MPLAB® ASSEMBLER, LINKER AND UTILITIES FOR PIC24 MCUs AND dsPIC® DSCs USER'S GUIDE

© 2008 Microchip Technology Inc.

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in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates.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, Accuron, dsPIC, KEELOQ, KEELOQ logo, , PIC, PICmicro, PICSTART, PRO MATE, rfPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Linear Active Thermistor, MXDEV, MXLAB, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, Certified logo, MPLIB, MPLINK, mTouch, PICkit, PICDEM, PICDEM.net, PICtail, PIC^{32} logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rfLAB, Select Mode, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

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

© 2008, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 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, KEELOQ® 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.

© 2008 Microchip Technology Inc.

MPLAB® ASSEMBLER, LINKER AND UTILITIES FOR PIC24 MCUs AND dsPIC® DSCs USER'S GUIDE

Table of Contents

Preface 1

Part 1 – MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs

Chapter 1. Assembler Overview

1.1 Introduction 11
1.2 Highlights 11
1.3 Assembler and Other Development Tools 11
1.4 Feature Set 12
1.5 Input/Output Files 13

Chapter 2. Assembler Command Line Interface

2.1 Introduction 17
2.2 Highlights 17
2.3 Syntax 17
2.4 Options that Modify the Listing Output 18
2.5 Options that Control Informational Output 28
2.6 Options that Control Output File Creation 29
2.7 Other Options 30

Chapter 3. Assembler Syntax

3.1 Introduction 31
3.2 Highlights 31
3.3 Internal Preprocessor 31
3.4 Source Code Format 32
3.5 Constants 35
3.6 Summary 37

Chapter 4. Assembler Expression Syntax and Operation

4.1 Introduction 39
4.2 Highlights 39
4.3 Expressions 39
4.4 Operators 40
4.5 Special Operators 41

© 2008 Microchip Technology Inc.

Chapter 5. Assembler Symbols

5.1 Introduction 45
5.2 Highlights 45
5.3 What are Symbols 45
5.4 Reserved Names 45
5.5 Local Symbols 46
5.6 Giving Symbols Other Values 47
5.7 The Special DOT Symbol 47
5.8 Using Executable Symbols in a Data Context 47
5.9 Predefined Symbols 48

Chapter 6. Assembler Directives

6.1 Introduction 49
6.2 Highlights 49
6.3 Directives that Define Sections 50
6.4 Directives that Fill Program Memory 56
6.5 Directives that Initialize Constants 58
6.6 Directives that Declare Symbols 62
6.7 Directives that Define Symbols 63
6.8 Directives that Modify Section Alignment 64
6.9 Directives that Format the Output Listing 69
6.10 Directives that Control Conditional Assembly 70
6.11 Directives for Substitution/Expansion 71
6.12 Miscellaneous Directives 74
6.13 Directives for Debug Information 76

Part 2 – MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs

Chapter 7. Linker Overview

7.1 Introduction 81
7.2 Highlights 81
7.3 Linker and Other Development Tools 81
7.4 Feature Set 83
7.5 Input/Output Files 83

Chapter 8. Linker Command Line Interface

8.1 Introduction 87
8.2 Highlights 87
8.3 Syntax 87
8.4 Options that Control Output File Creation 88
8.5 Options that Control Run-time Initialization 94
8.6 Options that Control Informational Output 95
8.7 Options that Modify the Link Map Output 98
8.8 Options that Specify CodeGuard™ Security Features 99

© 2008 Microchip Technology Inc.

Chapter 9. Linker Scripts
9.1 Introduction 101
9.2 Highlights 101
9.3 Overview of Linker Scripts 101
9.4 Command Line Information 102
9.5 Contents of a Linker Script 102
9.6 Creating a Custom Linker Script 113
9.7 Linker Script Command Language 113
9.8 Expressions in Linker Scripts 128

Chapter 10. Linker Processing
10.1 Introduction 135
10.2 Highlights 135
10.3 Overview of Linker Processing 135
10.4 Memory Addressing 137
10.5 Linker Allocation 139
10.6 Global and Weak Symbols 143
10.7 Handles 143
10.8 Initialized Data 144
10.9 Read-only Data 147
10.10 Stack Allocation 149
10.11 Heap Allocation 150
10.12 Interrupt Vector Tables 150
10.13 Optimizing Memory Usage 161
10.14 Boot and Secure Segments 164
10.15 Notable Symbols 165

Chapter 11. Linker Examples
11.1 Introduction 167
11.2 Highlights 167
11.3 Memory Addresses and Relocatable Code 168
11.4 Locating a Variable at a Specific Address 169
11.5 Locating a Function at a Specific Address 169
11.6 Saving and Restoring the PSVPAG Register 170
11.7 Locating a Constant at a Specific Address in Program Memory 171
11.8 Locating and Accessing Data in EEPROM Memory 172
11.9 Creating an Incrementing Modulo Buffer in X Memory 174
11.10 Creating a Decrementing Modulo Buffer in Y Memory 174
11.11 Locating the Stack at a Specific Address 175
11.12 Locating and Reserving Program Memory 175

Part 3 – MPLAB Object Archiver/Librarian for PIC24 MCUs and dsPIC® DSCs

Chapter 12. MPLAB Object Archiver/Librarian for 16-bit Devices

12.1 Introduction 179
12.2 Highlights 179
12.3 Arichiver/Libraian and Other Development Tools 180
12.4 Feature Set 180
12.5 Input/Output Files 180
12.6 Syntax 181
12.7 Options 181
12.8 Scripts 183

Part 4 – Utilities

Chapter 13. Utilities Overview

13.1 Introduction 187
13.2 Highlights 187
13.3 What are Utilities 187

Chapter 14. pic30-bin2hex Utility

14.1 Introduction 189
14.2 Highlights 189
14.3 Input/Output Files 189
14.4 Syntax 190
14.5 Options 190

Chapter 15. pic30-nm Utility

15.1 Introduction 191
15.2 Highlights 191
15.3 Input/Output Files 191
15.4 Syntax 191
15.5 Options 192
15.6 Output Formats 193

Chapter 16. pic30-objdump Utility

16.1 Introduction 195
16.2 Highlights 195
16.3 Input/Output Files 195
16.4 Syntax 195
16.5 Options 196

Chapter 17. pic30-ranlib Utility

17.1 Introduction 199
17.2 Highlights 199
17.3 Input/Output Files 199
17.4 Syntax 199
17.5 Options 199

© 2008 Microchip Technology Inc.

Chapter 18. pic30-strings Utility
18.1 Introduction 201
18.2 Highlights 201
18.3 Input/Output Files 201
18.4 Syntax 201
18.5 Options 202

Chapter 19. pic30-strip Utility
19.1 Introduction 203
19.2 Highlights 203
19.3 Input/Output Files 203
19.4 Syntax 203
19.5 Options 204

Chapter 20. pic30-lm Utility
20.1 Introduction 205
20.2 Highlights 205
20.3 Syntax 205
20.4 Options 205

Part 5 – Command-Line Simulator

Chapter 21. SIM30 Command-Line Simulator
21.1 Introduction 209
21.2 Highlights 209
21.3 Syntax 209
21.4 Options 210

Part 6 – Appendices

Appendix A. Assembler Errors/Warnings/Messages
A.1 Introduction 215
A.2 Highlights 215
A.3 Fatal Errors 215
A.4 Errors 216
A.5 Warnings 223
A.6 Messages 228

Appendix B. Linker Errors/Warnings
B.1 Introduction 229
B.2 Highlights 229
B.3 Errors 229
B.4 Warnings 234

Appendix C. Deprecated Features
C.1 Introduction 237
C.2 Highlights 237
C.3 Assembler Directives that Define Sections 237
C.4 Reserved Section Names with Implied Attributes 238

Appendix D. MPASM™ Assembler Compatibility
D.1 Introduction 239
D.2 Highlights 239
D.3 Compatibility 239
D.4 Examples 242
D.5 Converting PIC18F MCU Assembly Code to dsPIC30F DSC Assembly Code 243

Appendix E. MPLINK™ Linker Compatibility
E.1 Introduction 249
E.2 Highlights 249
E.3 Compatibility 249
E.4 Migration to the 16-Bit Linker 249

Appendix F. MPLIB™ Librarian Compatibility
F.1 Introduction 251
F.2 Highlights 251
F.3 Compatibility 251
F.4 Examples 252

Appendix G. Useful Tables
G.1 Introduction 253
G.2 Highlights 253
G.3 ASCII Character Set 253
G.4 Hexadecimal to Decimal Conversion 254

Appendix H. GNU Free Documentation License
H.1 Preamble 255
H.2 Applicability and Definitions 255
H.3 Verbatim Copying 257
H.4 Copying In Quantity 257
H.5 Modifications 258
H.6 Combining Documents 259
H.7 Collections of Documents 259
H.8 Aggregation with Independent Works 260
H.9 Translation 260
H.10 Termination 260
H.11 Future Revisions of this License 260

Glossary 261
Index 271
Worldwide Sales and Service 282

© 2008 Microchip Technology Inc.

MPLAB® ASSEMBLER, LINKER AND UTILITIES FOR PIC24 MCUs AND dsPIC® DSCs USER'S 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 "DSXXXXXA", where "XXXXX" is the document number and "A" is the revision level of the document.

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

INTRODUCTION

This chapter contains general information that will be useful to know before using 16-bit language tools. Items discussed include:

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

DOCUMENT LAYOUT

This document describes how to use GNU language tools to write code for 16-bit applications. The document layout is as follows:

Part 1 – MPLAB® Assembler for PIC24 MCUs and dsPIC® DSCs

- **Chapter 1: Assembler Overview** – gives an overview of assembler operation.
- **Chapter 2: Assembler Command Line Interface** – details command line options for the assembler.
- **Chapter 3: Assembler Syntax** – describes syntax used with the assembler.
- **Chapter 4: Assembler Expression Syntax and Operation** – provides guidelines for using complex expressions in assembler source files.
- **Chapter 5: Assembler Symbols** – describes what symbols are and how to use them.
- **Chapter 6: Assembler Directives** – details the available assembler directives.

Part 2 – MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs

- **Chapter 7: Linker Overview** – gives an overview of linker operation.
- **Chapter 8: Linker Command Line Interface** – details command line options for the linker.
- **Chapter 9: Linker Scripts** – describes how to generate and use linker scripts to control linker operation.
- **Chapter 10: Linker Processing** – discusses how the linker builds an application from input files.
- **Chapter 11: Linker Examples** – discusses a number of 16-bit specific linker examples and shows the equivalent syntax in C and assembly language.

Part 3 – MPLAB Object Archiver/Librarian for PIC24 MCUs and dsPIC® DSCs

- **Chapter 12: Archiver/Librarian** – details command line options for the librarian.

Part 4 – Utilities

- **Chapter 13: Utilities Overview** – gives an overview of utilities and their operation.
- **Chapter 14: pic30-bin2hex Utility** – details command line options for binary-to-hexadecimal conversion.
- **Chapter 15: pic30-nm Utility** – details command line options for listing symbols in an object file.
- **Chapter 16: pic30-objdump Utility** – details command line options for displaying information about object files.
- **Chapter 17: pic30-ranlib Utility** – details command line options for creating an archive index.
- **Chapter 18: pic30-strings Utility** – details command line options for printing character sequences.
- **Chapter 19: pic30-strip Utility** – details command line options for discarding all symbols from an object file.
- **Chapter 20: pic30-lm Utility** – details command line options for displaying information about the 16-bit compiler license.

Part 5 – Command-Line Simulator

- **Chapter 21: SIM30 Command Line Simulator** – describes the command line simulator that supports 16-bit tools.

© 2008 Microchip Technology Inc.

Part 6 – Appendices

- **Appendix A: Assembler Errors/Warnings/Messages** – contains a descriptive list of the errors, warnings and messages generated by the 16-bit assembler.
- **Appendix B: Linker Errors/Warnings** – contains a descriptive list of the errors and warnings generated by the 16-bit linker.
- **Appendix C: Deprecated Features –** describes features that are considered obsolete.
- **Appendix D: MPASM™ Assembler Compatibility** – contains information on compatibility with MPASM assembler (for 8-bit devices), examples and recommendations for migration to MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs.
- **Appendix E: MPLINK™ Object Linker Compatibility** – contains information on compatibility with MPLINK linker (for 8-bit devices), examples and recommendations for migration to MPLAB Linker for PIC24 MCUs and dsPIC® DSCs.
- **Appendix F: MPLIB™ Object Librarian Compatibility** – contains information on compatibility with MPLIB librarian (for 8-bit devices), examples and recommendations for migration to MPLAB Archiver/Librarian for PIC24 MCUs and dsPIC® DSCs.
- **Appendix G: Useful Tables** – lists some useful tables: the ASCII character set and hexadecimal to decimal conversion.
- **Appendix H: GNU Free Documentation License** – details the license requirements for using the GNU language tools.

CONVENTIONS USED IN THIS GUIDE

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	*File>Save*
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier font:		
Plain Courier	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	A variable argument	*`file`*`.o`, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`mpasmwin [options] file [options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void) { ... }`
Icon		
	This feature supported only in the full version of the software.	
	This feature is not supported on all devices. Devices supported will be listed in the title or text.	

© 2008 Microchip Technology Inc.

RECOMMENDED READING

This documentation describes how to use 16-bit language tools. 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 Microchip tools, read the associated Readme files (HTML files) included with the software.

16-Bit Language Tools Getting Started (DS70094)

A guide to installing and working with the Microchip language tools for 16-bit devices. Examples using the 16-bit simulator SIM30 (a component of MPLAB SIM) are provided.

MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User's Guide (DS51284)

A guide to using the 16-bit C compiler. The 16-bit linker is used with this tool.

16-Bit Language Tools Libraries (DS51456)

A descriptive listing of libraries available for Microchip 16-bit devices. This includes standard (including math) libraries and compiler built-in functions. DSP and 16-bit peripheral libraries are described in Readme files provided with each peripheral library type.

Device-Specific Documentation

The Microchip website contains many documents that describe 16-bit device functions and features. Among these are:

- Individual and family data sheets
- Family reference manuals
- Programmer's reference manuals

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

DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata 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, assemblers, linkers and other language tools. These include all MPLAB C compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
- **Emulators** – The latest information on Microchip in-circuit emulators.These include the MPLAB REAL ICE™, MPLAB ICE 2000 and MPLAB ICE 4000 in-circuit emulators
- **In-Circuit Debuggers** – The latest information on Microchip in-circuit debuggers. These include the MPLAB ICD 2 in-circuit debugger and PICkit™ 2 debug express.
- **MPLAB® IDE** – The latest information on Microchip MPLAB IDE, the Windows® Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the MPLAB PM3 and PRO MATE® II device programmers and the PICSTART® Plus and PICkit 1 and 2 development programmers.

© 2008 Microchip Technology Inc.

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

NOTES:

© 2008 Microchip Technology Inc.

MPLAB® ASSEMBLER, LINKER AND UTILITIES FOR PIC24 MCUs AND dsPIC® DSCs USER'S GUIDE

Part 1 – MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs

Chapter 1. Assembler Overview 11
Chapter 2. Assembler Command Line Interface 17
Chapter 3. Assembler Syntax..... 31
Chapter 4. Assembler Expression Syntax and Operation..... 39
Chapter 5. Assembler Symbols 45
Chapter 6. Assembler Directives 49

NOTES:

© 2008 Microchip Technology Inc.

Chapter 1. Assembler Overview

1.1 INTRODUCTION

MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB ASM30) produces relocatable machine code from symbolic assembly language for the dsPIC30F/33F DSC and PIC24X MCU family of devices. The assembler is a Windows console application that provides a platform for developing assembly language code. The assembler is a port of the GNU assembler from the Free Software Foundation.

1.2 HIGHLIGHTS

Topics covered in this chapter are:

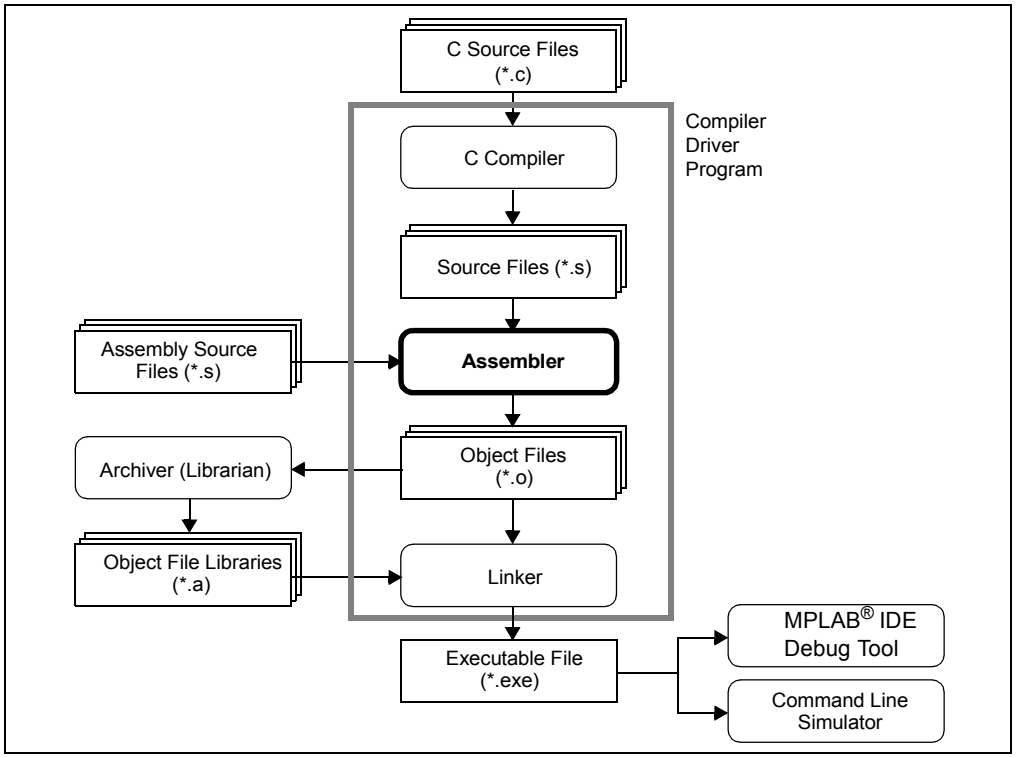
- Assembler and Other Development Tools
- Feature Set
- Input/Output Files

1.3 ASSEMBLER AND OTHER DEVELOPMENT TOOLS

MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs translates user assembly source files. In addition, the MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs uses the assembler to produce its object file. The assembler generates relocatable object files that can then be put into an archive or linked with other relocatable object files and archives to create an executable file. See Figure 1-1 for an overview of the tools process flow.

For information on using the assembler with MPLAB IDE, see *"16-Bit Language Tools Getting Started*" (DS70094).

FIGURE 1-1: TOOLS PROCESS FLOW



1.4 FEATURE SET

Notable features of the assembler include:

- Support for the entire 16-bit instruction set
- Support for fixed-point and floating-point data
- Support for COFF and ELF object formats
- Available for Windows
- Command Line Interface
- Rich Directive Set
- Flexible Macro Language
- Integrated component of MPLAB® IDE

© 2008 Microchip Technology Inc.

1.5 INPUT/OUTPUT FILES

Standard assembler input and output files are listed below.

Extension	Description
Input	
.s	Source File
Output	
.o	Object File
.lst	Listing File

Unlike the MPASM™ assembler (for use with 8-bit PIC® MCUs), MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs does not generate error files, hex files, or symbol and debug files. The assembler is capable of creating a listing file and a relocatable object file (that may or may not contain debugging information). MPLAB Linker for PIC24 MCUs and dsPIC® DSCs is used with the assembler to produce the final object files, map files and final executable file for debugging with MPLAB IDE (see Figure 1-1).

1.5.1 Source File

The assembler accepts, as input, a source file that consists of dsPIC30FXXXX instructions, assembler directives and comments. A sample source file is shown in Example 1-1.

Note: Microchip Technology strongly suggests a .s extension for assembly source files. This will enable you to easily use the C compiler driver without having to specify the option to tell the driver that the file should be treated as an assembly file. See the "*MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User's Guide"* (DS51284) for more details on the C compiler driver.

EXAMPLE 1-1: SAMPLE ASSEMBLER CODE

```
        .title " Sample dsPIC Assembler Source Code"
        .sbttl " For illustration only."

        ; dsPIC registers
        .equ CORCONL, CORCON
        .equ PSV,2

        .section .const,psv
hello:
        .ascii "Hello world!\n\0"

        .text
        .global __reset
__reset:
        ; set PSVPAG to page that contains 'hello'
        mov     #psvpage(hello),w0
        mov     w0,PSVPAG

        ; enable Program Space Visibility
        bset.b  CORCONL,#PSV

        ; make a pointer to 'hello'
        mov     #psvoffset(hello),w0

        .end
```

For more information, see also **Chapter 3. "Assembler Syntax"** and **Chapter 6. "Assembler Directives"**.

1.5.2 Object File

The assembler creates a relocatable object file. These object files do not yet have addresses resolved and must be linked before they can be used for executables.

By default, the name of the object file created is `a.out`. Specify the `-o` option (See **Chapter 2. "Assembler Command Line Interface"**) on the command line to override the default name.

