry. This device is always mounted on a header board and used to debug with an in-circuit emulator or debugger.

ICSP

In-Circuit Serial Programming. A method of programming Microchip embedded devices using serial communication and a minimum number of device pins.

IDE

Integrated Development Environment, as in MPLAB IDE.

Identifier

A function or variable name.

IEEE

Institute of Electrical and Electronics Engineers.

Import

Bring data into the MPLAB IDE from an outside source, such as from a hex file.

Initialized Data

Data which is defined with an initial value. In C,

```
int myVar=5;
```

defines a variable which will reside in an initialized data section.

Instruction Set

The collection of machine language instructions that a particular processor understands.

Instructions

A sequence of bits that tells a central processing unit to perform a particular operation and can contain data to be used in the operation.

Internal Linkage

A function or variable has internal linkage if it can not be accessed from outside the module in which it is defined.

International Organization for Standardization

An organization that sets standards in many businesses and technologies, including computing and communications. Also known as ISO.

Interrupt

A signal to the CPU that suspends the execution of a running application and transfers control to an Interrupt Service Routine (ISR) so that the event may be processed. Upon completion of the ISR, normal execution of the application resumes.

Interrupt Handler

A routine that processes special code when an interrupt occurs.

Interrupt Service Request (IRQ)

An event which causes the processor to temporarily suspend normal instruction execution and to start executing an interrupt handler routine. Some processors have several interrupt request events allowing different priority interrupts.

Interrupt Service Routine (ISR)

Language tools – A function that handles an interrupt.

MPLAB IDE – User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

Interrupt Vector

Address of an interrupt service routine or interrupt handler.

L

L-value

An expression that refers to an object that can be examined and/or modified. An l-value expression is used on the left-hand side of an assignment.

Latency

The time between an event and its response.

Library/Librarian

See Archive/Archiver.

Linker

A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

Linker Script Files

Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

Listing Directives

Listing directives are those directives that control the assembler listing file format. They allow the specification of titles, pagination and other listing control.

Listing File

A listing file is an ASCII text file that shows the machine code generated for each C source statement, assembly instruction, assembler directive, or macro encountered in a source file.

Little Endian

A data ordering scheme for multibyte data whereby the LSb is stored at the lower addresses.

Local Label

A local label is one that is defined inside a macro with the LOCAL directive. These labels are particular to a given instance of a macro's instantiation. In other words, the symbols and labels that are declared as local are no longer accessible after the ENDM macro is encountered.

Logic Probes

Up to 14 logic probes can be connected to some Microchip emulators. The logic probes provide external trace inputs, trigger output signal, +5V, and a common ground.

Loop-Back Test Board

Used to test the functionality of the MPLAB REAL ICE in-circuit emulator.

LVDS

Low Voltage Differential Signaling. A low noise, low-power, low amplitude method for high-speed (gigabits per second) data transmission over copper wire.

With standard I/O signaling, data storage is contingent upon the actual voltage level. Voltage level can be affected by wire length (longer wires increase resistance, which lowers voltage). But with LVDS, data storage is distinguished only by positive and negative voltage values, not the voltage level. Therefore, data can travel over greater lengths of wire while maintaining a clear and consistent data stream.

Source: <http://www.webopedia.com/TERM/L/LVDS.html>.

M

Machine Code

The representation of a computer program that is actually read and interpreted by the processor. A program in binary machine code consists of a sequence of machine instructions (possibly interspersed with data). The collection of all possible instructions for a particular processor is known as its "instruction set".

Machine Language

A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated.

Macro

Macro instruction. An instruction that represents a sequence of instructions in abbreviated form.

Macro Directives

Directives that control the execution and data allocation within macro body definitions.

Makefile

Export to a file the instructions to Make the project. Use this file to Make your project outside of MPLAB IDE, i.e., with a `make`.

Make Project

A command that rebuilds an application, recompiling only those source files that have changed since the last complete compilation.

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also `uC`.