By default, object files are created in the COFF format. To specify COFF or ELF format explicitly, use the `-omf` option on the command line, as shown:

```
pic30-as -omf=coff test.s
pic30-as -omf=elf test2.s
```

Alternatively, the environment variable `PIC30_OMF` may be used to specify object file format for the dsPIC30F language tools.

© 2008 Microchip Technology Inc.

1.5.3 Listing File

The assembler has the capability to produce listing files. These listing files are not absolute listing files, and the addresses that appear in the listing are relative to the start of sections.

By default, the listing file is displayed on standard output. Specify the `-a=<file>` option (See **Chapter 2. "Assembler Command Line Interface"**) on the command line to send the listing file to the specified `file`.

The listing files produced by the assembler are composed of the elements listed below. Example 1-2 shows a sample listing file.

- **Header** – contains the name of the assembler, the name of the file being assembled, and a page number. This is not shown if the `-an` option is specified.
- **Title Line** – contains the title specified by the `.title` directive. This is not shown if the `-an` option is specified.
- **Subtitle** – contains the subtitle specified by the `.sbttl` directive. This is not shown if the `-an` option is specified.
- **High-level source** if the `-ah` option is given to the assembler. The format for high-level source is:
  ```
  <line #>:<filename>       **** <source>
  ```
  For example:
  ```
  1:hello.c       **** #include <stdio.h>
  ```
- **Assembler source** if the `-al` option is given to the assembler. The format for assembler source is:
  ```
  <line #> <addr> <encoded bytes> <source>
  ```
  For example:
  ```
  245 000004 00 0F 78            mov     w0,[w14]
  ```

Note 1:	Line numbers may be repeated.
2:	Addresses are relative to sections in this module and are not absolute.
3:	Instructions are encoded in "little endian" order.

- **Symbol table** if the `-as` option is given to the assembler. Both, a list of defined and undefined symbols will be given. The defined symbols will have the format:
  ```
  DEFINED SYMBOLS
  <filename>:<line #> <section>:<addr> <symbol>
  ```
  For example:
  ```
  DEFINED SYMBOLS
  foo.s:229    .text:00000000 _main
  ```
  The undefined symbols will have the format:
  ```
  UNDEFINED SYMBOLS
  <symbol>
  ```
  For example:
  ```
  UNDEFINED SYMBOLS
  printf
  ```

EXAMPLE 1-2: SAMPLE ASSEMBLER LISTING FILE

```
MPLAB ASM30 Listing:  example1.1.s                      page 1
 Sample dsPIC Assembler Source Code
 For illustration only.
   1
   2                            .title " Sample dsPIC Assembler Source Code"
   3                            .sbttl " For illustration only."
   4
   5                            ; dsPIC registers
   6                            .equ CORCONL, CORCON
   7                            .equ PSV,2
   8
   9                            .section .const,psv
  10                     hello:
  11 0000 48 65 6C 6C           .ascii "Hello world!\n\0"
  11      6F 20 77 6F
  11      72 6C 64 21
  11      0A 00
  12
  13                            .text
  14                            .global __reset
  15                     __reset:
  16                            ; set PSVPAG to page that contains 'hello'
  17 000000  00 00 20           mov     #psvpage(hello),w0
  18 000002  00 00 88           mov     w0,PSVPAG
  19
  20                            ; enable Program Space Visibility
  21 000004  00 40 A8           bset.b  CORCONL,#PSV
  22
  23                            ; make a pointer to 'hello'
  24 000006  00 00 20           mov     #psvoffset(hello),w0
  25
  26                            .end


MPLAB ASM30 Listing:  example1.1.s                      page 2
 Sample dsPIC Assembler Source Code
 For illustration only.
DEFINED SYMBOLS
                            *ABS*:00000000 fake
        example1.1.s:10     .const:00000000 hello
        example1.1.s:15     .text:00000000 __reset
                            .text:00000000 .text
                            .data:00000000 .data
                             .bss:00000000 .bss
                           .const:00000000 .const

UNDEFINED SYMBOLS
CORCON
PSVPAG
```


© 2008 Microchip Technology Inc.

Chapter 2. Assembler Command Line Interface

2.1 INTRODUCTION

MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB ASM30) may be used on the command line interface as well as with MPLAB IDE. For information on using the assembler with MPLAB IDE, please refer to "*16-bit Language Tools Getting Started"* (DS70094).

2.2 HIGHLIGHTS

Topics covered in this chapter are:

- Syntax
- Options that Modify the Listing Output
- Options that Control Informational Output
- Options that Control Output File Creation
- Other Options

2.3 SYNTAX

The assembler command line may contain options and file names. Options may appear in any order and may be before, after or between file names. The order of file names determines the order of assembly.

```
pic30-as [options|sourcefiles]...
```

'`--`' (two hyphens) by itself names the standard input file explicitly, as one of the files for the assembler to translate. Except for '`--`', any command line argument that begins with a hyphen ('`-`') is an option. Each option changes the behavior of the assembler, but no option changes the way another option works.

Some options require exactly one file name to follow them. The file name may either immediately follow the option's letter or it may be the next command line argument. For example, to specify an output file named `test.o`, either of the following options would be acceptable:

- `-o test.o`
- `-otest.o`

Note:	Command line options are case sensitive.

2.4 OPTIONS THAT MODIFY THE LISTING OUTPUT

The following options are used to control the listing output. For debugging and general analysis of code operation, a listing file is helpful. Constructing one with useful information is accomplished using the options in this section.

- `-a[suboption] [=file]`
- `--listing-lhs-width #`
- `--listing-lhs-width2 #`
- `--listing-rhs-width #`
- `--listing-cont-lines #`

2.4.1 -a[*suboption*] [=*file*]

The `-a` option enables listing output. The `-a` option supports the following sub options to further control what is included in the assembly listing:

`-ac`	Omit false conditionals
`-ad`	Omit debugging directives
`-ah`	Include high-level source
`-ai`	Include section information
`-al`	Include assembly
`-am`	Include macro expansions
`-an`	Omit forms processing
`-as`	Include symbols
`-a=file`	Output listing to specified file (must be in current directory).

If no sub-options are specified, the default sub-options used are `hls`; the `-a` option by itself requests high-level, assembly, and symbolic listing. You can use other letters to select specific options for the listing output.

The letters after the `-a` may be combined into one option. So for example instead of specifying `-al -an` on the command line, you could specify `-aln`.

© 2008 Microchip Technology Inc.

2.4.1.1 -ac

-ac omits false conditionals from a listing. Any lines that are not assembled because of a false .if or .ifdef (or the .else of a true .if or .ifdef) will be omitted from the listing. Example 2-1 shows a listing where the -ac option was not used. Example 2-2 shows a listing for the same source where the -ac option was used.

EXAMPLE 2-1: LISTING FILE GENERATED WITH -al COMMAND LINE OPTION

```
MPLAB ASM30 Listing:  example2.1.s           page 1


  1                             .data
  2                             .if 0
  3                               .if 1
  4                               .endif
  5                               .long 0
  6                               .if 0
  7                                 .long 0
  8                               .endif
  9                             .else
 10                               .if 1
 11                               .endif
 12 0000 02 00 00 00              .long 2
 13                               .if 0
 14                                 .long 3
 15                               .else
 16 0004 04 00 00 00                .long 4
 17                               .endif
 18                            .endif
 19
 20                            .if 0
 21                              .long 5
 22                            .elseif 1
 23                              .if 0
 24                                .long 6
 25                              .elseif 1
 26 0008 07 00 00 00                .long 7
 27                              .endif
 28                            .elseif 1
 29                              .long 8
 30                            .else
 31                              .long 9
 32                            .endif
```

EXAMPLE 2-2: LISTING FILE GENERATED WITH -alc COMMAND LINE OPTION

```
MPLAB ASM30 Listing:  example2.2.s           page 1

   1                            .data
   2                            .if 0
   9                            .else
  10                              .if 1
  11                              .endif
  12 0000 02 00 00 00             .long 2
  13                              .if 0
  15                              .else
  16 0004 04 00 00 00               .long 4
  17                              .endif
  18                           .endif
  19
  20                           .if 0
  22                           .elseif 1
  23                              .if 0
  25                              .elseif 1
  26 0008 07 00 00 00               .long 7
  27                              .endif
  28                           .elseif 1
  30                           .else
  32                           .endif
```

Note: Some lines omitted due to `-ac` option, i.e., lines 3-8, 14, 21, 24, 29 and 31.

© 2008 Microchip Technology Inc.

2.4.1.2 -ad

-ad omits debugging directives from the listing. This is useful if a compiler that was given a debugging option generated the assembly source code. The compiler-generated debugging directives will not clutter the listing. Example 2-3 shows a listing using both the d and h sub-options. Compared to using the h sub-option alone (see next section), the listing is much cleaner.

EXAMPLE 2-3: LISTING FILE GENERATED WITH -alhd COMMAND LINE OPTION

```
MPLAB ASM30 Listing:  example2.3.s            page 1


   1                            .file "example2.3.c"
   2                            .text
   3                            .align  2
   9                            .global _main  ; export
  10                     _main:
   1:example2.3.c **** extern int ADD (int, int);
   2:example2.3.c ****
   3:example2.3.c **** int
   4:example2.3.c **** main(void)
   5:example2.3.c **** {
  16                            .set    __PA__,1
  17 000000  00 00 FA           lnk     #0
  18
   6:example2.3.c ****   return ADD(4, 5);
  20 000002  51 00 20           mov     #5,w1
  21 000004  40 00 20           mov     #4,w0
  22 000006  00 00 02           call    _ADD
  22         00 00 00
   7:example2.3.c **** }
  29
  30 00000a  00 80 FA           ulnk
  31 00000c  00 00 06           return
  32                            .set    __PA__,0
  37
  38                            .end
```

2.4.1.3 -ah

-ah requests a high-level language listing. High-level listings require that the assembly source code was generated by a compiler, a debugging option like -g was given to the compiler, and that assembly listings (-al) also be requested. -al requests an output program assembly listing. Example 2-4 shows a listing that was generated using the -alh command line option.

EXAMPLE 2-4: LISTING FILE GENERATED WITH -alh COMMAND LINE OPTION

```
MPLAB ASM30 Listing:  example2.4.s            page 1


   1                            .file "example2.4.c"
   2                            .text
   3                            .align  2
   4                            .def    _main
   5                            .val    _main
   6                            .scl    2
   7                            .type   044
   8                            .endef
   9                            .global _main  ; export
  10                     _main:
  11                            .def    .bf
  12                            .val    .
  13                            .scl    101
   1:example2.4.c **** extern int ADD (int, int);
   2:example2.4.c ****
   3:example2.4.c **** int
   4:example2.4.c **** main(void)
   5:example2.4.c **** {
  14                            .line   5
  15                            .endef
  16                            .set    __PA__,1
  17 000000  00 00 FA           lnk     #0
  18
   6:example2.4.c ****   return ADD(4, 5);
  19                            .ln     6
  20 000002  51 00 20           mov     #5,w1
  21 000004  40 00 20           mov     #4,w0
  22 000006  00 00 02           call    _ADD
  22         00 00 00
   7:example2.4.c **** }
  23                            .ln     7
  24                            .def    .ef
  25                            .val    .
  26                            .scl    101
  27                            .line   7
  28                            .endef
  29
  30 00000a  00 80 FA           ulnk
  31 00000c  00 00 06           return
  32                            .set    __PA__,0
  33                            .def    _main
  34                            .val    .
  35                            .scl    -1
  36                            .endef
  37
  38                            .end
```


© 2008 Microchip Technology Inc.

2.4.1.4 -ai

-ai displays information on each of the code and data sections. This information contains details on the size of each of the sections and then a total usage of program and data memory. Example 2-5 shows a listing where the -ai option was used.

EXAMPLE 2-5: LISTING FILE GENERATED WITH -ai COMMAND LINE OPTION

```
SECTION INFORMATION:

Section        Length (PC units)   Length (bytes) (dec)
-------        -----------------   --------------------
.text                       0x16             0x21  (33)

TOTAL PROGRAM MEMORY USED (bytes):           0x21  (33)

Section                             Length (bytes) (dec)
-------                             --------------------
.data                                              0  (0)
.bss                                               0  (0)

   TOTAL DATA MEMORY USED (bytes):                 0  (0)
```

2.4.1.5 -al

-al requests an assembly listing. This sub-option may be used with other sub-options. See the other examples in this section.

2.4.1.6 -am

-am expands macros in a listing. Example 2-6 shows a listing where the -am option was not used. Example 2-7 shows a listing for the same source where the -am option was used.

EXAMPLE 2-6: LISTING FILE GENERATED WITH -al COMMAND LINE OPTION

```
MPLAB ASM30 Listing:  example2.5.s            page 1


   1                            .text
   2                            .macro div_s reg1, reg2
   3                              repeat #18-1
   4                              div.sw \reg1,\reg2
   5                            .endm
   6
   7                            .macro div_u reg1, reg2
   8                              repeat #18-1
   9                              div.uw \reg1,\reg2
  10                            .endm
  11
  12 000000  40 01 20           mov #20, w0
  13 000002  52 00 20           mov #5, w2
  14 000004  11 00 09           div_u w0, w2
  14         02 80 D8
  15
  16 000008  00 02 BE           mov.d w0, w4
  17
  18 00000a  40 01 20           mov #20, w0
  19 00000c  B3 FF 2F           mov #-5, w3
  20 00000e  11 00 09           div_s w0, w3
  20         03 00 D8
```

EXAMPLE 2-7: LISTING FILE GENERATED WITH -alm COMMAND LINE OPTION

```
MPLAB ASM30 Listing:  example2.6.s            page 1


   1                            .text
   2                            .macro div_s reg1, reg2
   3                              repeat #18-1
   4                              div.sw \reg1,\reg2
   5                            .endm
   6
   7                            .macro div_u reg1, reg2
   8                              repeat #18-1
   9                              div.uw \reg1,\reg2
  10                            .endm
  11
  12 000000  40 01 20           mov #20, w0
  13 000002  52 00 20           mov #5, w2
  14                            div_u w0, w2
  14 000004  11 00 09   > repeat #18-1
  14 000006  02 80 D8   > div.uw w0,w2
  15
  16 000008  00 02 BE           mov.d w0, w4
  17
  18 00000a  40 01 20           mov #20, w0
  19 00000c  B3 FF 2F           mov #-5, w3
  20                            div_s w0, w3
  20 00000e  11 00 09   > repeat #18-1
  20 000010  03 00 D8   > div.sw w0,w3
```

Note: > signifies expanded macro instructions.

© 2008 Microchip Technology Inc.

2.4.1.7 -an

-an turns off all forms processing that would be performed by the listing directives .psize, .eject, .title and .sbttl. Example 2-8 shows a listing where the -an option was not used. Example 2-9 shows a listing for the same source where the -an option was used.

EXAMPLE 2-8: LISTING FILE GENERATED WITH -al COMMAND LINE OPTION

```
MPLAB ASM30 Listing:  example2.7.s            page 1
User's Guide Example
  Listing Options
   1                            .text
   2                        .title "User's Guide Example"
   3                        .sbttl "  Listing Options"
   4                              .psize 10
   5
   6 000000  50 00 20             mov #5, w0
   7 000002  61 00 20             mov #6, w1
MPLAB ASM30 Listing:  example2.7.s            page 2
User's Guide Example
  Listing Options
   8 000004  01 01 40             add w0, w1, w2
   9                              .eject
MPLAB ASM30 Listing:  example2.7.s            page 3
User's Guide Example
  Listing Options
  10
  11 000006  24 00 20             mov #2, w4
  12 000008  03 00 09             repeat #3
  13 00000a  04 22 B8             mul.uu w4, w4, w4
  14
  15 00000c  16 00 20             mov #1, w6
  16 00000e  64 33 DD             sl w6, #4, w6
MPLAB ASM30 Listing:  example2.7.s            page 4
User's Guide Example
  Listing Options
  17
  18 000010  06 20 E1             cp w4, w6
  19 000012  00 00 32             bra z, done
  20
  21 000014  00 00 00             nop
  22
  23                      done:
MPLAB ASM30 Listing:  example2.7.s            page 5
User's Guide Example
  Listing Options
  24
  25                              .end
```

EXAMPLE 2-9: LISTING FILE GENERATED WITH -aln COMMAND LINE OPTION

```
  1                             .text
  2                          .title "User's Guide Example"
  3                          .sbttl "  Listing Options"
  4                             .psize 10
  5
  6 000000  50 00 20            mov #5, w0
  7 000002  61 00 20            mov #6, w1
  8 000004  01 01 40            add w0, w1, w2
  9                             .eject
 10
 11 000006  24 00 20            mov #2, w4
 12 000008  03 00 09            repeat #3
 13 00000a  04 22 B8            mul.uu w4, w4, w4
 14
 15 00000c  16 00 20            mov #1, w6
 16 00000e  64 33 DD            sl w6, #4, w6
 17
 18 000010  06 20 E1            cp w4, w6
 19 000012  00 00 32            bra z, done
 20
 21 000014  00 00 00            nop
 22
 23                       done:
 24
 25                             .end
```

2.4.1.8 `-as`

`-as` requests a symbol table listing. Example 2-10 shows a listing that was generated using the `-as` command line option. Note that both defined and undefined symbols are listed.

EXAMPLE 2-10: LISTING FILE GENERATED WITH -as COMMAND LINE OPTION

```
MPLAB ASM30 Listing:  sample2b.s

DEFINED SYMBOLS
                            *ABS*:00000000 fake
          sample2b.s:4      .text:00000000 __reset
          sample2b.s:13     .text:0000001c L2
                            .text:00000000 .text
                            .data:00000000 .data
                             .bss:00000000 .bss

UNDEFINED SYMBOLS
_i
_j
```

2.4.1.9 `-a=`*`file`*

`=file` defines the name of the output file. This file must be in the current directory.

© 2008 Microchip Technology Inc.

2.4.2 --listing-lhs-width

The --listing-lhs-width option is used to set the width of the output data column of the listing file. By default, this is set to 3 for program memory and 4 for data memory. The following line is extracted from a listing. The output data column is in bold text.

```
6 000000   50 00 20          mov #5, w0
```

If the option --listing-lhs-width 2 is used, then the same line will appear as follows in the listing:

```
6 000000   50 00            mov #5, w0
6          20
```

2.4.3 --listing-lhs-width2

The --listing-lhs-width2 option is used to set the width of the continuation lines of the output data column of the listing file. By default, this is set to 3 for program memory and 4 for data memory. If the specified width is smaller than the first line, this option is ignored. The following lines are extracted from a listing. The output data column is in bold.

```
2 0000 50 6C 65 61   .ascii "Please pay inside."
2      73 65 20 70
2      61 79 20 69
2      6E 73 69 64
2      65 2E
```

If the option --listing-lhs-width2 7 is used, then the same line will appear as follows in the listing:

```
2 0000 50 6C 65 61   .ascii "Please pay inside."
2      73 65 20 70 61 79 20
2      69 6E 73 69 64 65 2E
```

2.4.4 --listing-rhs-width

The --listing-rhs-width option is used to set the maximum width in characters of the lines from the source file. By default, this is set to 100. The following lines are extracted from a listing that was created without using the --listing-rhs-width option. The text in bold are the lines from the source file.

```
2 0000 54 68 69 73 .ascii "This line is long."
2      20 6C 69 6E
2      65 20 69 73
2      20 6C 6F 6E
2      67 65 72 20
```

If the option --listing-rhs-width 20 is used, then the same line will appear as follows in the listing:

```
2 0000 54 68 69 73 .ascii "This line i
2      20 6C 69 6E
2      65 20 69 73
2      20 6C 6F 6E
2      67 65 72 20
```

The line is truncated (not wrapped) in the listing, but the data is still there.

2.4.5 --listing-cont-lines

The --listing-cont-lines option is used to set the maximum number of continuation lines used for the output data column of the listing. By default, this is 8. The following lines are extracted from a listing that was created without using the--listing-cont-lines option. The text in bold shows the continuation lines used for the output data column of the listing.

```
2 0000 54  68  69  73     .ascii "This is a long character sequence."
2      20  69  73  20
2      61  20  6C  6F
2      6E  67  20  63
2      68  61  72  61
2      63  74  65  72
2      20  73  65  71
2      75  65  6E  63
2      65  2E
```

Notice that the number of bytes displayed matches the number of bytes in the ASCII string; however, if the option --listing-cont-lines 2 is used, then the output data will be truncated after 2 continuation lines as shown below.

```
2 0000 54  68  69  73     .ascii "This is a long character sequence."
2      20  69  73  20
2      61  20  6C  6F
```

2.5 OPTIONS THAT CONTROL INFORMATIONAL OUTPUT

The options in this section control how information is output. Errors, warnings and messages concerning code translation and execution are controlled through several of the options in this section.

Any item in parenthesis shows the short method of specifying the option, e.g., --no-warn also may be specified as -W.

2.5.1 --fatal-warnings

Warnings are treated as if they were errors.

2.5.2 --no-warn (-W)

Warnings are suppressed. If you use this option, no warnings are issued. This option only affects the warning messages. It does not change how your file is assembled. Errors are still reported.

2.5.3 --warn

Warnings are issued, if appropriate. This is the default behavior.

2.5.4 -J

No warnings are issued about signed overflow.

2.5.5 --help

The assembler will show a message regarding the command line usage and options. The assembler then exits.

2.5.6 --target-help

The assembler will show a message regarding the 16-bit device specific command line options. The assembler then exits.

© 2008 Microchip Technology Inc.

2.5.7 --version

The assembler version number is displayed. The assembler then exits.

2.5.8 --verbose (-v)

The assembler version number is displayed. The assembler does not exit. If this is the only command line option used, then the assembler will print out the version and wait for entry of the assembly source through standard input. Use <CTRL>-D to send an EOF character to end assembly.

2.6 OPTIONS THAT CONTROL OUTPUT FILE CREATION

The options in this section control how the output file is created. For example, to change the name of the output object file, use -o.

Any item in parenthesis shows the short method of specifying the option, e.g., --keep-locals may be specified as -L also.

2.6.1 -g

Generate symbolic debugging information.

2.6.2 --keep-locals (-L)

Keep local symbols, i.e., labels beginning with .L (upper case only). Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs. Normally both the assembler and linker discard such symbols. This option tells the assembler to retain those symbols in the object files.

2.6.3 -o *objfile*

Name the object file output *objfile*. In the absence of errors, there is always one object file output when you run the assembler. By default, it has the name a.out. Use this option (which takes exactly one filename) to give the object file a different name. Whatever the object file is called, the assembler overwrites any existing file with the same name.

2.6.4 -omf = *format*

Use this option to specify the object file format. Valid format names are COFF and ELF. Object file format names are not case sensitive.

2.6.5 -R

This option tells the assembler to write the object file as if all data-section data lives in the text section. The data section part of your object file is zero bytes long because all its bytes are located in the text section.

2.6.6 --relax

Turn relaxation on. Convert absolute calls and gotos to relative calls and branches when possible.

2.6.7 --no-relax

Turn relaxation off. This is the default behavior.

2.6.8 -Z

Generate object file even after errors. After an error message, the assembler normally produces no output. If for some reason, you are interested in object file output even after the assembler gives an error message, use the `-Z` option. If there are any errors, the assembler continues anyway, and writes an object file after a final warning message of the form "n errors, m warnings, generating bad object file".

2.6.9 -MD *file*

Write dependency information to *file*. The assembler can generate a dependency file. This file consists of a single rule suitable for describing the dependencies of the main source file. The rule is written to the file named in its argument. This feature can be used in the automatic updating of makefiles.

2.7 OTHER OPTIONS

The options in this section perform functions not defined in previous sections.

2.7.1 --defsym *sym=value*

Define symbol *sym* to given *value*.

2.7.2 -I *dir*

Use this option to add *dir* to the list of directories that the assembler searches for files specified in `.include` directives. You may use `-I` as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, the assembler searches any `-I` directories in the same order as they were specified (left to right) on the command line.

2.7.3 -p, --processor=PROC

Specify the target processor, e.g.:

```
pic30-as -p30F2010 file.s
```

The assembler defines macros based on the target processor setting, which can be tested by conditional directives in source code. For example, include file `p30f2010.inc` contains the following:

```
.ifndef __30F2010
  .error "Include file does not match processor setting"
.endif
```

In addition to the target processor, a macro to identify the device family is also defined. For example:

```
.ifdef __dsPIC30F
  .print "dsPIC30F family selected"
.endif
```

Macros for the following device families are defined based on target processor setting:

Macro	Support
`__dsPIC30F`	dsPIC30Fxxxx device
`__dsPIC33F`	dsPIC33Fxxxx device
`__PIC24F`	PIC24Fxxxx device
`__PIC24H`	PIC24Hxxxx device
`__MCHP16`	Generic 16-bit device

© 2008 Microchip Technology Inc.

Chapter 3. Assembler Syntax

3.1 INTRODUCTION

Syntax for MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB ASM30) source code is defined here.

3.2 HIGHLIGHTS

Topics covered in this chapter are:

- Internal Preprocessor
- Source Code Format
- Constants
- Summary

3.3 INTERNAL PREPROCESSOR

The assembler has an internal preprocessor. The internal processor:

1. Adjusts and removes extra white space. It leaves one space or tab before the keywords on a line, and turns any other white space on the line into a single space.
2. Removes all comments, replacing them with a single space, or an appropriate number of new lines.
3. Converts character constants into the appropriate numeric value.

> **Note:** If you have a single character (e.g., 'b') in your source code, this will be changed to the appropriate numeric value. If you have a syntax error that occurs at the single character, the assembler will not display 'b', but instead display the first digit of the decimal equivalent.

For example, if you had `.global mybuf`, 'b' in your source code, the error message would say "Error: Rest of line ignored. First ignored character is '9'." Notice the error message says '9'. This is because the 'b' was converted to its decimal equivalent 98. The assembler is actually parsing `.global mybuf,98`

The internal processor does **not** do:

1. macro preprocessing
2. include file handling
3. anything else you may get from your C compiler's preprocessor

You can do include file preprocessing with the `.include` directive (See **Chapter 6. "Assembler Directives"**.) You can use the C compiler driver to get other C preprocessing style preprocessing by giving the input file a `.S` suffix (See the "*MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User's Guide"* (DS51284) for more information.)

If the first line of an input file is `#NO_APP` or if you use the `-f` option, white space and comments are not removed from the input file. Within an input file, you can ask for white space and comment removal in certain portions by putting a line that says `#APP` before the text that may contain white space or comments, and putting a line that says `#NO_APP` after this text. This feature is mainly intended to support assembly statements in compilers whose output is otherwise free of comments and white space.

Note: Excess white space, comments and character constants cannot be used in the portions of the input text that are not preprocessed.

3.4 SOURCE CODE FORMAT

Assembly source code consists of statements and white spaces.

White space is one or more spaces or tabs. White space is used to separate pieces of a source line. White space should be used to make your code easier for people to read. Unless within character constants, any white space means the same as exactly one space.

Each *statement* has the following general format and is followed by a new line.

```
[label:]  [mnemonic   [operands] ]     [; comment]
OR
[label:]  [directive  [arguments] ]    [; comment]
```

- Label
- Mnemonic
- Directive
- Operands
- Arguments
- Comments

3.4.1 Label

A label is one or more characters chosen from the set of all letters, digits and the two characters underline (_) and period (.). Labels may not begin with a decimal digit, except for the special case of a local symbol. (See **Section 5.5 "Local Symbols"** for more information.) Case is significant. There is no length limit; all characters are significant.

Label definitions must be immediately followed by a colon. A space, tab, end of line or an assembler mnemonic or directive may follow the colon.

Label definitions may appear on a line by themselves and will reference the next address.

The value of a label after linking is the absolute address of a location in memory.

3.4.2 Mnemonic

Mnemonics tell the assembler what machine instructions to assemble. For example, addition (`ADD`), branches (`BRA`) or moves (`MOV`). Unlike labels that you create yourself, mnemonics are provided by the assembly language. Mnemonics are not case sensitive.

See the "*dsPIC30F/33F Programmer's Reference Manual*" (DS70157) for more details.

© 2008 Microchip Technology Inc.

3.4.3 Directive

Assembler directives are commands that appear in the source code but are not translated directly into machine code. Directives are used to control the assembler; its input, output and data allocation. The first character of a directive is a period (.). More details are provided in **Chapter 6. "Assembler Directives"** on the available directives.

3.4.4 Operands

Each machine instruction takes from 0 up to 8 operands. (See the *"dsPIC30F/33F Programmer's Reference Manual"* (DS70157). These operands give information to the instruction on the data that should be used and the storage location for the instruction. Operands must be separated from mnemonics by one or more spaces or tabs.

Commas should separate multiple operands. If commas do not separate operands, a warning will be displayed and the assembler will take its best guess on the separation of the operands. Operands consist of literals, file registers condition codes, destination select and accumulator select.

3.4.4.1 LITERALS

Literal values are distinguished with a preceding pound sign ('#'). Literal values can be hexadecimal, octal, binary or decimal format. Hexadecimal numbers are distinguished by a leading `0x`. Octal numbers are distinguished by a leading `0`. Binary numbers are distinguished by a leading `B`. Decimal numbers require no special leading or trailing character.

Examples:

`#0xe`, `#016`, `#0b1110` and `#14` all represents the literal value 14.

`#-5` represents the literal value -5.

`#symbol` represents the value of `symbol`.

3.4.4.2 FILE REGISTERS

File registers represent on-chip general purpose and special function registers. File registers are distinguished from literal values because they do not have the preceding pound sign.

Each of the following examples tells the processor to move the data located in the file register whose address is 14 to `w0`:

```
mov 0xE, w0
mov 016, w0
mov 14, w0
.equ symbol, 14
mov symbol, w0
```

3.4.4.3 REGISTERS

The following register names are built into the assembler:

w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14, w15, W0, W1, W2, W3, W4, W5, W6, W7, W8, W9, W10, W11, W12, W13, W14, W15.

3.4.4.4 CONDITION CODES

Condition codes are used with `BRA` instructions. See the *"dsPIC30F/33F Programmer's Reference Manual" (DS70157)* for more details.

```
bra C, label
```

3.4.4.5 DESTINATION SELECT

The PIC18CXXX-compatible instructions accept `WREG` as an optional argument to specify whether the result should be placed into `WREG` (W0) or into the file register. See the *"dsPIC30F/33F Programmer's Reference Manual" (DS70157)* for more details.

```
add sym, WREG
```

3.4.4.6 ACCUMULATOR SELECT

The DSP instructions take an accumulator select operand (A or B) to specify which accumulator to use.

```
ADD A
```

3.4.5 Arguments

Each directive takes from 0 up to 3 arguments. These arguments give additional information to the directive on how it should carry out the command. Arguments must be separated from directives by one or more spaces or tabs. Commas must separate multiple arguments. More details are provided in **Chapter 6. "Assembler Directives"** on the available directives.

3.4.6 Comments

Comments can be represented in the assembler in one of two ways described below.

3.4.6.1 SINGLE LINE COMMENT

This type of comment extends from the comment character to the end of the line. For a single line comment, use a semicolon (';').

Example:

```
mov w0, w1;The rest of this line is a comment.
```

3.4.6.2 MULTILINE COMMENT

This type of comment can span multiple lines. For a multi-line comment, use

/* ... */. These comments cannot be nested.

Example:

```
/* All
of these
lines
are
comments */
```


© 2008 Microchip Technology Inc.

3.5 CONSTANTS

A constant is a value written so that its value is known by inspection, without knowing any context. Examples are:

```
.byte 74, 0112, 0b01001010, 0x4A, 0x4a, 'J', '\J';All the same value
.ascii "Ring the bell\7";A string constant
.float 0f-3141592653589793238462643383279502884l971.693993751E-40
```

- Numeric Constants
- Character Constants

3.5.1 Numeric Constants

The assembler distinguishes three kinds of numbers according to how they are stored in the machine. Integers are numbers that would fit into a `long` in the C language. Floating-point numbers are IEEE 754 floating-point numbers. Fixed-point numbers are Q-15 fixed-point format.

3.5.1.1 INTEGERS

A binary integer is '0b' or '0B' followed by zero or more of the binary digits '01'.

An octal integer is '0' followed by zero or more of the octal digits '01234567'.

A decimal integer starts with a non-zero digit followed by zero or more decimal digits '0123456789'.

A hexadecimal integer is '0x' or '0X' followed by one or more hexadecimal digits '0123456789abcdefABCDEF'.

To denote a negative integer, use the prefix operator '-'.

3.5.1.2 FLOATING-POINT NUMBERS

A floating-point number is represented in IEEE 754 format. A floating-point number is written by writing (in order):

- An optional prefix, which consists of the digit '0', followed by the letter 'e', 'f' or 'd' in upper or lower case. Because floating point constants are used only with `.float` and `.double` directives, the precision of the binary representation is independent of the prefix.
- An optional sign: either '+' or '-'.
- An optional integer part: zero or more decimal digits.
- An optional fractional part: '.' followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - An 'E' or 'e'.
  - Optional sign: either '+' or '-'.
  - One or more decimal digits.

At least one of the integer part or fractional part must be present. The floating-point number has the usual base-10 value.

Floating-point numbers are computed independently of any floating-point hardware in the computer running the assembler.

3.5.1.3 FIXED-POINT NUMBERS

A fixed-point number is represented in Q-15 format. This means that 15 bits are used to represent the fractional portion of the number. The most significant bit is the sign bit, followed by an implied binary point, and 15 bits of magnitude, i.e.:

bit no.	15	.	14	13	12	...	1	0
value	$\pm 2^0$	.	2^{-1}	2^{-2}	2^{-3}	...	2^{-14}	2^{-15}

The smallest number in this format is -1, represented by:

```
0x8000 (1.000 0000 0000 0000)
```

the largest number is nearly 1 (.99996948), represented by:

```
0x7FFF (0.111 1111 1111 1111)
```

A fixed-point number is written in the same format as a floating-point number, but its value is constrained to be in the range [-1.0, 1.0).

3.5.2 Character Constants

There are two kinds of character constants. A *character* stands for one character in one byte and its value may be used in numeric expressions. A *string* can contain potentially many bytes and their values may not be used in arithmetic expressions.

3.5.2.1 CHARACTERS

A single character may be written as a single quote immediately followed by that character, or as a single quote immediately followed by that character and another single quote. The assembler accepts the following escape characters to represent special control characters:

TABLE 3-1: ESCAPE CHARACTERS

Escape Character	Description	Hex Value
`\a`	Bell (alert) character	07
`\b`	Backspace character	08
`\f`	Form-feed character	0C
`\n`	New-line character	0A
`\r`	Carriage return character	0D
`\t`	Horizontal tab character	09
`\v`	Vertical tab character	0B
`\\`	Backslash	5C
`\?`	Question mark character	3F
`\"`	Double quote character	22
`\digit digit digit`	Octal character code. The numeric code is 3 octal digits.	
`\x hex-digits`	Hex character code. All trailing hex digits are combined. Either upper or lower case x works.	

The value of a character constant in a numeric expression is the machine's byte-wide code for that character. The assembler assumes your character code is ASCII.

3.5.2.2 STRINGS

A string is written between double quotes. It may contain double quotes or null characters. The way to get special characters into a string is to escape the characters, preceding them with a backslash '\' character. The same escape sequences that apply to strings also apply to characters.

© 2008 Microchip Technology Inc.

3.6 SUMMARY

Table 3-2 summarizes the general syntax rules that apply to the assembler:

TABLE 3-2: SYNTAX RULES

Character	Character Description	Syntax Usage
.	period	begins a directive
;	semicolon	begin single-line comment
/*	slash, asterisk	begin multiple-line comment
*/	asterisk, slash	end multiple-line comment
:	colon	end a label definition
#	pound	begin a literal value
'c'	character in single quotes	specifies single character value
"string"	character string in double quotes	specifies a character string

NOTES:

© 2008 Microchip Technology Inc.

Chapter 4. Assembler Expression Syntax and Operation

4.1 INTRODUCTION

Expression syntax and operation for MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB ASM30) is discussed here.

4.2 HIGHLIGHTS

Topics covered in this chapter are:

- Expressions
- Operators
- Special Operators

4.3 EXPRESSIONS

An expression specifies an address or numeric value. White space may precede and/or follow an expression. The result of an expression must be an absolute number, or else an offset into a particular section. If an expression is not absolute, and there is not enough information when the assembler sees the expression to know its section, the assembler terminates with an error message in this situation.

4.3.1 Empty Expressions

An empty expression has no value: it is just white space or null. Wherever an absolute expression is required, you may omit the expression, and the assembler assumes a value of (absolute) 0.

4.3.2 Integer Expressions

An integer expression is one or more arguments delimited by operators. Arguments are symbols, numbers or sub expressions. Sub expressions are a left parenthesis '(' followed by an integer expression, followed by a right parenthesis ')'; or a prefix operator followed by an argument.

Integer expressions involving symbols in program memory are evaluated in Program Counter (PC) units. On the 16-bit device, the Program Counter increments by 2 for each instruction word. For example, to branch to the next instruction after label `L`, specify `L+2` as the destination.

Example:

```
bra L+2
```

4.4 OPERATORS

Operators are arithmetic functions, like + or %. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and/or followed by white space.

Prefix operators have higher precedence than infix operators. Infix operators have an order of precedence dependent on their type.

4.4.1 Prefix Operators

The assembler has the following prefix operators. Each takes one argument, which must be absolute.

TABLE 4-1: PREFIX OPERATORS

Operator	Description	Example
-	Negation. Two's complement negation.	`-1`
~	Bit-wise not. One's complement.	`~flags`

4.4.2 Infix Operators

Infix operators take two arguments, one on either side. Operators have a precedence, by type, as shown in the table below, but operations with equal precedence are performed left to right. Apart from + or –, both operators must be absolute, and the result is absolute.

TABLE 4-2: INFIX OPERATORS

Operator	Description	Example
Arithmetic		
*	Multiplication	`5 * 4 (=20)`
/	Division. Truncation is the same as the C operator '/'.	`23 / 4 (=5)`
%	Remainder	`30 % 4 (=2)`
<<	Shift Left. Same as the C operator '<<'	`2 << 1 (=4)`
>>	Shift Right. Same as the C operator '>>'	`2 >> 1 (=1)`
Bit-Wise		
&	Bit-wise And	`4 & 6 (=4)`
^	Bit-wise Exclusive Or	`4 ^ 6 (=2)`
!	Bit-wise Or Not	`0x1010 ! 0x5050 (=0xBFBF)`
|	Bit-wise Inclusive Or	`2 | 4 (=6)`
Simple Arithmetic		
+	Addition. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.	`4 + 10 (=14)`
-	Subtraction. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.	`14 - 4 (=10)`
Relational		
==	Equal to	`.if (x == y)`
!=	Not equal to (also <>)	`.if (x != y)`
<	Less than	`.if (x < 5)`
<=	Less than or equal to	`.if (y <= 0)`
>	Greater than	`.if (x > a)`

© 2008 Microchip Technology Inc.

TABLE 4-2: INFIX OPERATORS (CONTINUED)

Operator	Description	Example
>=	Greater than or equal to	.if (x >= b)
Logical		
&&	Logical AND	.if ((x > 1) && (x < 10))
||	Logical OR	.if ((y != x) || (y < 100))

4.5 SPECIAL OPERATORS

The assembler provides a set of special operators for the following:

- Accessing Data in Program Memory
- Obtaining a Program Address of a Symbol or Constant
- Obtaining a Handle to a Program Address
- Obtaining the DMA Offset of a Symbol - PIC24H/dsPIC33F Devices Only
- Obtaining the Size of a Specific Section
- Obtaining the Starting Address of a Specific Section
- Accessing Functions in Boot or Secure Segments

TABLE 4-3: SPECIAL OPERATORS

Operators	Description	Support
tblpage(*name*)	Get page for table read/write operations	All
tbloffset(*name*)	Get pointer for table read/write operations	All
psvpage(*name*)	Get page for PSV data window operations	All
psvoffset(*name*)	Get pointer for PSV data window operations	All
paddr(*label*)	Get 24-bit address of *label* in program memory	All
handle(*label*)	Get 16-bit reference to *label* in program memory	All
dmaoffset(*name*)	Get the offset of a symbol within DMA memory	24H/33
.sizeof.(*name*)	Get size of section *name* in address units	All
.startof.(*name*)	Get starting address of section *name*	All
boot(num)	Get address of access slot num in the boot segment.	All
secure(num)	Get address of access slot num in the secure segment.	All

Legend: All = Support for all devices
24H = Support for PIC24H MCUs
33 = Support for dsPIC33F DSCs

4.5.1 Accessing Data in Program Memory

The 16-bit device modified-Harvard architecture is comprised of two separate address spaces: one for data storage and one for program storage. Data memory is 16 bits wide and is accessed with a 16-bit address; program memory is 24 bits wide and is accessed with a 24-bit address.

Normally, 16-bit instructions can read or write data values only from data memory, while program memory is reserved for instruction storage. This arrangement allows for very fast execution, since the two memory buses can work simultaneously and independently of each other. In other words, a 16-bit instruction can read, modify and write a location in data memory at the same time the next instruction is being fetched from program memory.

Occasionally, circumstances may arise when the programmer or application designer is willing to sacrifice some execution speed in return for the ability to read constant data directly from program memory. For example, certain DSP algorithms require large tables of coefficients that would otherwise consume data memory needed to buffer real-time data. To accommodate these needs, the 16-bit device modified-Harvard architecture permits instructions to access data stored in program memory.

There are two methods available for accessing data in program memory:

- Table Read/Write Instructions
- Program Space Visibility (PSV) Data Window

In either case, the programmer must compensate for the different address width between data memory and program memory. For example, a pointer is commonly used to access constant data tables, yet pointers for table read/write instructions can specify an address of only 16 bits. A pointer used to access the PSV data window can specify only 15 bits – the most significant bit must be set for an address in the data window range (0x8000 to 0xFFFF).

As explained in the *"dsPIC30F/33F Programmer's Reference Manual" (DS70157)*, two special function registers can be used to specify the upper bits of a PSV or table read/write address: `DSPPAG` and `TBLPAG`, respectively.