Memory Model

For C compilers, a representation of the memory available to the application. For the PIC18 C compiler, a description that specifies the size of pointers that point to program memory.

Message

Text displayed to alert you to potential problems in language tool operation. A message will not stop operation.

Microcontroller

A highly integrated chip that contains a CPU, RAM, program memory, I/O ports and timers.

Microcontroller Mode

One of the possible program memory configurations of PIC18 microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

Microprocessor Mode

One of the possible program memory configurations of PIC18 microcontrollers. In microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

Mnemonics

Text instructions that can be translated directly into machine code. Also referred to as opcodes.

Module

The preprocessed output of a source file after preprocessor directives have been executed. Also known as a translation unit.

MPASM™ Assembler

Microchip Technology's relocatable macro assembler for PIC microcontroller devices, KeeLoq® devices and Microchip memory devices.

MPLAB Language Tool for Device

Microchip's C compilers, assemblers and linkers for specified devices. Select the type of language tool based on the device you will be using for your application, e.g., if you will be creating C code on a PIC18 MCU, select the MPLAB C Compiler for PIC18 MCUs.

MPLAB ICD

Microchip's in-circuit debuggers that works with MPLAB IDE. See ICE/ICD.

MPLAB IDE

Microchip's Integrated Development Environment. MPLAB IDE comes with an editor, project manager and simulator.

MPLAB PM3

A device programmer from Microchip. Programs PIC18 microcontrollers and dsPIC digital signal controllers. Can be used with MPLAB IDE or stand-alone. Replaces PRO MATE II.

MPLAB REAL ICE™ In-Circuit Emulator

Microchip's next-generation in-circuit emulators that works with MPLAB IDE. See ICE/ICD.

MPLAB SIM

Microchip's simulator that works with MPLAB IDE in support of PIC MCU and dsPIC DSC devices.

MPLIB™ Object Librarian

Microchip's librarian that can work with MPLAB IDE. MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler (mpasm or mpasmwin v2.0) or MPLAB C18 C compiler.

MPLINK™ Object Linker

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip C18 C compiler. MPLINK linker also may be used with the Microchip MPLIB librarian. MPLINK linker is designed to be used with MPLAB IDE, though it does not have to be.

MRU

Most Recently Used. Refers to files and windows available to be selected from MPLAB IDE main pull down menus.

N

Native Data Size

For Native trace, the size of the variable used in a Watch window must be of the same size as the selected device's data memory: bytes for PIC18 devices and words for 16-bit devices.

Nesting Depth

The maximum level to which macros can include other macros.

Node

MPLAB IDE project component.

Non-Extended Mode (PIC18 MCUs)

In Non-Extended mode, the compiler will not utilize the extended instructions nor the indexed with literal offset addressing.

Non Real Time

Refers to the processor at a breakpoint or executing single-step instructions or MPLAB IDE being run in simulator mode.

Non-Volatile Storage

A storage device whose contents are preserved when its power is off.

NOP

No Operation. An instruction that has no effect when executed except to advance the program counter.

O

Object Code/Object File

Object code is the machine code generated by an assembler or compiler. An object file is a file containing machine code and possibly debug information. It may be immediately executable or it may be relocatable, requiring linking with other object files, e.g., libraries, to produce a complete executable program.

Object File Directives

Directives that are used only when creating an object file.

Octal

The base 8 number system that only uses the digits 0-7. The rightmost digit counts ones, the next digit counts multiples of 8, then $8^2 = 64$, etc.

Off-Chip Memory

Off-chip memory refers to the memory selection option for the PIC18 device where memory may reside on the target board, or where all program memory may be supplied by the emulator. The **Memory** tab accessed from *Options>Development Mode* provides the Off-Chip Memory selection dialog box.

Opcodes

Operational Codes. See Mnemonics.

Operators

Symbols, like the plus sign '+' and the minus sign '-', that are used when forming well-defined expressions. Each operator has an assigned precedence that is used to determine order of evaluation.