4.5.1.1 TABLE READ/WRITE INSTRUCTIONS

The `tblpage()` and `tbloffset()` operators provided by the assembler can be used with table read/write instructions. These operators may be applied to any symbol (usually representing a table of constant data) in program memory.

Suppose a table of constant data is declared in program memory like this:

```
      .text
fib_data:
      .word 0, 1, 2, 3, 5, 8, 13
```

To access this table with table read/write instructions, use the `tblpage()` and `tbloffset()` operators as follows:

```
; Set TBLPAG to the page that contains the fib_data array.
   mov    #tblpage(fib_data), w0
   mov    w0, _TBLPAG
; Make a pointer to fib_data for table instructions
   mov    #tbloffset(fib_data), w0
; Load the first data value
   tblrdl [w0++], w1
```

The programmer must ensure that the constant data table does not exceed the program memory page size that is implied by the TBLPAG register. The maximum table size implied by the TBLPAG register is 64 Kbytes. If additional constant data storage is required, simply create additional tables each with its own symbol, and repeat the code sequence above to load the TBLPAG register and derive a pointer.

4.5.1.2 PROGRAM SPACE VISIBILITY (PSV) DATA WINDOW

The `psvpage()` and `psvoffset()` operators can be used with the PSV data window. These operators may be applied to any symbol (usually representing a table of constant data) in program memory.

Suppose a table of constant data is declared in program memory like this:

```
      .section *,psv
fib_data:
      .word 0, 1, 2, 3, 5, 8, 13
```

To access this table through the PSV data window, use the `psvpage()` and `psvoffset()` operators as follows:

© 2008 Microchip Technology Inc.

```
; Enable Program Space Visibility
    bset.b CORCONL, #PSV

; Set PSVPAG to the page that contains the fib_data array.
    mov    #psvpage(fib_data), w0
    mov    w0, _PSVPAG
; Make a pointer to fib_data in the PSV data window
    mov    #psvoffset(fib_data), w0
; Load the first data value
    mov    [w0++], w1
```

The programmer must ensure that the constant data table does not exceed the program memory page size that is implied by the PSVPAG register. The maximum table size implied by the PSVPAG register is 32 Kbytes. If additional constant data storage is required, simply create additional tables each with its own symbol, and repeat the code sequence above to load the PSVPAG register and derive a pointer.

4.5.2 Obtaining a Program Address of a Symbol or Constant

The `paddr()` operator can be used to obtain the program address of a constant or symbol. For example, if you wanted to set up an interrupt vector table without using the default naming conventions, you could use the `paddr()` operator.

```
.section ivt, code
 goto reset
.pword paddr(iv1)
.pword paddr(iv2)
...
```

4.5.3 Obtaining a Handle to a Program Address

The `handle()` operator can be used to obtain the a 16-bit reference to a label in program memory. If the final resolved program counter address of the label fits in 16 bits, that value is returned by the `handle()` operator. If the final resolved address exceeds 16 bits, the address of a jump table entry is returned instead. The jump table entry is a `GOTO` instruction containing a 24-bit absolute address. The handle jump table is created by the linker and is always located in low program memory. Handles permit any location in program memory to be reached via a 16-bit address and are provided to facilitate the use of C function pointers.

The handle jump table is created by the linker and contains an entry for each unique label that is used with the `handle()` operator.

4.5.4 Obtaining the DMA Offset of a Symbol - PIC24H/dsPIC33F Devices Only

The `dmaoffset()` operator can be used to obtain the offset of a symbol within DMA memory. For example, to declare a buffer in DMA memory, and load its offset into a register, you could use:

```
.section *,bss,dma
buf:  .space 256

.text
mov  #dmaoffset(buf), W0
```

To construct a table of DMA offsets for several symbols, you could use:

```
.word  dmaoffset(buf1)
.word  dmaoffset(buf2)
.word  dmaoffset(buf3)
...
```

4.5.5 Obtaining the Size of a Specific Section

The `.sizeof.(section_name)` operator can be used to obtain the size of a specific section after the link process has occurred. For example, if you wanted to find the final size of the .data section, you could use:

```
mov #.sizeof.(.data), w0
```

> **Note:** When the `.sizeof.(section_name)` operator is used on a section in program memory, the size returned is the size in program counter units. The 16-bit device program counter increments by 2 for each instruction word.

4.5.6 Obtaining the Starting Address of a Specific Section

The `.startof.(section_name)` operator can be used to obtain the starting address of a specific section after the link process has occurred. For example, if you wanted to obtain the starting address of the .data section, you could use:

```
mov #.startof.(.data), w1
```

4.5.7 Accessing Functions in Boot or Secure Segments

Functions in the boot or secure segments without access entries can be referenced like any other function:

```
call func1            ; call func1
mov #handle(func1),w1 ; create 16 bit pointer to func1 (instr)
.word handle(func1)   ; create 16 bit pointer to func1 (data)
.pword func1          ; create 24 bit pointer to func1
```

In order to support the separate linking of boot and secure application segments, access entry points may be defined. Access entry points provide a way to transfer control across segments to a function which may not be defined at link time. For more information about access entry points, see **Section 6.3 "Directives that Define Sections"** and **Section 10.14 "Boot and Secure Segments"**.

The `boot()` and `secure()` operators can be used to reference boot or secure functions via access entry points. These operators can be applied in both instructions and data directives, and will return 16, 24, or 32 bits depending on context.

```
call boot(4)         ; call access entry 4 in the boot segment
rcall secure(4)      ; pc-relative call to secure access entry 4
mov #boot(4),w1      ; load 16 bit pointer to boot entry 4
.word secure(5)      ; create 16 bit pointer to secure entry 5
.pword secure(5)     ; create 24 bit pointer to secure entry 5
.long boot(6)        ; create 32 bit pointer to boot entry 6
goto boot(7)         ; jump to boot entry 7
bra secure(7)        ; unconditional branch to secure entry 7
bra cc, boot(8)      ; conditional branch to boot entry 8
```


© 2008 Microchip Technology Inc.

Chapter 5. Assembler Symbols

5.1 INTRODUCTION

Symbols are defined and their use with MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB ASM30) is discussed.

5.2 HIGHLIGHTS

Topics covered in this chapter are:

- What are Symbols
- Reserved Names
- Local Symbols
- Giving Symbols Other Values
- The Special DOT Symbol
- Using Executable Symbols in a Data Context
- Predefined Symbols

5.3 WHAT ARE SYMBOLS

A symbol is one or more characters chosen from the set of all letters, digits and the two characters underline (_) and period (.). Symbols may not begin with a digit. Case is significant (e.g., foo is a different symbol than Foo). There is no length limit and all characters are significant.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

5.4 RESERVED NAMES

The following symbol names (case-insensitive) are reserved for the assembler. Do not use `.equ`, `.equiv` or `.set` (See **Chapter 6. "Assembler Directives"**) with these symbols.

TABLE 5-1: SYMBOL NAMES – RESERVED

W0	W1	W2	W3	W4	W5	W6	W7
W8	W9	W10	W11	W12	W13	W14	W15
WREG	A	B	OV	C	Z	N	GE
LT	GT	LE	NOV	NC	NZ	NN	GEU
LTU	GTU	LEU	OA	OB	SA	SB	

5.5 LOCAL SYMBOLS

Local symbols are used when temporary scope for a label is needed. There are ten local symbol names, which can be reused throughout the program. They may be referred to using the names '0', '1', ..., '9'. To define a local symbol, write a label of the form 'N:' (where N represents any digit 0-9). To refer to the most recent previous definition of that symbol, write 'Nb', using the same digit as when you defined the label. To refer to the next definition of a local label, write 'Nf'. The 'b' stands for "backwards" and the 'f' stands for "forwards". There is no restriction on how to use these labels, but remember that at any point in assembly, at most, 10 prior local labels and, at most, 10 forward local labels may be referred to.

EXAMPLE 5-1:

```
print_string:
     mov    w0,w1
1:
     cp0.b  [w1]
     bra    z,9f
     mov.b  [w1++],w0
     call   print_char
     bra    1b
9:
     return
```

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages, and optionally emitted to the object file have the following parts:

TABLE 5-2: SYMBOL PARTS

Parts	Description
L	All local labels begin with 'L'.
Digit	If the label is written '0:', then the digit is '0'. If the label is written '1', then the digit is '1'. And so on up through '9'.
CTRL-A	This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value '\001'.
Ordinal number	This is a serial number to keep the labels distinct. The first '0:' gets the number '1'; the 15th '0:' gets the number '15'; and so on. Likewise for the other labels '1:' through '9:'. For instance, the first '1:' is named L1C-A1, the 44th '3:' is named L3C-A44.

EXAMPLE 5-2:

```
00000100 <print_string>:
 100:   80 00 78        mov.w     w0, w1

00000102 <L1·1>:
 102:   11 04 e0        cp0.b     [w1]
 104:   03 00 32        bra       Z, . + 0x8
 106:   31 40 78        mov.b     [w1++], w0
 108:   02 00 07        rcall     . + 0x6
 10a:   fb ff 37        bra       . + 0xFFFFFFF8

0000010c <L9·1>:
 10c:   00 00 06        return
```


© 2008 Microchip Technology Inc.

5.6 GIVING SYMBOLS OTHER VALUES

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign '=', followed by an expression. This is equivalent to using the `.set` directive (See **Chapter 6. "Assembler Directives"**.)

Example:

```
PSV = 4
```

5.7 THE SPECIAL DOT SYMBOL

The special symbol '.' refers to the current address that is being assembled into. Thus, the expression:

```
melvin:  .word .     ; in a data section
```

defines `melvin` to contain its own data address. Assigning a value to . is treated the same as a `.org` directive. Thus the expression:

```
. = .+2
```

is the same as saying:

```
.org .+2
```

The symbol '$' is accepted as a synonym for '.'

When used in an executable section, '.' refers to a Program Counter address. On the 16-bit device, the Program Counter increments by 2 for each instruction word. Odd values are not permitted.

5.8 USING EXECUTABLE SYMBOLS IN A DATA CONTEXT

The 16-bit device modified-Harvard architecture includes separate address spaces for data storage and program storage. Most instructions and assembler directives imply a context which is compatible with symbols from one address space or the other. For example, the CALL instruction implies an executable context, so the assembler reports an error if a program tries to CALL a symbol located in a data section.

Likewise, instructions and directives that imply a data context cannot be used with symbols located in an executable section. Assembling the following code sequence will result in an error, as shown:

```
      .text
msg:  .asciz "Here is an important message"
      mov #msg,w0
:
:
Assembler messages:
Error: Cannot reference executable symbol (msg) in a data context
```

In this example the `mov` instruction implies a data context. Because symbol `msg` is located in an executable section, an error is reported. Possibly the programmer was trying to derive a pointer for use with the PSV window. The special operators described in **Section 4.5 "Special Operators"** can be used whenever an executable symbol must be referenced in a data context:

```
      .text
msg:  .asciz "Here is an important message"
      mov #psvoffset(msg),w0
```

Here the `psvoffset()` operator derives a 16-bit value which is suitable for use in a data context.

The next example shows how the special symbol "." can be used with a data directive in an executable section:

```
        .text
fred:   .long paddr(.)
```

Here the `paddr()` operator derives a 24-bit value which is suitable for use in a data context. The `.long` directive pads the value to 32 bits and encodes it into the `.text` section.

5.9 PREDEFINED SYMBOLS

The assembler predefines several symbols which can be tested by conditional directives in source code.

TABLE 5-3: PREDEFINED SYMBOLS

Symbol	Definition
`__C30COFF`	16-bit compiler COFF output
`__C30ELF`	16-bit compiler ELF output
`__dsPIC30F`	dsPIC30F target device family
`__dsPIC33F`	dsPIC33F target device family
`__PIC24F`	PIC24F target device family
`__PIC24H`	PIC24H target device family
`__MCHP16`	No target device family specified

© 2008 Microchip Technology Inc.

Chapter 6. Assembler Directives

6.1 INTRODUCTION

Directives are assembler commands that appear in the source code but are not usually translated directly into opcodes. They are used to control the assembler: its input, output, and data allocation.

Note: Directives are ***not*** instructions (`movlw`, `btfss`, `goto`, etc). For instruction set information, consult your device data sheet.

While there are some similarities with MPASM assembler directives, most MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB ASM30) directives are new or different in some way. The differences between MPASM assembler and the 16-bit assembler directives have been pointed out in **Appendix D. "MPASM™ Assembler Compatibility"**. All 16-bit directives are preceded by a period ".".

Note: Directives that are supported, but deprecated, are listed in **Appendix C. "Deprecated Features"**.

6.2 HIGHLIGHTS

Topics covered in this chapter are:

- Directives that Define Sections
- Directives that Fill Program Memory
- Directives that Initialize Constants
- Directives that Declare Symbols
- Directives that Define Symbols
- Directives that Modify Section Alignment
- Directives that Format the Output Listing
- Directives that Control Conditional Assembly
- Directives for Substitution/Expansion
- Miscellaneous Directives
- Directives for Debug Information

6.3 DIRECTIVES THAT DEFINE SECTIONS

Sections are locatable blocks of code or data that will occupy contiguous locations in the 16-bit device memory. Three sections are pre-defined: `.text` for executable code, `.data` for initialized data and `.bss` for uninitialized data. Other sections may be defined; the linker defines several that are useful for locating data in specific areas of 16-bit memory.

Section directives are:

- `.bss`
- `.data`
- `.memory name, size(nn) [, origin(aa)]`
- `.pushsection name [, attr1[,...,attrn]]`
- `.popsection`
- `.section name [, "flags"] (deprecated)`
- `.section name [, attr1[,...,attrn]]`
- `.text`

.bss

Definition

Assemble the following statements onto the end of the `.bss` (uninitialized data) section.

Example

```
   ; The following symbols (B1 and B2) will be placed in
   ; the uninitialized data section.
.bss
B1:   .space 4   ; 4 bytes reserved for B1
B2:   .space 1   ; 1 byte reserved for B2
```

.data

Definition

Assemble the following statements onto the end of the `.data` (initialized data) section.

Example

```
   ; The following symbols (D1 and D2) will be placed in
   ; the initialized data section.
.data
D1:   .long 0x12345678   ; 4 bytes
D2:   .byte 0xFF         ; 1 byte
```


© 2008 Microchip Technology Inc.

.memory *name*, size(*nn*) [, origin(*aa*)]

Definition

Define an external memory region for allocation by the linker. Sections may be assigned to region *name* by use of the `memory` section attribute.

Example

```
        ; define an external memory region
        .memory _memory1, size(8192), origin(0)

        ; allocate a section in external memory
        .section  mem1_sec1,memory(_memory1)
        .global _mem1_array1
_mem1_array1:
        .skip   50
```

.pushsection *name* [, $attr_1$[,...,$attr_n$]]

Push the current section description onto the section stack, and assemble the following code into a section named *name*. The syntax is identical to `.section`. Every `.pushsection` should have a matching `.popsection`.

.popsection

Replace the current section description with the top section on the section stack. This section is popped off the stack.

.section *name* [, "*flags*"] (deprecated)

.section *name* [, $attr_1$[,...,$attr_n$]]

Assembles the following code into a section named `name`. If the character * is specified for `name`, the assembler will generate a unique name for the section based on the input file name in the format `filename.scn`*n*, where *n* represents the number of auto-generated section names.

Sections named * can be used to conserve memory because the assembler will not add alignment padding to these sections. Sections that are not named * may be combined across several files, so the assembler must add padding in order to guarantee the requested alignment.

If the optional argument is not present, the section attributes depend on the section name. A table of reserved section names with implied attributes is given in Reserved Section Names with Implied Attributes. If the section name matches a reserved name, the implied attributes will be assigned to that section. If the section name is not recognized as a reserved name, the default attribute will be `data` (initialized storage in data memory).

Implied attributes for reserved section names other than [`.text`, `.data`, `.bss`] are deprecated. A warning will be issued if implied attributes for these reserved section are used.

If the first optional argument is quoted, it is taken as one or more flags that describe the section attributes. Quoted section flags are deprecated (see **Appendix C. "Deprecated Features"**). A warning will be issued if quoted section flags are used.

If the first optional argument is not quoted, it is taken as the first element of an attribute list. Attributes may be specified in any order, and are case-insensitive. Two categories of section attributes exist: attributes that represent section types, and attributes that modify section types.

Attributes that Represent Section Types

Attributes that represent section types are mutually exclusive. At most one of the attributes listed below may be specified for a given section.

TABLE 6-1: ATTRIBUTES THAT REPRESENT SECTION TYPES

Attribute	Description	Support
code	executable code in program memory	All
data	initialized storage in data memory	All
bss	uninitialized storage in data memory	All
persist	persistent storage in data memory	All
psv	constants in program memory	All
eedata	non-volatile storage in data EEPROM	30
memory	external or user-defined memory	All

All = Supported on all devices
30 = Supported on dsPIC30F DSCs

Attributes that Modify Section Types

Depending on the attribute, all or some section types may be modified by it, as below.

TABLE 6-2: ATTRIBUTES THAT MODIFY SECTION TYPES

Attribute	Description	Attribute applies to						
		code	data	bss	persist	psv	eedata	memory
address(a)	locate at absolute address a	All	All	All	All	All	30	All
near	locate in the first 8K of memory	—	All	All	All	—	—	—
xmemory	locate in X address space	—	30/33	30/33	30/33	—	—	—
ymemory	locate in Y address space	—	30/33	30/33	30/33	—	—	—
reverse(n)	align the ending address +1	—	All	All	All	All	30	All
align(n)	align the starting address	All	All	All	All	All	30	All
noload	allocate, do not load	All	All	All	All	All	30	All
merge(n)	mergable elements of size n**	All	All	—	—	All	30	—
info	do not allocate or load	All	All	All	—	—	—	—
dma	locate in DMA space	—	24H/33	24H/33	24H/33	—	—	—
boot	locate in boot segment	CG	—	CG	—	CG	CG	—
secure	locate in secure segment	CG	—	CG	—	CG	CG	—

All = Attribute applies to section – All devices
24H = Attribute applies to section – PIC24H MCUs
30 = Attribute applies to section – dsPIC30F DSCs
33 = Attribute applies to section – dsPIC33F DSCs
CG = Attribute applies to section – CodeGuard™ Security-enabled devices
— = Attribute does not apply to section
** = This attribute could be used by a linker to merge identical constants across input files. If n=0, the section contains null-terminated strings of variable length.

Attributes that modify section types may be used in combination. For example, "xmemory,address(a)" is a valid attribute string, but "xmemory,address(a),ymemory" is not.

© 2008 Microchip Technology Inc.

TABLE 6-3: COMBINING ATTRIBUTES THAT MODIFY SECTION TYPES

	address	near	xmemory	ymemory	reverse	align	noload	merge	info	dma	boot	secure
address		All	All	All	—	—	All	—	—	24H/33	CG	CG
near	All		All	All	All	All	All	All	—	—	—	—
xmemory	30/33	30/33		—	30/33	30/33	30/33	30/33	—	—	—	—
ymemory	30/33	30/33	—		30/33	30/33	30/33	30/33	—	—	—	—
reverse	—	All	All	All		—	All	All	—	24H/33	CG	CG
align	—	All	All	All	—		All	All	—	24H/33	CG	CG
noload	All	All	All	All	All	All		—	—	24H/33	CG	CG
merge	—	All	All	All	All	All	—		—	—	—	—
info	—	—	—	—	—	—	—	—		—	—	—
dma	24H/33	—	—	—	24H/33	24H/33	24H/33	—	—		—	—
boot	CG	—	—	—	CG	CG	CG	—	—	—		—
secure	CG	—	—	—	CG	CG	CG	—	—	—	—	

All = May be combined – All devices
24H = Supported on PIC24H MCUs
30 = Supported on dsPIC30F DSCs
33 = Supported on dsPIC33F DSCs
CG = Supported on CodeGuard™ Security-enabled devices
— = May not be combined

© 2008 Microchip Technology Inc.

Reserved Section Names with Implied Attributes

The following section names are available for user applications and are recognized to have implied attributes:

Reserved Name	Implied Attribute(s)	Support
.text	code	All
.data	data	All
.bss	bss	All
.xbss	bss, xmemory	30/33
.xdata	data, xmemory	30/33
.nbss	bss, near	All
.ndata	data, near	All
.ndconst	data, near	All
.pbss	bss, persist	All
.dconst	data	All
.ybss	bss, ymemory	30/33
.ydata	data, ymemory	30/33
.const	psv	All
.eedata	eedata	30

All = Supported on all devices
30 = Supported on dsPIC30F DSCs
33 = Supported on dsPIC33F DSCs

Reserved section names may be used with explicit attributes. If the explicit attribute(s) conflict with any implied attribute(s), an error will be reported.

Implied attributes for reserved section names other than [.text, .data, .bss] are deprecated. A warning will be issued if these names are used without explicit attributes.

Section Directive Examples

```
.section foo                 ;foo is initialized data memory.

.section bar,bss,xmemory,align(256) ;bar is uninitialized
                                    ;X data memory, aligned.
.section *,data,near                ;section is near
                                    ;initialized data memory.
.section buf1,bss,address(0x800)    ;buf1 is uninitialized
                                    ;data memory at 0x800.
.section tab1,psv,address(0x10000)  ;tab1 is psv constants
                                    ;at 0x10000.
```

Section Directive Examples - Boot/Secure Segments

Program Memory

Attributes can be used to declare protected functions in secure segments:

```
  .section *,code,boot
  .global func1
func1:
  return

  .section *,code,secure
  .global func2
func2:
  return
```


© 2008 Microchip Technology Inc.

A secure function is defined by the combination of `.section` and `.global` directives, and a label. It is recommended that each secure function be defined in a separate section. If the function will be assigned an access entry point, separate sections are required.

An optional argument to `boot` or `secure` can be used to specify a protected access entry point:

```
  .section *,code,boot(3)
  .global func3
func3:
  return

  .section *,code,secure(4)
  .global func4
func4:
  return
```

The optional argument is valid only in code sections. Integers that represent access entry slots must be in the range 0..15 or 17..31. In addition to an entry slot number, the value `unused` may be used to specify an entry for all unused slots in the access entry area:

```
  .section *,code,boot(unused)
  .global func_default
func_default:
  return
```

An interrupt service routine may be specified with the value `isr`:

```
  .section *,code,boot(isr)
  .global func_isr
func_default:
  retfie
```

A section identified with `boot(isr)` or `secure(isr)` will be assigned to access entry slot 16, which is reserved for interrupt functions.

Data Memory

The `boot` and `secure` attributes can be used to define protected variables in boot RAM or secure RAM:

```
  .section *,bss,boot
  .global boot_dat
boot_dat:
  .space 32

  .section *,bss,secure
  .global secure_dat
secure_dat:
  .space 32
```

There is no support for initialized data in protected RAM segments. Therefore `boot` or `secure` cannot be used in combination with attribute `data`. A diagnostic will be reported if initial values are specified in a section that has been designated `boot` or `secure`.

Constants in Non-Volatile Memory

Constants in non-volatile memory can be protected by using the `boot` or `secure` attribute in combination with psv or eedata:

```
  .section *,psv,boot
  .global key1
key1:
  .ascii "abcdefg"
```

```
  .section *,eedata,boot
  .global key2
key2:
  .ascii "hijklm"
```

.text

Definition

Assemble the following statements onto the end of the `.text` (executable code) section.

Example

```
   ; The following code will be placed in the executable
   ; code section.
.text
.global __reset
__reset:
   mov BAR, w1
   mov FOO, w0
LOOP:
   cp0.b [w0]
   bra Z, DONE
   mov.b [w0++], [w1++]
   bra LOOP
DONE:
   .end
```

6.4 DIRECTIVES THAT FILL PROGRAM MEMORY

These directives are only allowed in a code (executable) section. If they are not in a code section, a warning is generated and the rest of the line is ignored.

Fill directives are:

- `.fillupper [value]`
- `.fillvalue [value]`
- `.pfillvalue [value]`
- Section Example

.fillupper [*value*]

Definition

Define the upper byte (bits 16-23) to be used when this byte is skipped due to alignment or data defining directives. If *value* is not specified, it is reset to the default 0x00. Directives that may cause an upper byte to be filled are: `.align`, `.ascii`, `.asciz`, `.byte`, `.double`, `.fill`, `.fixed`, `.float`, `.hword`, `.int`, `.long`, `.skip`, `.space`, `.string` and `.word`. The value is persistent for a given code section, throughout the entire source file, and may be changed to another value by issuing subsequent `.fillupper` directives.

Example

See Section Example that follows.

© 2008 Microchip Technology Inc.

.fillvalue [*value*]

Definition

Define the byte value to be used as fill in a code section when the lower word (bits 0-15) is skipped due to alignment or data defining directives. If *value* is not specified, the default value of 0x0000 is used. Directives that may cause the lower word to filled are: .align, .fill, .skip, .org and .space. The value is persistent for a given code section, throughout the entire source file, and may be changed to another value by issuing subsequent .fillvalue directives.

Example

See Section Example that follows.

.pfillvalue [*value*]

Definition

Define the byte value to be used as fill in a code section when memory (bits 0-23) is skipped due to an alignment or data defining **p** directive. If *value* is not specified, it is reset to its default 0x000000. Directives that may cause a program word to be filled are: .palign, .pfill, .pskip, .porg, and .pspace. The value is persistent for a given code section, throughout the entire source file, and may be changed to another value by issuing subsequent .pfillvalue directives.

Example

See Section Example below.

Section Example

			.section .myconst, code
			.fillvalue 0x12
			.fillupper 0x34
			.pfillvalue 0x56
0x12	0x12	0x34	.fill 4
0x12	0x12		
		0x34	.align 2 ;Align to next p-word
0x56	0x56	0x56	.pfill 8
0x56	0x56	0x56	
0x56	0x56		
		0x56	.palign 2 ;Align to next p-word
			.fillvalue ;Reset fillvalue
			.pfillvalue ;Reset pfillvalue
0x00	0x00	0x34	.fill 4
0x00	0x00		
		0x34	.align 2 ;Align to next p-word
0x00	0x00	0x00	.pfill 8
0x00	0x00	0x00	
0x00	0x00		
		0x00	.palign 2 ;Align to next p-word

6.5 DIRECTIVES THAT INITIALIZE CONSTANTS

Constant initialization directives are:

- .ascii "string$_1$" | <##>$_1$ [, ..., "string$_n$" | <##>$_n$]
- .pascii "string$_1$" | <##>$_1$ [, ..., "string$_n$" | <##>$_n$]
- .pascii "string$_1$"
- .asciz "string$_1$" | <##>$_1$ [, ..., "string$_n$" | <##>$_n$]
- .pasciz "string$_1$" | <##>$_1$ [, ..., "string$_n$" | <##>$_n$]
- .pasciz "string$_2$"
- .byte expr$_1$[, ..., expr$_n$]
- .pbyte expr$_1$[, ..., expr$_n$]
- .double value$_1$[, ..., value$_n$]
- .fixed value$_1$[, ..., value$_n$]
- .float value$_1$[, ..., value$_n$]
- .single value$_1$[, ..., value$_n$]
- .hword expr$_1$[, ..., expr$_n$]
- .int expr$_1$[, ..., expr$_n$]
- .long expr$_1$[, ..., expr$_n$]
- .short expr$_1$[, ..., expr$_n$]
- .string "str"
- .pstring "str"
- .pstring "string$_2$"
- .word expr$_1$[, ..., expr$_n$]
- .pword expr$_1$[, ..., expr$_n$]

.ascii "*string*$_1$" | <##>$_1$ [, ..., "*string*$_n$" | <##>$_n$]

Assembles each string (with no automatic trailing zero byte) or <##> into successive bytes in the current section.<##> is a way of specifying a character by its ASCII code. For example, given that the ASCII code for a new line character is `0xa`, the following two lines are equivalent:

```
.ascii "hello\n","line 2\n"
.ascii "hello",<0xa>,"line 2",<0xa>
```

> **Note:** If the ## is not a number, 0 will be assembled. If the ## is greater than 255, then the value will be truncated to a byte.

If in a code (executable) section, the upper program memory byte will be filled with the last `.fillupper` value specified or the `NOP` opcode (0x00) if no `.fillupper` has been specified.

.pascii "*string*$_1$" | <##>$_1$ [, ..., "*string*$_n$" | <##>$_n$]

Assembles each string (with no automatic trailing zero byte) or <##> into successive bytes into program memory, including the upper byte.<##> is a way of specifying a character by its ASCII code. For example, given that the ASCII code for a new line character is `0xa`, the following two lines are equivalent:

```
.pascii "hello\n","line 2\n"
.pascii "hello",<0xa>,"line 2",<0xa>
```

> **Note:** If the ## is not a number, 0 will be assembled. If the ## is greater than 255, then the value will be truncated to a byte.

© 2008 Microchip Technology Inc.

.pascii "*string*$_1$"

Stores a sequence of ASCII characters (with no automatic trailing zero byte) into program memory, including the upper byte.

.asciz "*string*$_1$" | <##>$_1$ [, ..., "*string*$_n$" | <##>$_n$]

Assembles each string with an automatic trailing zero byte or <##> into successive bytes in the current section.

> **Note:** If the ## is not a number, 0 will be assembled. If the ## is greater than 255, then the value will be truncated to a byte.

If in a code (executable) section, the upper program memory byte will be filled with the last `.fillupper` value specified or the `NOP` opcode (0x00) if no `.fillupper` has been specified.

.pasciz "*string*$_1$" | <##>$_1$ [, ..., "*string*$_n$" | <##>$_n$]

Assembles each string with an automatic trailing zero byte or <##> into program memory, including the upper byte.

> **Note:** If the ## is not a number, 0 will be assembled. If the ## is greater than 255, then the value will be truncated to a byte.

.pasciz "*string*$_2$"

Stores a sequence of ASCII characters (with an automatic trailing zero byte) into program memory, including the upper byte.

.byte *expr*$_1$[, ..., *expr*$_n$]

Assembles one or more successive bytes in the current section.

If in a code (executable) section, the upper program memory byte will be filled with the last `.fillupper` value specified or the `NOP` opcode (0x00) if no `.fillupper` has been specified.

.pbyte *expr*$_1$[, ..., *expr*$_n$]

Assembles one or more successive bytes in the current section. This directive will allow you to create data in the upper byte of program memory.

This directive is only allowed in a code section. If not in a code section, a warning is generated and the rest of the line is ignored.

.double *value*$_1$[, ..., *value*$_n$]

Assembles one or more double-precision (64-bit) floating-point constants into consecutive addresses in little-endian format.

If in a code (executable) section, the upper program memory byte will be filled with the last `.fillupper` value specified or the `NOP` opcode (0x00) if no `.fillupper` has been specified.

Floating point numbers are in IEEE format (see **Section 3.5.1.2 "Floating-Point Numbers"**).

The following statements are equivalent:

```
.double 12345.67
.double 1.234567e4
.double 1.234567e04
.double 1.234567e+04
.double 1.234567E4
.double 1.234567E04
.double 1.234567E+04
```

It is also possible to specify the hexadecimal encoding of a floating point constant. The following statements are equivalent and encode the value 12345.67 as a 64-bit double-precision number:

```
.double 0e:40C81CD5C28F5C29
.double 0f:40C81CD5C28F5C29
.double 0d:40C81CD5C28F5C29
```

.fixed *value*$_1$[, ..., *value*$_n$]

Assembles one or more 2-byte fixed-point constants (range -1.0 <= f < 1.0) into consecutive addresses in little-endian format. Fixed-point numbers are in Q-15 format (**Section 3.5.1.3 "Fixed-Point Numbers"**).

.float *value*$_1$[, ..., *value*$_n$]

Assembles one or more single-precision (32-bit) floating-point constants into consecutive addresses in little-endian format.

If in a code (executable) section, the upper program memory byte will be filled with the last `.fillupper` value specified or the `NOP` opcode (0x00) if no `.fillupper` has been specified.

Floating point numbers are in IEEE format (see **Section 3.5.1.2 "Floating-Point Numbers"**).

The following statements are equivalent:

```
.float 12345.67
.float 1.234567e4
.float 1.234567e04
.float 1.234567e+04
.float 1.234567E4
.float 1.234567E04
.float 1.234567E+04
```

It is also possible to specify the hexadecimal encoding of a floating-point constant. The following statements are equivalent and encode the value 12345.67 as a 32-bit double-precision number:

```
.float 0e:4640E6AE
.float 0f:4640E6AE
.float 0d:4640E6AE
```

.single *value*$_1$[, ..., *value*$_n$]

Assembles one or more single-precision (32-bit), floating-point constants into consecutive addresses in little-endian format.

If in a code (executable) section, the upper program memory byte will be filled with the last `.fillupper` value specified or the `NOP` opcode (0x00) if no `.fillupper` has been specified.

Floating point numbers are in IEEE format.

© 2008 Microchip Technology Inc.

.hword *expr*$_1$[, ..., *expr*$_n$]

Assembles one or more 2-byte numbers into consecutive addresses in little-endian format.

.int *expr*$_1$[, ..., *expr*$_n$]

Assembles one or more 2-byte numbers into consecutive addresses in little-endian format.

.long *expr*$_1$[, ..., *expr*$_n$]

Assembles one or more 4-byte numbers into consecutive addresses in little-endian format.

.short *expr*$_1$[, ..., *expr*$_n$]

Same as `.word`.

.string "*str*"

Same as `.asciz`.

.pstring "*str*"

Same as `.pasciz`.

.pstring "*string*$_2$"

Same as `.pasciz "string`$_2$`"`.

.word *expr*$_1$[, ..., *expr*$_n$]

Assembles one or more 2-byte numbers into consecutive addresses in little-endian format.

.pword *expr*$_1$[, ..., *expr*$_n$]

Assembles one or more 3-byte numbers into consecutive addresses in the current section.

This directive is only allowed in a code section. If not in a code section, a warning is generated and the rest of the line is ignored.

6.6 DIRECTIVES THAT DECLARE SYMBOLS

Declare symbol directives are:

- .bss symbol, length [, algn]
- .comm symbol, length [, algn]
- .extern symbol
- .global symbol .globl symbol
- .lcomm symbol, length
- .weak symbol

.bss *symbol*, *length* [, *algn*]

Reserve *length* (an absolute expression) bytes for a local symbol. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. *symbol* is declared local so it is not visible to other objects. If algn is specified, it is the address alignment required for *symbol*. The bss location counter is advanced until it is a multiple of the requested alignment. The requested alignment must be a power of 2.

.comm *symbol*, *length* [, *algn*]

Declares a common symbol named *symbol*. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If the linker does not see a definition for that symbol, then it will allocate length bytes of uninitialized memory. If the linker sees multiple common symbols with the same name, and they do not all have the same size, the linker will allocate space using the largest size.

If *algn* is specified, it is the address alignment required for *symbol*. The requested alignment must be a power of two. *algn* is supported when the object file format is ELF; otherwise, it is ignored.

.extern *symbol*

Declares a symbol name that may be used in the current module, but it is defined as global in a different module.

.global *symbol*
.globl *symbol*

Declares a symbol *symbol* that is defined in the current module and is available to other modules.

.lcomm *symbol*, *length*

Reserve *length* bytes for a local common denoted by symbol. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time, the bytes start off zeroed. *symbol* is not declared global so it is normally not visible to the linker.

.weak *symbol*

Marks the symbol named *symbol* as weak. When a weak-defined symbol is linked with a normal-defined symbol, the normal-defined symbol is used with no error. When a weak-undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

© 2008 Microchip Technology Inc.

6.7 DIRECTIVES THAT DEFINE SYMBOLS

Define symbol directives are:

- `.equ symbol, expression`
- `.equiv symbol, expression`
- `.set symbol, expression`

.equ *symbol, expression*

Set the value of *symbol* to *expression*. You may set a symbol any number of times in assembly. If you set a global symbol, the value stored in the object file is the last value equated to it.

.equiv *symbol, expression*

Like `.equ`, except the assembler will signal an error if symbol is already defined.

.set *symbol, expression*

Same as `.equ`.

6.8 DIRECTIVES THAT MODIFY SECTION ALIGNMENT

There are two ways to modify section alignment: implicitly and explicitly. Implicit alignment occurs first.

- Implicit Alignment in Program Memory
- Explicit Section Alignment Directives

6.8.1 Implicit Alignment in Program Memory

In addition to directives that explicitly align the location counter (such as `.align`, `.palign`, `.org`, `.porg`, etc) many statements cause an implicit alignment to occur under certain conditions. Implicit alignment occurs when padding is inserted so that the next statement begins at a valid address. Padding uses the current `.fillvalue` and `.fillupper` values if specified; otherwise the value zero is used.

In data memory, a valid address is available for each byte. Since no data directives specify memory in quantities of less than one byte, implicit alignment is not required in data memory.

In program memory, a valid address is available for each instruction word (3 bytes). Since data directives can specify individual bytes, implict alignment to the next valid address is sometimes required.

The following conditions cause implicit alignment in program memory:

1. Labels must be aligned to a valid address.

   For example, the following source code:

```
   .text
   .pbyte 0x11
L1:
   .pbyte 0x22
   .pbyte 0x33,0x44
```

   generates implicit alignment as shown:

```
Disassembly of section .text:
00000000 <.text>:
   0:   11 00 00        nop
00000002 <L1>:
   2:   22 33 44        .pword 0x443322
```

> **Note:** Two bytes of padding were inserted so that label `L1` would be aligned to a valid address.

2. Instructions must be aligned to a valid address.

   For example, the following source code:

```
.text
.pbyte 0x11
 mov   w2,w3
```

   generates implicit alignment as shown:

```
Disassembly of section .text:
00000000 <.text>:
   0:   11 00 00        nop
   2:   82 01 78        mov.w     w2, w3
```

> **Note:** Two bytes of padding were inserted so that the `mov` instruction would be aligned to a valid address.

© 2008 Microchip Technology Inc.

3. Transitions between p-type data directives (`.pbyte`, `.pspace`, etc). and normal data directives (`.byte`, `.space`, etc.), in either direction, are aligned to a valid address.

   For example, the following source code:

```
.text
.byte  0x11
.pbyte 0x22
.pbyte 0x33,0x44
```

   generates implicit alignment as shown:

```
Disassembly of section .text:
00000000 <.text>:
   0:   11 00 00        nop
   2:   22 33 44        .pword 0x443322
```

Note: Two bytes of padding were inserted so that the transition from normal to p-type directive would be aligned to a valid address.

6.8.2 Explicit Section Alignment Directives

Directives that explicitly modify section alignment are:

- `.align algn[, fill[, max-skip]]`
- `.palign algn[, fill[, max-skip]]`
- `.fill repeat[, size[, fill]]`
- `.pfill repeat[, size[, fill]]`
- `.org new-lc[, fill]`
- `.porg new-lc[, fill]`
- `.skip size[, fill] .space size[, fill]`
- `.pskip size[, fill] .pspace size[, fill]`
- `.struct expression`

`.align` *`algn`*`[,` *`fill`*`[,` *`max-skip`*`]]`

Pad the location counter (in the current subsection) to a particular storage boundary.

`algn` is the address alignment required. The location counter is advanced until it is a multiple of the requested alignment. If the location counter is already a multiple of the requested alignment, no change is needed or made. In a code section, an alignment of 2 is required to align to the next instruction word. The requested alignment must be a power of 2.

`fill` is optional. If not specified:

- In a data section, a value of 0x00 is used to fill the skipped bytes.
- In a code section, the last specified `.fillvalue` is used to fill the lower two bytes of program memory and the last specified `.fillupper` is used to fill the upper program memory byte.

`max-skip` is optional. If specified, it is the maximum number of bytes that should be skipped by this directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all.

Alignment within a section is required for modulo addressing. It is worth noting that the overall section alignment reflects the greatest alignment of any `.align` directives that are included. Further, the assembler must pad out the section length to match its alignment. This is done in order to preserve the requested alignment in case the section is combined with other sections of the same name during the link. To avoid unneccessary padding of aligned sections, use the section name *, which identifies a unique section that will never be combined.

.palign *align*[, *fill*[, *max-skip*]]

Pad the location counter (in the current subsection) to a particular storage boundary.

This directive is only allowed in a code section. If not in a code section, a warning is generated and the rest of the line is ignored.

align is the address alignment required. The location counter is advanced until it is a multiple of the requested alignment. If the location counter is already a multiple of the requested alignment, no change is needed. In a code section, an alignment of 2 is required to align to the next instruction word. The requested alignment must be a power of 2.

fill is optional. If not specified, the last .pfillvalue specified is used to fill the skipped bytes. All three bytes of the program memory word are filled.

max-skip is optional. If specified, it is the maximum number of bytes (including the upper byte) that should be skipped by this directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all.

.fill *repeat*[, *size*[, *fill*]]

Reserve `repeat` copies of *size* bytes. *repeat* may be zero or more. `size` may be zero or more, but if it is more than 8, then it is deemed to have the value 8. The content of each *repeat* bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are value rendered in the little-endian byte-order. Each *size* bytes in a repetition is taken from the lowest order *size* bytes of this number.

size is optional and defaults to one if omitted.

fill is optional. If not specified:

- In a data section, a value of 0x00 is used to fill the skipped bytes.
- In a code section, the last specified `.fillvalue` is used to fill the lower two bytes of program memory and the last specified `.fillupper` is used to fill the upper program memory byte.

.pfill *repeat*[, *size*[, *fill*]]

Reserve *repeat* copies of *size* bytes including the upper byte. *repeat* may be zero or more. *size* may be zero or more, but if it is more than 8, then it is deemed to have the value 8. The content of each *repeat* byte is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are value rendered in the little-endian byte-order. Each *size* byte in a repetition is taken from the lowest order *size* bytes of this number.

This directive is only allowed in a code section. If not in a code section, a warning is generated and the rest of the line is ignored.

size is optional and defaults to one if omitted. Size is the number of bytes to reserve (including the upper byte).

© 2008 Microchip Technology Inc.

fill is optional. If not specified, it defaults to the last .pfillvalue specified. All three bytes of each instruction word are filled.

.org *new-lc*[, *fill*]

Advance the location counter of the current section to *new-lc*. In program memory, *new-lc* is specified in Program Counter units. On the 16-bit device, the Program Counter increments by 2 for each instruction word. Odd values are not permitted.