OTP

One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

P

Pass Counter

A counter that decrements each time an event (such as the execution of an instruction at a particular address) occurs. When the pass count value reaches zero, the event is satisfied. You can assign the Pass Counter to break and trace logic, and to any sequential event in the complex trigger dialog.

PC

Personal Computer or Program Counter.

PC Host

Any PC running a supported Windows operating system.

Persistent Data

Data that is never cleared or initialized. Its intended use is so that an application can preserve data across a device reset.

Phantom Byte

An unimplemented byte in the dsPIC architecture that is used when treating the 24-bit instruction word as if it were a 32-bit instruction word. Phantom bytes appear in dsPIC hex files.

PIC MCUs

PIC microcontrollers (MCUs) refers to all Microchip microcontroller families.

PICKit 2 and 3

Microchip's developmental device programmers with debug capability through Debug Express. See the Readme files for each tool to see which devices are supported.

Plug-ins

The MPLAB IDE has both built-in components and plug-in modules to configure the system for a variety of software and hardware tools. Several plug-in tools may be found under the Tools menu.

Pod

The enclosure for an in-circuit emulator or debugger. Other names are "Puck", if the enclosure is round, and "Probe", not be confused with logic probes.

Power-on-Reset Emulation

A software randomization process that writes random values in data RAM areas to simulate uninitialized values in RAM upon initial power application.

Pragma

A directive that has meaning to a specific compiler. Often a pragma is used to convey implementation-defined information to the compiler. MPLAB C30 uses attributes to convey this information.

Precedence

Rules that define the order of evaluation in expressions.

Production Programmer

A production programmer is a programming tool that has resources designed in to program devices rapidly. It has the capability to program at various voltage levels and completely adheres to the programming specification. Programming a device as fast as possible is of prime importance in a production environment where time is of the essence as the application circuit moves through the assembly line.

Profile

For MPLAB SIM simulator, a summary listing of executed stimulus by register.

Program Counter

The location that contains the address of the instruction that is currently executing.

Program Counter Unit

16-bit assembler – A conceptual representation of the layout of program memory. The program counter increments by 2 for each instruction word. In an executable section, 2 program counter units are equivalent to 3 bytes. In a read-only section, 2 program counter units are equivalent to 2 bytes.

Program Memory

MPLAB IDE – The memory area in a device where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

16-bit assembler/compiler – The memory area in a device where instructions are stored.

Project

A project contains the files needed to build an application (source code, linker script files, etc.) along with their associations to various build tools and build options.

Prologue

A portion of compiler-generated code that is responsible for allocating stack space, preserving registers and performing any other machine-specific requirement specified in the runtime model. This code executes before any user code for a given function.

Prototype System

A term referring to a user's target application, or target board.

Psect

The OCG equivalent of a GCC section, short for program section. A block of code or data which is treated as a whole by the linker.

PWM Signals

Pulse Width Modulation Signals. Certain PIC MCU devices have a PWM peripheral.

Q

Qualifier

An address or an address range used by the Pass Counter or as an event before another operation in a complex trigger.

R

Radix

The number base, hex, or decimal, used in specifying an address.

RAM

Random Access Memory (Data Memory). Memory in which information can be accessed in any order.

Raw Data

The binary representation of code or data associated with a section.

Read Only Memory

Memory hardware that allows fast access to permanently stored data but prevents addition to or modification of the data.

Real Time

When an in-circuit emulator or debugger is released from the halt state, the processor runs in Real Time mode and behaves exactly as the normal chip would behave. In Real Time mode, the real time trace buffer of an emulator is enabled and constantly captures all selected cycles, and all break logic is enabled. In an in-circuit emulator or debugger, the processor executes in real time until a valid breakpoint causes a halt, or until the user halts the execution.

In the simulator, real time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

Recursive Calls

A function that calls itself, either directly or indirectly.