The bytes between the current location counter and the new location counter are filled with fill. *new-lc* is an absolute expression. You cannot .org backwards. You cannot use .org to cross sections.

The new location counter is relative to the current module and is not an absolute address.

fill is optional. If not specified:

- In a data section, a value of 0x00 is used to fill the skipped bytes.
- In a code section, the last specified .fillvalue is used to fill the lower two bytes of program memory and the last specified .fillupper is used to fill the upper program memory byte.

.porg *new-lc*[, *fill*]

Advance the location counter of the current section to *new-lc*. In program memory, *new-lc* is specified in Program Counter units. On the 16-bit device, the Program Counter increments by 2 for each instruction word. Odd values are not permitted.

The bytes between the current location counter and the new location counter are filled with fill. *new-lc* is an absolute expression. You cannot .porg backwards. You cannot use .porg to cross sections.

The new location counter is relative to the current module and is not an absolute address.

This directive is only allowed in a code section. If not in a code section, a warning is generated and the rest of the line is ignored.

fill is optional. If not specified, it defaults to the last .pfillvalue specified. All three bytes of each instruction word are filled.

.skip *size*[, *fill*]
.space *size*[, *fill*]

Reserve *size* bytes. Each byte is filled with the value *fill*.

fill is optional. If the value specified for *fill* is larger than a byte, a warning is displayed and the value is truncated to a byte. If not specified:

- In a data section, a value of 0x00 is used to fill the skipped bytes.
- In a code section, the last specified .fillvalue is used to fill the lower two bytes of program memory and the last specified .fillupper is used to fill the upper program memory byte.

.pskip *size*[, *fill*]
.pspace *size*[, *fill*]

Reserve *size* bytes (including the upper byte). Each byte is filled with the value *fill*.

This directive is only allowed in a code section. If not in a code section, a warning is generated and the rest of the line is ignored.

The new location counter is relative to the current module and is not an absolute address.

fill is optional. If the value specified for *fill* is larger than a byte, a warning is displayed and the value is truncated to a byte. If not specified, it defaults to the last .pfillvalue specified. All three bytes of each instruction word are filled.

.struct *expression*

Switch to the absolute section, and set the section offset to *expression*, which must be an absolute expression. You might use this as follows:

```
     .struct 0
field1:
     .struct field1 + 4
field2:
     .struct field2 + 4
field3:
```

This would define the symbol field1 to have the value 0, the symbol field2 to have the value 4, and the symbol field3 to have the value 8. Assembly would be left in the absolute section, and you would need to use a .section directive of some sort to change to some other section before further assembly.

© 2008 Microchip Technology Inc.

6.9 DIRECTIVES THAT FORMAT THE OUTPUT LISTING

Output listing format directives are:

- `.eject`
- `.list`
- `.nolist`
- `.psize lines[, columns]`
- `.sbttl "subheading"`
- `.title "heading"`

.eject

Force a page break at this point when generating assembly listings.

.list

Controls (in conjunction with `.nolist`) whether assembly listings are generated. This directive increments an internal counter (which is one initially). Assembly listings are generated if this counter is greater than zero.

Only functional when listings are enabled with the `-a` command line option and forms processing has not been disabled with the `-an` command line option.

.nolist

Controls (in conjunction with `.list`) whether assembly listings are generated. This directive decrements an internal counter (which is one initially). Assembly listings are generated if this counter is greater than zero.

Only functional when listings are enabled with the `-a` command line option and forms processing has not been disabled with the `-an` command line option.

.psize *lines*[, *columns*]

Declares the number of lines, and optionally, the number of columns to use for each page when generating listings.

Only functional when listings are enabled with the `-a` command line option and forms processing has not been disabled with the `-an` command line option.

.sbttl "*subheading*"

Use subheading as a subtitle (third line, immediately after the title line) when generating assembly listings. This directive affects subsequent pages, as well as the current page, if it appears within ten lines of the top.

.title "*heading*"

Use heading as the title (second line, immediately after the source file name and page number) when generating assembly listings.

6.10 DIRECTIVES THAT CONTROL CONDITIONAL ASSEMBLY

Conditional assembly directives are:

- `.else`
- `.elseif expr`
- `.endif`
- `.err`
- `.error "string"`
- `.if expr`
- `.ifdecl symbol`
- `.ifndecl symbol .ifnotdecl symbol`
- `.ifdef symbol`
- `.ifndef symbol .ifnotdef symbol`

.else

Used in conjunction with the `.if` directive to provide an alternative path of assembly code should the `.if` evaluate to false.

.elseif *expr*

Used in conjunction with the `.if` directive to provide an alternative path of assembly code should the `.if` evaluate to false and a second condition exists.

.endif

Marks the end of a block of code that is only assembled conditionally.

.err

If the assembler sees an `.err` directive, it will print an error message, and unless the `-Z` option was used, it will not generate an object file. This can be used to signal an error in conditionally compiled code.

.error "*string*"

Similar to `.err`, except that the specified string is printed.

.if *expr*

Marks the beginning of a section of code that is only considered part of the source program being assembled if the argument *expr* is non-zero. The end of the conditional section of code must be marked by an `.endif`; optionally, you may include code for the alternative condition, flagged by `.else`.

.ifdecl *symbol*

Assembles the following section of code if the specified symbol has been declared.

.ifndecl *symbol*
.ifnotdecl *symbol*

Assembles the following section of code if the specified symbol has **not** been declared.

© 2008 Microchip Technology Inc.

.ifdef *symbol*

Assembles the following section of code if the specified symbol has been defined (i.e., assigned a value.)

.ifndef *symbol*
.ifnotdef *symbol*

Assembles the following section of code if the specified symbol has **not** been defined (i.e., not assigned a value.)

6.11 DIRECTIVES FOR SUBSTITUTION/EXPANSION

Substitution/expansion directives are:

- `.exitm`
- `.irp symbol, value1 [, ..., valuen] ... .endr`
- `.irpc symbol, value ... .endr`
- `.macro symbol arg1[=default] [, ..., argn [=default] ] ... .endm`
- `.purgem "name"`
- `.rept count ... .endr`

.exitm

Exit early from the current marco definition. See `.macro` directive.

.irp *symbol*, *value*$_1$
[, ..., *value*$_n$]
...
.endr

Evaluate a sequence of statements assigning different values to *`symbol`*. The sequence of statements starts at the `.irp` directive, and is terminated by a `.endr` directive. For each *`value`*, *`symbol`* is set to *`value`*, and the sequence of statements is assembled. If no `value` is listed, the sequence of statements is assembled once, with `symbol` set to the null string. To refer to *`symbol`* within the sequence of statements, use `\`*`symbol`*.

For example, assembling

```
.irp reg,0,1,2,3
push w\reg
.endr
```

is equivalent to assembling

```
push w0
push w1
push w2
push w3
```

.irpc *symbol*, *value*
...
.endr

Evaluate a sequence of statements assigning different values to *symbol*. The sequence of statements starts at the .irpc directive and is terminated by a .endr directive. For each character in value, *symbol* is set to the character, and the sequence of statements is assembled. If no *value* is listed, the sequence of statements is assembled once, with symbol set to the null string. To refer to *symbol* within the sequence of statements, use *symbol*.

For example, assembling

```
irpc reg,0123
push w\reg
.endr
```

is equivalent to assembling

```
push w0
push w1
push w2
push w3
```

.macro *symbol* arg_1 [=*default*]
[, ..., arg_n [=*default*]]
...
.endm

Define macros that generate assembly output. A macro accepts optional arguments, and can call other macros or even itself, recursively.

If a macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. To refer to arguments within the macro block, use *arg* or &*arg*&. The second form can be used to combine an argument with additional characters to create a symbol name.

For example, assembling:

```
.macro display_int sym
  mov \sym,w0
  rcall display
.endm

display_int result
```

is equivalent to assembling:

```
mov result,w0
rcall display
```


© 2008 Microchip Technology Inc.

In the next example, a macro is used to define HI- and LO-word constants for a 32-bit integer.

```
        .macro LCONST name,value
        .equ   \name,\value
        .equ   &name&LO,(\value) & 0xFFFF
        .equ   &name&HI,((\value)>>16) & 0xFFFF
        .endm

        LCONST seconds_per_day 60*60*24

        mov    #seconds_per_dayLO,w0
        mov    #seconds_per_dayHI,w1
```

pic30-as maintains a counter of how many macros have been executed in the psuedo-variable \@. This value can be copied to the assembly output, but only within a macro definition. In the following example, a recursive macro is used to allocate an arbitrary number of labeled buffers.

```
        .macro make_buffers num,size
BUF\@:  .space \size
        .if (\num - 1)
        make_buffers (\num - 1),\size
        .endif
        .endm

        .bss
        make_buffers 4,16       ; create BUF0..BUF3, 16 bytes each
```

.purgem "*name*"

Undefine the macro *name*, so that later uses of the string will not be expanded. See .marco directive.

.rept *count*
...
.endr

Repeat the sequence of lines between the .rept directive and the next .endr directive count times.

For example, assembling

```
.rept 3
.long 0
.endr
```

is equivalent to assembling

```
.long 0
.long 0
.long 0
```

6.12 MISCELLANEOUS DIRECTIVES

Miscellaneous directives are:

- `.abort`
- `.appline line-number .ln line-number`
- `.end`
- `.fail expression`
- `.ident "comment"`
- `.incbin "file"[,skip[,count]]`
- `.include "file"`
- `.loc file-number, line-number`
- `.pincbin "file"[,skip[,count]]`
- `.print "string"`
- `.version "string"`

.abort

Prints out the message ".abort detected. Abandoning ship." and exits the program.

.appline *line-number*
.ln *line-number*

Change the logical line number. The next line has that logical line number.

.end

End program

.fail *expression*

Generates an error or a warning. If the value of the *expression* is 500 or more, `as` will print a warning message. If the value is less than 500, `as` will print an error message. The message will include the value of *expression*. This can occasionally be useful inside complex nested macros or conditional assembly.

.ident "*comment*"

Appends *comment* to the section named `.comment`. This section is created if it does not exist. The 16-bit linker will ignore this section when allocating program and data memory, but will combine all `.comment` sections together, in link order.

.incbin "*file*"[,*skip*[,*count*]]

The `.incbin` directive includes *file* verbatim at the current location. The file is assumed to contain binary data. The search paths used can be specified with the `-I` command-line option (see **Chapter 2. "Assembler Command Line Interface"**). Quotation marks are required around *file*.

The *skip* argument skips a number of bytes from the start of the file. The *count* argument indicates the maximum number of bytes to read. Note that the data is not aligned in any way, so it is the user's responsibility to make sure that proper alignment is provided both before and after the `.incbin` directive.

When used in an executable section, `.incbin` fills only the lower 16 bits of each program word.

© 2008 Microchip Technology Inc.

.include "*file*"

Provides a way to include supporting files at specified points in your source code. The code is assembled as if it followed the point of the `.include`. When the end of the included file is reached, assembly of the original file continues at the statement following the `.include`.

.loc *file-number, line-number*

`.loc` is essentially the same as `.ln`. Expects that this directive occurs in the `.text` section. *`file-number`* is ignored.

.pincbin "*file*"[,*skip*[,*count*]]

The `.pincbin` directive includes *`file`* verbatim at the current location. The file is assumed to contain binary data. The search paths used can be specified with the `-I` command-line option (see **Chapter 2. "Assembler Command Line Interface"**). Quotation marks are required around *`file`*.

The *`skip`* argument skips a number of bytes from the start of the file. The *`count`* argument indicates the maximum number of bytes to read. Note that the data is not aligned in any way, so it is the user's responsibility to make sure that proper alignment is provided both before and after the `.pincbin` directive.

`.pincbin` is supported only in executable sections, and fills all 24 bits of each program word.

.print "*string*"

Prints *`string`* on the standard output during assembly.

.version "*string*"

This directive creates a `.note` section and places into it an ELF formatted note of type NT_VERSION. The note's name is set to `string`. `.version` is supported when the output file format is ELF; otherwise, it is ignored.

6.13 DIRECTIVES FOR DEBUG INFORMATION

Debug information directives are:

- `.def name`
- `.dim`
- `.endef`
- `.file "string"`
- `.line line-number`
- `.scl class`
- `.size expression`
- `.size name, expression`
- `.sleb128 expr1 [, ..., exprn]`
- `.tag structname`
- `.type value`
- `.type name, description`
- `.uleb128 expr1[,...,exprn]`
- `.val addr`

.def *name*

Begin defining debugging information for a symbol *name*; the definition extends until the `.endef` directive is encountered.

.dim

Generated by compilers to include auxiliary debugging information in the symbol table. Only permitted inside `.def`/`.endef` pairs.

.endef

Flags the end of a symbol definition begun with `.def`.

.file "*string*"

Tells the assembler that it is about to start a new logical file. This information is placed into the object file.

.line *line-number*

Generated by compilers to include auxiliary symbol information for debugging. Only permitted inside `.def`/`.endef` pairs.

.scl *class*

Set the storage class value for a symbol. May only be used within `.def`/`.endef` pairs.

.size *expression*

Generated by compilers to include auxiliary debugging information in the symbol table. Only permitted inside `.def`/`.endef` pairs.

© 2008 Microchip Technology Inc.

.size *name, expression*

Generated by compilers to include auxilliary information for debugging. This variation of `.size` is supported when the output file format is ELF.

.sleb128 $expr_1$ [, ..., $expr_n$]

Signed little endian base 128. Compact variable length representation of numbers used by the DWARF symbolic debugging format.

.tag *structname*

Generated by compilers to include auxiliary debugging information in the symbol table. Only permitted inside `.def`/`.endef` pairs. Tags are used to link structure definitions in the symbol table with instances of those structures.

.type *value*

Records the integer value as the type attribute of a symbol table entry. Only permitted within `.def`/`.endef` pairs.

.type *name, description*

Sets the type of symbol name to be either a function symbol or an object symbol. This variation of `.type` is supported when the output file format is ELF. For example,

```
        .text
        .type foo,@function
foo:
        return

        .data
        .type dat,@object
dat:    .word 0x1234
```

.uleb128 $expr_1$[,...,$expr_n$]

Unsigned little endian base 128. Compact variable length representation of numbers used by the DWARF symbolic debugging format.

.val *addr*

Records the address addr as the value attribute of a symbol table entry. Only permitted within `.def`/`.endef` pairs.

NOTES:

© 2008 Microchip Technology Inc.

Part 2 – MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs

Chapter 7. Linker Overview 81
Chapter 8. Linker Command Line Interface 87
Chapter 9. Linker Scripts 101
Chapter 10. Linker Processing 135
Chapter 11. Linker Examples 167

NOTES:

© 2008 Microchip Technology Inc.

Chapter 7. Linker Overview

7.1 INTRODUCTION

MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB LINK30) produces binary code from relocatable object code and archives for the dsPIC30F/33F DSC and PIC24X MCU family of devices. The 16-bit linker is a Windows console application that provides a platform for developing executable code. The linker is a part of the GNU linker from the Free Software Foundation.

7.2 HIGHLIGHTS

Topics covered in this chapter are:

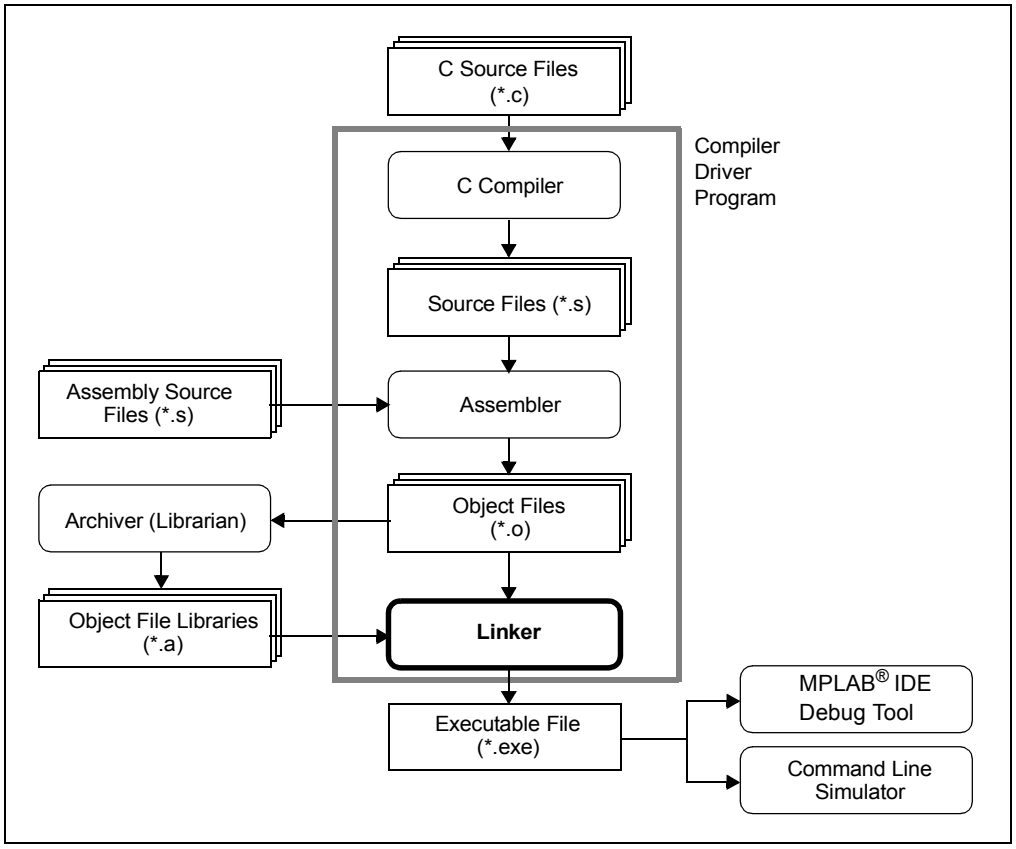
- Linker and Other Development Tools
- Feature Set
- Input/Output Files

7.3 LINKER AND OTHER DEVELOPMENT TOOLS

The 16-bit linker translates object files from the 16-bit assembler and archives files from the 16-bit archiver/librarian into an executable file. See Figure 7-1 for an overview of the tools process flow.

For information on using the 16-bit linker with MPLAB IDE, see *"16-Bit Language Tools Getting Started*" (DS70094).

FIGURE 7-1: TOOLS PROCESS FLOW



© 2008 Microchip Technology Inc.

7.4 FEATURE SET

Notable features of the linker include:

- Automatic or user-defined stack allocation
- Supports 16-bit Program Space Visibility (PSV) window
- Available for Windows
- Command Line Interface
- Linker scripts for all 16-bit devices
- Integrated component of MPLAB IDE

7.5 INPUT/OUTPUT FILES

Linker input and output files are listed below.

TABLE 7-1: LINKER FILES

Extension	Description
Input	
`.o`	object file
`.a`	library file
`.gld`	linker script file
Output	
`.exe, .out`	binary file
`.map`	map file

Unlike the MPLINK linker, the 16-bit linker does not generate absolute listing files. The 16-bit linker is capable of creating a map file and a binary file (that may or may not contain debugging information).

7.5.1 Object Files

Relocatable code produced from source files. The linker accepts COFF format object files by default. To specify COFF or ELF object format explicitly, use the `-omf` command line option, as shown:

```
pic30-ld -omf=elf ...
```

Alternatively, the environment variable `PIC30_OMF` may be used to specify object file format for the dsPIC30F language tools.

7.5.2 Library Files

A collection of object files grouped together for convenience.

7.5.3 Linker Script File

Linker scripts, or command files:

- Instruct the linker where to locate sections
- Specify memory ranges for a given part
- Can be customized to locate user-defined sections at specific addresses

For more on linker script files, see **Chapter 9. "Linker Scripts"**.

EXAMPLE 7-1: LINKER SCRIPT

```
OUTPUT_FORMAT("coff-pic30")
OUTPUT_ARCH("pic30")

MEMORY
{
  data  (a!xr) : ORIGIN = 0x800, LENGTH = 1024
  program (xr) : ORIGIN = 0, LENGTH = (8K * 2)
}

SECTIONS
{
.text :
  {
    *(.vector);
    *(.handle);
    *(.text);
  } >program

.bss (NOLOAD):
  {
    *(.bss);
  } >data

.data :
  {
    *(.data);
  } >data
} /* SECTIONS */

WREG0 = 0x00;
WREG1 = 0x02;
```

7.5.4 Linker Output File

By default, the name of the linker output binary file is `a.out`. You can override the default name by specifying the `-o` option on the command line. The format of the binary file is an executable COFF file by default. To specify a COFF or ELF executable file, use the `-omf` option as shown in **Section 7.5.1 "Object Files"**.

7.5.5 Map File

The map files produced by the linker consist of:

- Archive Member Table – lists the name of any members from archive files that are included in the link.
- Memory Usage Report – shows the starting address and length of all output sections in program memory, data memory and dynamic memory.
- External Symbol Table – lists all external symbols in data and program memory.
- Memory Configuration – lists all of the memory regions defined for the link.
- Linker Script and Memory Map – shows modules, sections and symbols that are included in the link as specified in the linker script.

© 2008 Microchip Technology Inc.