Recursion

The concept that a function or macro, having been defined, can call itself. Great care should be taken when writing recursive macros; it is easy to get caught in an infinite loop where there will be no exit from the recursion.

Reentrant

A function that may have multiple, simultaneously active instances. This may happen due to either direct or indirect recursion or through execution during interrupt processing.

Relaxation

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. MPLAB ASM30 currently knows how to RELAX a CALL instruction into an RCALL instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

Relocatable

An object whose address has not been assigned to a fixed location in memory.

Relocatable Section

16-bit assembler – A section whose address is not fixed (absolute). The linker assigns addresses to relocatable sections through a process called relocation.

Relocation

A process performed by the linker in which absolute addresses are assigned to relocatable sections and all symbols in the relocatable sections are updated to their new addresses.

ROM

Read Only Memory (Program Memory). Memory that cannot be modified.

Run

The command that releases the emulator from halt, allowing it to run the application code and change or respond to I/O in real time.

Run-time Model

Describes the use of target architecture resources.

Runtime Watch

A Watch window where the variables change in as the application is run. See individual tool documentation to determine how to set up a runtime watch. Not all tools support runtime watches.

S

Scenario

For MPLAB SIM simulator, a particular setup for stimulus control.

Section

The GCC equivalent of an OCG psect. A block of code or data which is treated as a whole by the linker.

Section Attribute

A GCC characteristic ascribed to a section (e.g., an `access` section).

Sequenced Breakpoints

Breakpoints that occur in a sequence. Sequence execution of breakpoints is bottom-up; the last breakpoint in the sequence occurs first.

Serialized Quick Turn Programming

Serialization allows you to program a serial number into each microcontroller device that the Device Programmer programs. This number can be used as an entry code, password or ID number.

Shell

The MPASM assembler shell is a prompted input interface to the macro assembler. There are two MPASM assembler shells: one for the DOS version and one for the Windows version.

Simulator

A software program that models the operation of devices.

Single Step

This command steps through code, one instruction at a time. After each instruction, MPLAB IDE updates register windows, watch variables, and status displays so you can analyze and debug instruction execution. You can also single step C compiler source code, but instead of executing single instructions, MPLAB IDE will execute all assembly level instructions generated by the line of the high level C statement.

Skew

The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed opcodes appears on the bus as a fetch during the execution of the previous instruction, the source data address and value and the destination data address appear when the opcodes is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

Skid

When a hardware breakpoint is used to halt the processor, one or more additional instructions may be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

Source Code

The form in which a computer program is written by the programmer. Source code is written in a formal programming language which can be translated into machine code or executed by an interpreter.

Source File

An ASCII text file containing source code.

Special Function Registers (SFRs)

The portion of data memory (RAM) dedicated to registers that control I/O processor functions, I/O status, timers or other modes or peripherals.

SQTP

See Serialized Quick Turn Programming.

Stack, Hardware

Locations in PIC microcontroller where the return address is stored when a function call is made.

Stack, Software

Memory used by an application for storing return addresses, function parameters, and local variables. This memory is dynamically allocated at runtime by instructions in the program. It allows for reentrant function calls.

Stack, Compiled

A region of memory managed and allocated by the compiler in which variables are statically assigned space. It replaces a software stack when such mechanisms cannot be efficiently implemented on the target device. It precludes reentrancy.

MPLAB Starter Kit for Device

Microchip's starter kits contains everything needed to begin exploring the specified device. View a working application and then debug and program your own changes.

Static RAM or SRAM

Static Random Access Memory. Program memory you can read/write on the target board that does not need refreshing frequently.

Status Bar

The Status Bar is located on the bottom of the MPLAB IDE window and indicates such current information as cursor position, development mode and device, and active tool bar.

Step Into

This command is the same as Single Step. Step Into (as opposed to Step Over) follows a CALL instruction into a subroutine.

Step Over

Step Over allows you to debug code without stepping into subroutines. When stepping over a CALL instruction, the next breakpoint will be set at the instruction after the CALL. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached. The Step Over command is the same as Single Step except for its handling of CALL instructions.

Step Out

Step Out allows you to step out of a subroutine which you are currently stepping through. This command executes the rest of the code in the subroutine and then stops execution at the return address to the subroutine.

Stimulus

Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous (pin), clocked and register.

Stopwatch

A counter for measuring execution cycles.

Storage Class

Determines the lifetime of the memory associated with the identified object.

Storage Qualifier

Indicates special properties of the objects being declared (e.g., `const`).

Symbol

A symbol is a general purpose mechanism for describing the various pieces which comprise a program. These pieces include function names, variable names, section names, file names, struct/enum/union tag names, etc. Symbols in MPLAB IDE refer mainly to variable names, function names and assembly labels. The value of a symbol after linking is its value in memory.

Symbol, Absolute

Represents an immediate value such as a definition through the assembly `.equ` directive.

System Window Control

The system window control is located in the upper left corner of windows and some dialogs. Clicking on this control usually pops up a menu that has the items "Minimize," "Maximize," and "Close."

T**Target**

Refers to user hardware.

Target Application

Software residing on the target board.

Target Board

The circuitry and programmable device that makes up the target application.

Target Processor

The microcontroller device on the target application board.

Template

Lines of text that you build for inserting into your files at a later time. The MPLAB Editor stores templates in template files.

Tool Bar

A row or column of icons that you can click on to execute MPLAB IDE functions.

Trace

An emulator or simulator function that logs program execution. The emulator logs program execution into its trace buffer which is uploaded to MPLAB IDE's trace window.

Trace Memory

Trace memory contained within the emulator. Trace memory is sometimes called the trace buffer.

Trace Macro

A macro that will provide trace information from emulator data. Since this is a software trace, the macro must be added to code, the code must be recompiled or reassembled, and the target device must be programmed with this code before trace will work.

Trigger Output

Trigger output refers to an emulator output signal that can be generated at any address or address range, and is independent of the trace and breakpoint settings. Any number of trigger output points can be set.

Trigraphs

Three-character sequences, all starting with ??, that are defined by ISO C as replacements for single characters.

U**Unassigned Section**

A section which has not been assigned to a specific target memory block in the linker command file. The linker must find a target memory block in which to allocate an unassigned section.

Uninitialized Data

Data which is defined without an initial value. In C,

```
int myVar;
```

defines a variable which will reside in an uninitialized data section.

Upload

The Upload function transfers data from a tool, such as an emulator or programmer, to the host PC or from the target board to the emulator.

USB

Universal Serial Bus. An external peripheral interface standard for communication between a computer and external peripherals over a cable using bi-serial transmission. USB 1.0/1.1 supports data transfer rates of 12 Mbps. Also referred to as high-speed USB, USB 2.0 supports data rates up to 480 Mbps.

V

Vector

The memory locations that an application will jump to when either a reset or interrupt occurs.

Volatile

A variable qualifier which prevents the compiler applying optimizations that affect how the variable is accessed in memory.

W

Warning

MPLAB IDE – An alert that is provided to warn you of a situation that would cause physical damage to a device, software file, or equipment.

16-bit assembler/compiler – Warnings report conditions that may indicate a problem, but do not halt processing. In MPLAB C30, warning messages report the source file name and line number, but include the text 'warning:' to distinguish them from error messages.

Watch Variable

A variable that you may monitor during a debugging session in a Watch window.

Watch Window

Watch windows contain a list of watch variables that are updated at each breakpoint.

Watchdog Timer (WDT)

A timer on a PIC microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using Configuration bits.

Workbook

For MPLAB SIM stimulator, a setup for generation of SCL stimulus.