EXAMPLE 7-2: MAP FILE

```
Archive member included       because of file (symbol)

./libpic30.a(crt0.o)          t1.o (_reset)


Program Memory Usage

section   address   length (PC units)   length (bytes) (dec)
-------   -------   -----------------   --------------------
.text           0               0x106           0x189  (393)
.libtext    0x106                0x80            0xc0  (192)
.dinit      0x186                 0x8             0xc  (12)

    Total program memory used (bytes):          0x255  (597)  2%


Data Memory Usage

section   address      alignment gaps    total length  (dec)
-------   -------      --------------    -------------------
.bss        0x800                   0           0x100  (256)

       Total data memory used (bytes):          0x100  (256) 25%


Dynamic Memory Usage

region    address                      maximum length  (dec)
------    -------                      ----------------------
heap        0x900                                 0  (0)
stack       0x900                            0x2f8  (760)

       Maximum dynamic memory (bytes):          0x2f8  (760)

External Symbols in Program Memory (by address):
               0x0000fc                  main
               0x000106                  _reset
               0x000106                  _resetPRI
               0x00011a                  _psv_init
               0x00012a                  _data_init

External Symbols in Program Memory (by name):
               0x00012a                  _data_init
               0x00011a                  _psv_init
               0x000106                  _reset
               0x000106                  _resetPRI
               0x0000fc                  main

Memory Configuration

Name             Origin             Length             Attributes
data             0x000800           0x000400           a !xr
program          0x000000           0x004000           xr
```

```
Linker script and memory map

LOAD t1.o

.text             0x000000        0x106
 *(.vector)
 .vector          0x000000         0xfc t1.o
 *(.handle)
 *(.text)
 .text            0x0000fc          0xa t1.o
                  0x0000fc                  main

.bss                0x0800        0x100
 *(.bss)
 .bss               0x0800        0x100 t1.o

.data               0x0900          0x0
 *(.data)
                    0x0000                  WREG0=0x0
                    0x0002                  WREG1=0x2
LOAD ./libpic30.a
OUTPUT(t.exe coff-pic30)
LOAD data_init

.libtext          0x000106         0x80
 .libtext         0x000106         0x80 ./libpic30.a(crt0.o)
                  0x000106                  _reset
                  0x000106                  _resetPRI
                  0x00011a                  _psv_init
                  0x00012a                  _data_init

.dinit            0x000186          0x8
 .dinit           0x000186          0x8 data_init
```


© 2008 Microchip Technology Inc.

Chapter 8. Linker Command Line Interface

8.1 INTRODUCTION

MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB LINK30) may be used on the command line interface as well as with MPLAB IDE. For information on using the linker with MPLAB IDE, please refer to "*16-bit Language Tools Getting Started*" (DS70094).

8.2 HIGHLIGHTS

Topics covered in this chapter are:

- Syntax
- Options that Control Output File Creation
- Options that Control Run-time Initialization
- Options that Control Informational Output
- Options that Modify the Link Map Output
- Options that Specify CodeGuard™ Security Features

8.3 SYNTAX

The linker supports a plethora of command line options, but in actual practice few of them are used in any particular context.

```
pic30-ld [options] file...
```

Note:	Command line options are case sensitive.

For instance, a frequent use of `pic30-ld` is to link object files and archives to produce a binary file. To link a file `hello.o`:

```
pic30-ld -o output hello.o -lpic30
```

This tells `pic30-ld` to produce a file called `output` as the result of linking the file `hello.o` with the archive `libpic30.a`.

When linking a C application, there are typically several archives (also known as "libraries") which are included in the link command. The list of archives may be specified within `--start-group`, `--end-group` options to help resolve circular references:

```
pic30-ld -o output hello.o --start-group -lpic30 -lm -lc --end-group
```

The command line options to `pic30-ld` may be specified in any order, and may be repeated at will. Repeating most options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that option. Options that may be meaningfully specified more than once are noted in the descriptions below.

Non-option arguments are object files that are to be linked together. They may follow, precede or be mixed in with command line options, except that an object file argument may not be placed between an option and its argument.

Usually the linker is invoked with at least one object file, but you can specify other forms of binary input files using `-l` and the script command language. If no binary input files are specified, the linker does not produce any output, and issues the message 'No input files'.

If the linker cannot recognize the format of an object file, it will assume that it is a linker script. A script specified in this way augments the main linker script used for the link (either the default linker script or the one specified by using `-T`). This feature permits the linker to link against a file that appears to be an object or an archive, but actually merely defines some symbol values, or uses `INPUT` or `GROUP` to load other objects.

For options whose names are a single letter, option arguments must either follow the option letter without intervening white space, or be given as separate arguments immediately following the option that requires them.

For options whose names are multiple letters, either one dash or two can precede the option name; for example, `-trace-symbol` and `--trace-symbol` are equivalent. There is one exception to this rule. Multiple-letter options that begin with the letter `o` can only be preceded by two dashes.

Arguments to multiple-letter options must either be separated from the option name by an equals sign, or be given as separate arguments immediately following the option that requires them. For example, `--trace-symbol srec` and `--trace-symbol=srec` are equivalent. Unique abbreviations of the names of multiple-letter options are accepted.

8.4 OPTIONS THAT CONTROL OUTPUT FILE CREATION

Output file creation options are:

- `--architecture arch (-A arch)`
- `-( archives -), --start-group archives, --end-group`
- `-d, -dc, -dp`
- `--defsym sym=expr`
- `--discard-all (-x)`
- `--discard-locals (-X)`
- `--fill-upper value`
- `--force-exe-suffix`
- `--force-link`
- `--no-force-link`
- `--isr`
- `--no-isr`
- `--library libname (-l libname)`
- `--library-path <dir> (-L <dir>)`
- `--no-keep-memory`
- `--noinhibit-exec`
- `-omf=format`
- `--output file (-o file)`
- `-p,--processor PROC`
- `--relocatable (-r, -i, -Ur)`
- `--retain-symbols-file file`
- `--script file (-T file)`
- `--smart-io`
- `--no-smart-io`
- `--strip-all (-s)`

© 2008 Microchip Technology Inc.

- --strip-debug (-S)
- -Tbss address
- -Tdata address
- -Ttext address
- --undefined symbol (-u symbol)
- --no-undefined
- --wrap symbol

8.4.1 --architecture *arch* (-A *arch*)

Set architecture.

The architecture argument identifies the particular architecture in the 16-bit devices, enabling some safeguards and modifying the archive-library search path.

8.4.2 -(*archives* -), --start-group *archives*, --end-group

Start and end a group.

The archives should be a list of archive files. They may be either explicit file names, or -l options. The specified archives are searched repeatedly until no new undefined references are created. Normally, an archive is searched only once in the order that it is specified on the command line. If a symbol in that archive is needed to resolve an undefined symbol referred to by an object in an archive that appears later on the command line, the linker would not be able to resolve that reference. By grouping the archives, they will all be searched repeatedly until all possible references are resolved. Using this option has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more archives.

8.4.3 -d, -dc, -dp

Force common symbols to be defined.

Assign space to common symbols even if a relocatable output file is specified (with -r).

8.4.4 --defsym *sym=expr*

Define a symbol.

Create a global symbol in the output file, containing the absolute address given by *expr*. You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the *expr* in this context: you may give a hexadecimal constant or the name of an existing symbol, or use + and - to add or subtract hexadecimal constants or symbols.

Note: There should be no white space between *sym*, the equals sign ("=") and *expr*.

8.4.5 --discard-all (-x)

Discard all local symbols.

8.4.6 --discard-locals (-X)

Discard temporary local symbols.

8.4.7 --fill-upper *value*

Set fill value for upper byte of data.

Use *value* as the upper byte (bits 16-23) when encoding data into program memory. This option affects the encoding of sections created with the psv or eedata attribute, and also the data initialization template if the --no-pack-data option is enabled. If this option is not specified, a default value of 0 will be used.

8.4.8 --force-exe-suffix

Force generation of file with .exe suffix.

8.4.9 --force-link

Force linking of objects that may not be compatible.

If a target processor has been specified with the -p, --processor option, the linker will compare it to information contained in the objects combined during the link. If a possible conflict is detected, an error (in the case of a possible instruction set incompatibility) or a warning (in the case of possible register incompatibility) will be reported. Specify this option to override such errors or warnings.

8.4.10 --no-force-link

Do not force linking of objects that may not be compatible. (**This is the default.**)

8.4.11 --isr

Create an interrupt function for unused vectors. (**This is the default.**)

If a function named __DefaultInterrupt is defined by an application, the linker will insert its address into unused slots in the primary and alternate vector tables. If this function is not defined, create a function that consists of a single reset instruction and insert the address of this function.

8.4.12 --no-isr

Don't create an interrupt function for unused vectors.

Do not create a default interrupt function if an application does not provide one. The value of 0 will be inserted into unused slots in the primary and alternate vector tables.

8.4.13 --library *libname* (-l *libname*)

Search for library *libname*.

Add archive file *libname* to the list of files to link. This option may be used any number of times. pic30-ld will search its path-list for occurrences of lib*libname*.a for every *libname* specified. The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol that was undefined in some object that appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again. See the -(option for a way to force the linker to search archives multiple times. You may list the same archive multiple times on the command line.

If the format of the archive file is not recognized, the linker will ignore it. Therefore, a version mismatch between libraries and the linker may result in "undefined symbol" errors.

If file lib*libname*.a is not found, the linker will search for an omf-specific version of the library with name lib*libname*-coff.a or lib*libname*-elf.a.

© 2008 Microchip Technology Inc.

8.4.14 --library-path <*dir*> (-L <*dir*>)

Add <*dir*> to library search path.

Add path <*dir*> to the list of paths that pic30-ld will search for archive libraries and pic30-ld control scripts. You may use this option any number of times. The directories are searched in the order in which they are specified on the command line. All -L options apply to all -l options, regardless of the order in which the options appear. The library paths can also be specified in a link script with the SEARCH_DIR command. Directories specified this way are searched at the point in which the linker script appears in the command line.

8.4.15 --no-keep-memory

Use less memory and more disk I/O.

pic30-ld normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This option tells pic30-ld to instead optimize for memory usage, by rereading the symbol tables as necessary. This may be required if pic30-ld runs out of memory space while linking a large executable.

8.4.16 --noinhibit-exec

Create an output file even if errors occur.

Retain the executable output file whenever it is still usable. Normally, the linker will not produce an output file if it encounters errors during the link process; it exits without writing an output file when it issues any error whatsoever.

8.4.17 -omf=*format*

pic30-ld produces COFF format output binary files by default. Use this option to specify COFF or ELF format explicitly. Alternatively, the environment variable PIC30_OMF may be used to specify object file format for the dsPIC30F language tools.

Note: The input and output file formats must match. The -omf option can be used to specify both input and output file formats.

8.4.18 --output *file* (-o *file*)

Set output file name.

Use *file* as the name for the program produced by pic30-ld; if this option is not specified, the name a.out is used by default.

8.4.19 -p,--processor *PROC*

Specify the target processor (e.g., 30F2010).

Specify a target processor for the link. This information will be used to detect possible incompatibility between objects during the link. See --force-link for more information.

8.4.20 --relocatable (-r, -i, -Ur)

Generate relocatable output.

I.e., generate an output file that can in turn serve as input to pic30-ld. This is often called partial linking. If this option is not specified, an absolute file is produced.

8.4.21 --retain-symbols-file *file*

Keep only symbols listed in *file*.

Retain only the symbols listed in the file *file*, discarding all others. *file* is simply a flat file, with one symbol name per line. This option is especially useful in environments where a large global symbol table is accumulated gradually, to conserve run-time memory. --retain-symbols-file does not discard undefined symbols, or symbols needed for relocations. You may only specify --retain-symbols-file once in the command line. It overrides -s and -S.

8.4.22 --script *file* (-T *file*)

Read linker script.

Read link commands from the file *file*. These commands replace pic30-ld's default link script (rather than adding to it), so *file* must specify everything necessary to describe the target format. If *file* does not exist, pic30-ld looks for it in the directories specified by any preceding -L options. Multiple -T options accumulate.

8.4.23 --smart-io

Merge I/O library functions when possible. **(This is the default.)**

Several I/O functions in the standard C library exist in multiple versions. For example, there are separate output conversion functions for integers, short doubles and long doubles. If this option is enabled, the linker will merge function calls to reduce memory usage whenever possible. Library function merging will not result in a loss of functionality.

8.4.24 --no-smart-io

Don't merge I/O library functions

Do not attempt to conserve memory by merging I/O library function calls. In some instances the use of this option will increase memory usage.

8.4.25 --strip-all (-s)

Strip all symbols.

Omit all symbol information from the output file.

8.4.26 --strip-debug (-S)

Strip debugging symbols.

Omit debugger symbol information (but not all symbols) from the output file.

8.4.27 -Tbss *address*

Set address of .bss section.

Use *address* as the starting address for the bss segment of the output file. *address* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading '0x' usually associated with hexadecimal values.

Normally the address of this section is specified in a linker script.

© 2008 Microchip Technology Inc.

8.4.28 -Tdata *address*

Set address of .data section.

Use *address* as the starting address for the data segment of the output file. *address* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading '0x' usually associated with hexadecimal values.

Normally the address of this section is specified in a linker script.

8.4.29 -Ttext *address*

Set address of .text section.

Use *address* as the starting address for the text segment of the output file. *address* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading '0x' usually associated with hexadecimal values.

Normally the address of this section is specified in a linker script.

8.4.30 --undefined *symbol* (-u *symbol*)

Start with undefined reference to *symbol*.

Force *symbol* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. -u may be repeated with different option arguments to enter additional undefined symbols.

8.4.31 --no-undefined

Allow no undefined symbols.

8.4.32 --wrap *symbol*

Use wrapper functions for *symbol*

Use a wrapper function for symbol. Any undefined reference to symbol will be resolved to __wrap_symbol. Any undefined reference to __real_symbol will be resolved to symbol. This can be used to provide a wrapper for a system function. The wrapper function should be called __wrap_symbol. If it wishes to call the system function, it should call __real_symbol.

Here is a trivial example:

```
void *
__wrap_malloc (int c)
{
  printf ("malloc called with %ld\n", c);
  return __real_malloc (c);
}
```

If you link other code with this file using --wrap malloc, then all calls to malloc will call the function __wrap_malloc instead. The call to __real_malloc in __wrap_malloc will call the real malloc function. You may wish to provide a __real_malloc function as well, so that links without the --wrap option will succeed. If you do this, you should not put the definition of __real_malloc in the same file as __wrap_malloc; if you do, the assembler may resolve the call before the linker has a chance to wrap it to malloc.

8.5 OPTIONS THAT CONTROL RUN-TIME INITIALIZATION

Run-time initialization options are:

- `--data-init`
- `--no-data-init`
- `--handles`
- `--no-handles`
- `--heap size`
- `--pack-data`
- `--no-pack-data`
- `--stack size`

8.5.1 `--data-init`

Support initialized data. **(This is the default.)**

Create a special output section named `.dinit` as a template for the run-time initialization of data. The C start-up module in `libpic30.a` interprets this template and copies initial data values into initialized data sections. Other data sections (such as `.bss`) are cleared before the `main()` function is called. Note that the persistent data section (`.pbss`) is not affected by this option.

8.5.2 `--no-data-init`

Don't support initialized data.

Suppress the template which is normally created to support run-time initialization of data. When this option is specified, the linker will select a shorter form of the C start-up module in `libpic30.a`. If the application includes data sections which require initialization, a warning message will be generated and the initial data values discarded. Storage for the data sections will be allocated as usual.

8.5.3 `--handles`

Support far code pointers. **(This is the default.)**

Create a special output section named `.handle` as a jump table for accessing far code pointers. Entries in the jump table are used only when the address of a code pointer exceeds 16 bits. The jump table must be loaded in the lowest range of program memory (as defined in the linker scripts).

8.5.4 `--no-handles`

Don't support far code pointers.

Suppress the handle jump table which is normally created to access far code pointers. The programmer is responsible for making certain that all code pointers can be reached with a 16 bit address. If this option is specified and the address of a code pointer exceeds 16 bits, an error is reported.

8.5.5 `--heap` *`size`*

Set heap to *`size`* bytes.

Allocate a run-time heap of *`size`* bytes for use by C programs. The heap is allocated from unused data memory. If not enough memory is available, an error is reported.

© 2008 Microchip Technology Inc.

8.5.6 --pack-data

Pack initial data values. **(This is the default.)**

Fill the upper byte of each instruction word in the data initialization template with data. This option conserves program memory and causes the template to appear as random and possibly invalid instructions if viewed in the disassembler.

8.5.7 --no-pack-data

Don't pack initial data values.

Fill the upper byte of each instruction word in the data initialization template with 0x0 or another value specified with `--fill-upper`. This option consumes additional program memory and causes the template to appear as `NOP` instructions if viewed in the disassembler (and will be executed as such by the 16-bit device).

8.5.8 --stack *size*

Set minimum stack to *size* bytes (default=16).

By default, the linker allocates all unused data memory for the run-time stack. Alternatively, the programmer may allocate the stack by declaring two global symbols: `__SP_init` and `__SPLIM_init`. Use this option to ensure that at least a minimum sized stack is available. The actual stack size is reported in the link map output file. If the minimum size is not available, an error is reported.

8.6 OPTIONS THAT CONTROL INFORMATIONAL OUTPUT

Information output options are:

- `--check-sections`
- `--no-check-sections`
- `--help`
- `--no-warn-mismatch`
- `--report-mem`
- `--trace (-t)`
- `--trace-symbol symbol (-y symbol)`
- `-V`
- `--verbose`
- `--version (-v)`
- `--warn-common`
- `--warn-once`
- `--warn-section-align`

8.6.1 --check-sections

Check section addresses for overlaps. **(This is the default.)**

8.6.2 --no-check-sections

Do not check section addresses for overlaps.

8.6.3 --help

Print option help.

Print a summary of the command line options on the standard output and exit.

8.6.4 --no-warn-mismatch

Do not warn about mismatched input files.

Normally `pic30-ld` will give an error if you try to link together input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for different endiannesses. This option tells `pic30-ld` that it should silently permit such possible errors. This option should only be used with care, in cases when you have taken some special action that ensures that the linker errors are inappropriate.

Note: This option does not apply to library files specified with `-l`.

8.6.5 --report-mem

Print a memory usage report.

Print a summary of memory usage to standard output during the link. This report also appears in the link map.

8.6.6 --trace (-t)

Trace file.

Print the names of the input files as `pic30-ld` processes them.

8.6.7 --trace-symbol *symbol* (-y *symbol*)

Trace mentions of *`symbol`*.

Print the name of each linked file in which *`symbol`* appears. This option may be given any number of times. On many systems, it is necessary to prep-end an underscore to the *`symbol`*. This option is useful when you have an undefined symbol in your link but do not know where the reference is coming from.

8.6.8 -V

Print version and other information.

8.6.9 --verbose

Output lots of information during link.

Display the version number for `pic30-ld`. Display the input files that can and cannot be opened. Display the linker script if using a default built-in script.

8.6.10 --version (-v)

Print version information.

© 2008 Microchip Technology Inc.

8.6.11 --warn-common

Warn about duplicate common symbols.

Warn when a common symbol is combined with another common symbol or with a symbol definition. Unix linkers allow this somewhat sloppy practice, but linkers on some other operating systems do not. This option allows you to find potential problems from combining global symbols. Unfortunately, some C libraries use this practice, so you may get some warnings about symbols in the libraries as well as in your programs.

There are three kinds of global symbols, illustrated here by C examples:

```
int i = 1;
```

A definition, which goes in the initialized data section of the output file.

```
extern int i;
```

An undefined reference, which does not allocate space. There must be either a definition or a common symbol for the variable somewhere.

```
int i;
```

A common symbol. If there are only (one or more) common symbols for a variable, it goes in the uninitialized data area of the output file.

The linker merges multiple common symbols for the same variable into a single symbol. If they are of different sizes, it picks the largest size. The linker turns a common symbol into a declaration, if there is a definition of the same variable.

The `--warn-common` option can produce five kinds of warnings. Each warning consists of a pair of lines: the first describes the symbol just encountered, and the second describes the previous symbol encountered with the same name. One or both of the two symbols will be a common symbol.

Turning a common symbol into a reference, because there is already a definition for the symbol.

```
file(section): warning: common of 'symbol' overridden by definition
file(section): warning: defined here
```

Turning a common symbol into a reference, because a later definition for the symbol is encountered. This is the same as the previous case, except that the symbols are encountered in a different order.

```
file(section): warning: definition of 'symbol' overriding common
file(section): warning: common is here
```

Merging a common symbol with a previous same-sized common symbol.

```
file(section): warning: multiple common of 'symbol'
file(section): warning: previous common is here
```

Merging a common symbol with a previous larger common symbol.

```
file(section): warning: common of 'symbol' overridden by larger common
file(section): warning: larger common is here
```

Merging a common symbol with a previous smaller common symbol. This is the same as the previous case, except that the symbols are encountered in a different order.

```
file(section): warning: common of 'symbol' overriding smaller common
file(section): warning: smaller common is here
```

8.6.12 --warn-once

Warn only once per undefined symbol.

Only warn once for each undefined symbol, rather than once per module that refers to it.