NOTES:

Index

Symbols

__section() specifier	34
_asm directive	45
#asm directive	45

A

absolute functions	34
absolute variables	34
anonymous structures	32
asm statement	45
aspc18 application	44
assembler applications	44
assembly code	14, 44
assembly instruction destination specifier	44
assembly symbols	44
auto specifier	28, 37

B

bankx qualifiers	34
bit data type	27
bit-fields	32

C

C and assembly	44
C18 compatibility mode	10, 12
CCI	14
char data type	27
clearing static objects	40
ClrWdt macro	26
code pragma	32, 34
CODE assembler directive	45
COFF files	16
command-line format	16
command-line options	18
command-line usage	15
common C interface	14
compiled stack	36
compiler messages	20
compiler-managed resources	41
config pragma	25
configuration bits	25
const qualifier	29
Customer Notification Service	8
Customer Support	8

D

data stack	36
data types	27
delay built-ins, EEPROM data	26
Documentation	
Conventions	6
Layout	5
driver applications	15
driver options	18

E

ELF files	16
-----------------	----

entry function	40
errors	20
extended instruction set	19, 44
extended mode	24

F

fake mccc18 driver	12
far qualifier	29
FCALL instruction	44
floating-point data types	27
function calling convention	38
function parameters	38
function return values	38
function variants	31

G

GLOBAL assembler directive	45
----------------------------------	----

H

header files	25
hybrid stack model	36

I

IDATA assembler directive	45
idata pragma	34
implementation defined behavior	13
indicating object locations	35
in-line assembly	45
instruction set	24
integer promotions	13, 24
intermediate files	16
Internet Address	7
interrupt specifier	32
interrupts	32

L

library files	14
linker options to position sections	34
linker scripts	14, 47
linking	47
LJMP instruction	44
locating objects	34
low_priority specifier	32

M

macros	26, 42
macros with variable argument lists	43
maximum function size	28
maximum object size	28
mccc18 application	15
mccc18 driver	12
memory models	24
messages	20
Microchip Internet Web Site	7
migration	10
MOVFW instruction	44
MPASM	44

MPLAB® C18 to XC8 C Compiler Migration Guide

MPLAB IDE	19	structure bit-fields	32
mplib application xc8 application.....	15	structure types	32
mplink application.....	15	Swap macro	26
mplink options	22	T	
N		temporary data location	35
non-extended mode	24	tmpdata pragma.....	35
nonreentrant specifier	37	U	
Nop macro.....	26	UDATA assembler directive.....	45
O		udata pragma.....	34
object files	14	union types	32
operating modes	24	USB	66
optimizations	20	V	
options.....	18	varlocate pragma	35
overlay specifier	28, 37	W	
P		warnings	20
parameters	38	Watchdog Timer	67
p-code files.....	16	WWW Address	7
p-code libraries.....	16	X	
placing objects at an address.....	34	xc.h header	25
placing objects in a bank.....	34	xc.inc header	25
pointer sizes	30		
pointer storage qualifiers	30		
pointer variables.....	24		
powerup routine	40		
predefined macros	26, 42		
preprocessing.....	42		
preprocessor commands.....	20		
preprocessor macros	42		
project migration.....	10		
PSECT assembler directive	45		
R			
ram qualifier	29		
Reading, Recommended	7		
Readme.....	7		
reentrancy	36		
reentrant specifier	37		
register usage	41		
regsused pragma	33		
RETFIE instruction	44		
return values	38		
Rlcf macro	26		
rom qualifier	29		
romdata pragma.....	34		
RUNTIME driver option	40		
runtime startup code	40		
S			
SECTION linker script directive.....	47		
sectiontype pragma.....	34		
SFRs	25		
short long int type.....	27		
size limitations.....	28		
size of data types	27		
size of functions	28		
size of objects	28		
Sleep macro	26		
software stack	36		
special function registers.....	25		
stack.....	36		
STACK driver option	37		
static objects	40		
static specifier	28, 37		
storage classes	28		

NOTES:

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:
<http://www.microchip.com/support>
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hangzhou
Tel: 86-571-2819-3187
Fax: 86-571-2819-3189

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880-3770
Fax: 81-3-6880-3771

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7828
Fax: 886-7-330-9305

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820