8.6.13 --warn-section-align

Warn if start of section changes due to alignment.

Warn if the address of an output section is changed because of alignment. This means a gap has been introduced into the (normally sequential) allocation of memory.

Typically, an input section will set the alignment. The address will only be changed if it is not explicitly specified; that is, if the `SECTIONS` command does not specify a start address for the section.

8.7 OPTIONS THAT MODIFY THE LINK MAP OUTPUT

Link map output modifying options are:

- `--cref`
- `--print-map (-M)`
- `-Map file`

8.7.1 --cref

Output cross reference table.

If a linker map file is being generated, the cross-reference table is printed to the map file. Otherwise, it is printed on the standard output. The format of the table is intentionally simple, so that a script may easily process it if necessary. The symbols are printed out, sorted by name. For each symbol, a list of file names is given. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

8.7.2 --print-map (-M)

Print map file on standard output.

Print a link map to the standard output. A link map provides information about the link, including the following:

Where object files and symbols are mapped into memory.

How common symbols are allocated.

All archive members included in the link, with a mention of the symbol which caused the archive member to be brought in.

8.7.3 -Map *file*

Write a map file.

Print a link map to the file *`file`*. See the description of the `--print-map (-M)` option.

© 2008 Microchip Technology Inc.

8.8 OPTIONS THAT SPECIFY CODEGUARD™ SECURITY FEATURES

Three linker options are related to CodeGuard Security:

- `--boot` *`LIST`* – Specify options for the boot segment
- `--secure` *`LIST`* – Specify options for the secure segment
- `--general` *`LIST`* – Specify options for the general segment

`LIST` may include a single segment option or several segment options separated by colons. Multiple instances of `boot`, `secure`, or `general` options are accepted and will be combined. An optional equals sign (=) may precede *`LIST`*.

8.8.1 CodeGuard Security Segment Options

The following segment options correspond to specific CodeGuard Security settings as described in the CodeGuard Security documentation. The linker will validate that any CodeGuard Security option(s) specified are supported by the target device. An error will be reported if the target device does not support a particular option.

These options will appear in the MPLAB IDE Build Options dialog as a sub-category of the **MPLAB LINK30** tab. They will be passed to the linker via command line. The Configuration Bits display for CodeGuard Security configuration words (FBS, FSS, and FGS) will become read-only.

TABLE 8-1: CODEGUARD™ SECURITY SEGMENT OPTIONS

Option	Segment(s) Supported		
	boot	**secure**	**general**
no_ram **	X	X	
small_ram	X	X	
medium_ram	X	X	
large_ram	X	X	
no_flash **	X	X	
small_flash_std	X	X	
medium_flash_std	X	X	
large_flash_std	X	X	
small_flash_high	X	X	
medium_flash_high	X	X	
large_flash_high	X	X	
no_eeprom **	X	X	
eeprom	X		
small_eeprom		X	
medium_eeprom		X	
large_eeprom		X	
no_write_protect **	X	X	X
write_protect	X	X	X
no_code_protect **			X
code_protect_std			X
code_protect_high			X

** default setting

EXAMPLE 8-1: CODEGUARD SECURITY SEGMENT OPTIONS

```
--boot small_flash_std
--boot=small_ram:medium_flash_std:eeprom

--secure no_ram:small_flash_std
--secure=medium_ram:large_flash_high

--general write_protect
--general=no_write_protect:code_protect_high
```

8.8.2 User-Defined Segment Options

The following segment options are supported for any device. They enable the programmer to take advantage of special language features created for CodeGuard Security, including separately linked application segments and access entry branch tables. These options do not require CodeGuard Security support in hardware and will not be encoded as configuration word settings.

> **Note:** User-defined segment options should not be combined with CodeGuard Security options. They are intended for debugging and/or special boot-loader applications.

TABLE 8-2: USER-DEFINED SEGMENT OPTIONS

Option	Segment(s) Supported		
	boot	**secure**	**general**
ram_size=*nn*	X	X	
flash_size=*nn*	X	X	
nn is a positive integer in decimal or hex format			

EXAMPLE 8-2: USER-DEFINED SEGMENT OPTIONS

```
--boot flash_size=128
--boot=ram_size=64:flash_size=256

--secure flash_size=256
--secure=ram_size=64:flash_size=256
```


© 2008 Microchip Technology Inc.

Chapter 9. Linker Scripts

9.1 INTRODUCTION

Linker scripts are used to control MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB LINK30) functions. You can customize your linker script for specialized control of the linker.

9.2 HIGHLIGHTS

Topics covered in this chapter are:

- Overview of Linker Scripts
- Command Line Information
- Contents of a Linker Script
- Creating a Custom Linker Script
- Linker Script Command Language
- Expressions in Linker Scripts

9.3 OVERVIEW OF LINKER SCRIPTS

Linker scripts control all aspects of the link process, including:

- allocation of data memory and program memory
- mapping of sections from input files into the output file
- construction of special data structures (such as interrupt vector tables)
- assignment of absolute SFR addresses for the target device

Linker scripts are text files that contain a series of commands. Each command is either a keyword, possibly followed by arguments, or an assignment to a symbol. Comments may be included just as in C, delimited by `/*` and `*/`. As in C, comments are syntactically equivalent to white space. Unlike C, white space is significant and is often not permitted between syntax elements.

The 16-bit Language Tools include a set of standard linker scripts: device-specific linker scripts (e.g., `p30f3014.gld`) and one generic linker script (`p30sim.gld`). If you will be using the MPLAB ICE 4000 emulator, you will need to choose the "e" version of the device linker script (e.g., `p30f3014e.gld`) so that XY data will be allocated properly for this tool.

9.4 COMMAND LINE INFORMATION

Linker scripts are specified on the command line using either the `-T` option or the `--script` option (see **Section 8.4 "Options that Control Output File Creation"**):

```
pic30-ld -o output.cof output.o --script ..\support\gld\p30f3014.gld
```

If the linker is invoked through `pic30-gcc`, add the `-Wl`, prefix to allow the option to be passed to the linker:

```
pic30-gcc -o output.cof output.s -Wl,--script,
  ..\support\gld\p30f3014.gld
```

If no linker script is specified, the linker will use an internal version known as the default linker script. The default linker script has memory range information and SFR definitions that are appropriate for `sim30`, the command line simulator. The default linker script can be examined by invoking the linker with the `--verbose` option:

```
pic30-ld --verbose
```

Note: The default linker script is functionally equivalent to the generic linker script `p30sim.gld`.

9.5 CONTENTS OF A LINKER SCRIPT

In the next several sections, a device-specific linker script for the dsPIC30F3014 will be examined. The linker script contains the following categories of information:

- Processor and Entry Points
- Memory Region Information
- Base Memory Addresses
- Input/Output Section Map
- Interrupt Vector Tables
- SFR Addresses

9.5.1 Processor and Entry Points

The first several lines of a linker script define the processor and entry points:

```
/*
** Linker Script for p30f3014
*/
OUTPUT_ARCH("30f3014")
EXTERN(__resetPRI)
EXTERN(__resetALT)
OPTIONAL(-lp30F3014)
```

The `OUTPUT_ARCH` command specifies the target processor. The `EXTERN` commands force two C run-time start-up modules to be loaded from archives. The linker will select one and discard the other, based on the `--data-init` option. The `OPTIONAL` command specifies a device-specific library that should be opened if available. If the library file cannot be found, the link will continue without error unless there are unresolved references in the application.

© 2008 Microchip Technology Inc.

9.5.2 Memory Region Information

The next section of a linker script defines the various memory regions for the target device using the `MEMORY` command.

For the dsPIC30F3014, several memory regions are defined:

```
/*
** Memory Regions
*/
MEMORY
{
  data          : ORIGIN = 0x800,    LENGTH = 2048
  program       : ORIGIN = 0x100,    LENGTH = ((8K * 2) - 0x100)
  reset         : ORIGIN = 0,        LENGTH = (4)
  ivt           : ORIGIN = 0x04,     LENGTH = (62 * 2)
  aivt          : ORIGIN = 0x84,     LENGTH = (62 * 2)
  __FOSC        : ORIGIN = 0xF80000, LENGTH = (2)
  __FWDT        : ORIGIN = 0xF80002, LENGTH = (2)
  __FBORPOR     : ORIGIN = 0xF80004, LENGTH = (2)
  __CONFIG4     : ORIGIN = 0xF80006, LENGTH = (2)
  __CONFIG5     : ORIGIN = 0xF80008, LENGTH = (2)
  __FGS         : ORIGIN = 0xF8000A, LENGTH = (2)
  __FUID0       : ORIGIN = 0x8005C0, LENGTH = (2)
  __FUID1       : ORIGIN = 0x8005C2, LENGTH = (2)
  __FUID2       : ORIGIN = 0x8005C4, LENGTH = (2)
  __FUID3       : ORIGIN = 0x8005C6, LENGTH = (2)
  eedata        : ORIGIN = 0x7FFC00, LENGTH = (1024)
}
```

Each memory region is range-checked as sections are added during the link process. If any region overflows, a link error is reported.

`MEMORY` regions are:

- Data Region
- Program Region
- Reset, Ivt and Aivt Regions
- Fuse Configuration Regions
- Unit ID Regions
- EEDATA Memory Region

9.5.2.1 DATA REGION

```
data          : ORIGIN = 0x800,     LENGTH = 2048
```

The data region corresponds to the RAM memory of the dsPIC30F3014 device, and is used for both initialized and uninitialized variables. The starting address of region `data` is `0x800`. This is the first usable location in RAM, after the space reserved for memory-mapped Special Function Registers (SFRs).

9.5.2.2 PROGRAM REGION

```
program      : ORIGIN = 0x100, LENGTH = ((8K * 2) - 0x100)
```

The program region corresponds to the Flash memory of the dsPIC30F3014 device that is available for user code, library code and constants. The starting address of region program is `0x100`. This is the first location in Flash that is available for general use. Addresses below `0x100` are reserved for the reset instruction and the two vector tables.

The length specification of the program region deserves particular emphasis. The (8K * 2) portion indicates that the dsPIC30F3014 has 8K instruction words of Flash memory, and that each instruction word is 2 address units wide. The `- 0x100` portion reflects the fact that some of the Flash is reserved for the reset instruction and vector tables.

> **Note:** Instruction words in the 16-bit devices are 24 bits, or 3 bytes, wide. However the program counter increments by 2 for each instruction word for compatibility with data memory. Address and lengths in program memory are expressed in program counter units.

9.5.2.3 RESET, IVT AND AIVT REGIONS

```
reset        : ORIGIN = 0,         LENGTH = (4)
```

The reset region corresponds to the 16-bit reset instruction at address `0` in program memory. The reset region is 4 address units, or 2 instruction words, long. This region always contains a `GOTO` instruction that is executed upon device reset. The `GOTO` instruction is encoded by data commands in the section map (see **Section 9.5.4.1 "Output Section .reset"**).

```
ivt          : ORIGIN = 0x04,      LENGTH = (62 * 2)
aivt         : ORIGIN = 0x84,      LENGTH = (62 * 2)
```

The `ivt` and `aivt` regions correspond to the interrupt vector table and alternate interrupt vector table, respectively. Each interrupt vector table contains 62 entries, each 2 address units in length. Each entry represents a word of program memory, which contains a 24-bit address. The linker initializes the vector tables with appropriate data, according to standard naming conventions.

Regions `reset`, `ivt` and `aivt` comprise the low address portion of Flash memory that is not available for user programs.

9.5.2.4 FUSE CONFIGURATION REGIONS

```
__FOSC       : ORIGIN = 0xF80000, LENGTH = (2)
__FWDT       : ORIGIN = 0xF80002, LENGTH = (2)
__FBORPOR    : ORIGIN = 0xF80004, LENGTH = (2)
__CONFIG4    : ORIGIN = 0xF80006, LENGTH = (2)
__CONFIG5    : ORIGIN = 0xF80008, LENGTH = (2)
__FGS        : ORIGIN = 0xF8000A, LENGTH = (2)
```

These regions correspond to the dsPIC30F3014 configuration registers.

Each fuse configuration region is exactly one instruction word long. If sections are defined in the application source code with the standard naming convention, the section contents will be written into the appropriate configuration register(s). Otherwise the registers are left uninitialized. If more than one value is defined for any configuration region, a link error will be reported.

© 2008 Microchip Technology Inc.

9.5.2.5 UNIT ID REGIONS

```
__FUID0      : ORIGIN = 0x8005C0, LENGTH = (2)
__FUID1      : ORIGIN = 0x8005C2, LENGTH = (2)
__FUID2      : ORIGIN = 0x8005C4, LENGTH = (2)
__FUID3      : ORIGIN = 0x8005C6, LENGTH = (2)
```

The unit ID regions correspond to locations in program memory that may be programmed with application-specific information.

9.5.2.6 EEDATA MEMORY REGION

```
eedata       : ORIGIN = 0x7FFC00,  LENGTH = (1024)
```

The `eedata` region corresponds to non-volatile data flash memory located in high memory. Although located in program memory space, the data flash is organized like data memory. The total length is 1024 bytes.

9.5.3 Base Memory Addresses

This portion of the linker script defines the base addresses of several output sections in the application. Each base address is defined as a symbol with the following syntax:

```
name = value;
```

The symbols are used to specify load addresses in the section map. For the dsPIC30F3014, several base memory addresses are defined:

```
/*
** Base Memory Addresses - Program Memory
*/
__RESET_BASE  = 0;        /* Reset Instruction               */
__IVT_BASE    = 0x04;     /* Interrupt Vector Table          */
__AIVT_BASE   = 0x84;     /* Alternate Interrupt Vector Table */
__CODE_BASE   = 0x100;    /* Handles, User Code, Library Code */


/*
** Base Memory Addresses - Data Memory
*/
__SFR_BASE    = 0;        /* Memory-mapped SFRs                */
__DATA_BASE   = 0x800;    /* X and General Purpose Data Memory  */
__YDATA_BASE  = 0x0C00;   /* Y Data Memory for DSP Instructions */
```

9.5.4 Input/Output Section Map

The section map is the heart of the linker script. It defines how input sections are mapped to output sections. Note that input sections are portions of an application that are defined in source code, while output sections are created by the linker. Generally, several input sections may be combined into a single output section.

For example, suppose that an application is comprised of five different functions, and each function is defined in a separate source file. Together, these source files will produce five input sections. The linker will combine these input sections into a single output section. Only the output section has an absolute address. Input sections are always relocatable.

If any input or output sections are empty, there is no penalty or storage cost for the linked application. Most applications will use only a few of the many sections that appear in the section map.

- Output Section .reset
- Output Section .text
- User-Defined Section in Program Memory

- Output Sections in Configuration Memory
- User-Defined Section in Data Flash Memory
- MPLAB ICD 2 Debugger Memory
- User-defined Section in Data Memory

9.5.4.1 OUTPUT SECTION .RESET

Section .reset contains a GOTO instruction, created at link time, from output section data commands:

```
/*
** Reset Instruction
*/
.reset __RESET_BASE :
  {
        SHORT(ABSOLUTE(__reset));
        SHORT(0x04);
        SHORT((ABSOLUTE(__reset) >> 16) & 0x7F);
        SHORT(0);
  } >reset
```

Each SHORT() data command causes a 2 byte value to be included. There are two expressions which include the symbol __reset, which by convention is the first function invoked after a device reset. Each expression calculates a portion of the address of the reset function. These declarations encode a 16-bit GOTO instruction, which is two instruction words long.

The ABSOLUTE() function specifies the final value of a program symbol after linking. If this function were omitted, a relative (before-linking) value of the program symbol would be used.

The >reset portion of this definition indicates that this section should be allocated in the reset memory region.

© 2008 Microchip Technology Inc.

9.5.4.2 OUTPUT SECTION .TEXT

Section `.text` collects executable code from all of the application's input files.

```
/*
** User Code and Library Code
*/
.text :
  {
        *(.init);
        *(.user_init);
        *(.handle);
        *(.libc) *(.libm) *(.libdsp); /* keep together in this order */
        *(.lib*);
  } >program
```

Several different input sections are collected into one output section. This was done to ensure the order in which the input sections are loaded.

Input section `.init` contains the startup code that is executed immediately after device reset. It is positioned first so that its address may be readily available.

The input section `.handle` is used for function pointers and is loaded first at low addresses.

The library sections `.libc`, `.libm` and `.libdsp` must be grouped together to ensure locality of reference.

The wildcard pattern `.lib*` then collects other libraries such as the peripheral libraries (which are allocated in section `.libperi`).

Previously, input section `.text` was also included, which contained application code. It is no longer explicitly mapped so that the linker may distribute the code around psv sections as needed in order to satisfy psv address alignment requirements.

9.5.4.3 USER-DEFINED SECTION IN PROGRAM MEMORY

A stub is included for user-defined output sections in program memory. This stub may be edited as needed to support the application requirements. Once a standard linker script has been modified, it is called a "custom linker script." In practice, it is often simpler to use section attributes in source code to locate user-defined sections in program memory. See **Chapter 11. "Linker Examples"** for more information.

```
/*
** User-Defined Section in Program Memory
**
** note: can specify an address using
**       the following syntax:
**
**       usercode 0x1234 :
**         {
**           *(usercode);
**         } >program
*/
usercode :
  {
        *(usercode);
  } >program
```

An exact, absolute starting address can be specified, if necessary. If the address is greater than the current location counter, the intervening memory space will be skipped and filled with zeros. If the address is less than the current location counter, a section overlap will occur. Whenever two output sections occupy the same address range, a link error will be reported. Overlapping sections in program memory can not be supported.

Note:	Each memory region has its own location counter.

9.5.4.4 OUTPUT SECTIONS IN CONFIGURATION MEMORY

Several sections are defined that match the Fuse Configuration memory regions:

```
/*
** Configuration Fuses
*/
__FOSC :
  { *(__FOSC.sec)     } >__FOSC
__FWDT :
  { *(__FWDT.sec)     } >__FWDT
__FBORPOR :
  { *(__FBORPOR.sec) } >__FBORPOR
__CONFIG4 :
  { *(__CONFIG4.sec) } >__CONFIG4
__CONFIG5 :
  { *(__CONFIG5.sec) } >__CONFIG5
__FGS :
  { *(__FGS.sec)      } >__FGS
__FICD :
  { *(__FICD.sec)     } >__FICD
__FUID0 :
  { *(__FUID0.sec)    } >__FUID0
__FUID1 :
  { *(__FUID1.sec)    } >__FUID1
__FUID2 :
  { *(__FUID2.sec)    } >__FUID2
__FUID3 :
  { *(__FUID3.sec)    } >__FUID3
```


© 2008 Microchip Technology Inc.

The Configuration Fuse sections are supported by macros defined in the 16-bit device-specific include files in `support/inc` and the C header files in `support/h`.

For example, to disable the Watchdog Timer in assembly language:

```
.include "p30f6014.inc"
config __FWDT, WDT_OFF
```

The equivalent operation in C would be:

```
#include "p30f6014.h"
_FWDT(WDT_OFF);
```

Configuration macros have the effect of changing the current section. In C, the macro should be used outside of any function. In assembly language, the macro should be followed by a `.section` directive.

9.5.4.5 USER-DEFINED SECTION IN DATA FLASH MEMORY

A stub is included for user-defined output sections in EEData memory. This stub may be edited as needed to support the application requirements. Once a standard linker script has been modified, it is called a "custom linker script." In practice, it is often simpler to use section attributes in source code to locate user-defined sections in data flash memory. See **Chapter 11. "Linker Examples"** for more information.

```
/*
** User-Defined Section in Data Flash Memory
**
** note: can specify an address using
**       the following syntax:
**
**       eedata 0x7FF100 :
**         {
**           *(eedata);
**         } >eedata
*/
eedata :
{
      *(eedata);
} >eedata
```

An exact, absolute starting address can be specified if necessary. If the address is greater than the current location counter, the intervening memory will be skipped and filled with zeros. If the address is less than the current location counter, a section overlap will occur. Whenever two output sections occupy the same address range, a link error will reported. Overlapping sections in eedata memory can not be supported.

Note:	Each memory region has its own location counter.

9.5.4.6 MPLAB ICD 2 DEBUGGER MEMORY

The MPLAB ICD 2 debugger requires a portion of data memory for its variables and stack. Since the debugger is linked separately and in advance of user applications, the block of memory must be located at a fixed address and dedicated for use by MPLAB ICD 2.

```
/*
** ICD Debug Exec
**
** This section provides optional storage for
** the ICD2 debugger. Define a global symbol
** named __ICD2RAM to enable ICD2. This section
** must be loaded at data address 0x800.
*/
.icd __DATA_BASE (NOLOAD):
  {
    . += (DEFINED (__ICD2RAM) ? 0x50 : 0 );
  } > data
```

Section `.icd` is designed to optionally reserve memory for MPLAB ICD 2. If global symbol `__ICD2RAM` is defined at link time, `0x50` bytes of memory at address `0x800` will be reserved. The (`NOLOAD`) attribute indicates that no initial values need to be loaded for this section.

9.5.4.7 USER-DEFINED SECTION IN DATA MEMORY

A stub is included for user-defined output sections in data memory. This stub may be edited as needed to support the application requirements. Once a standard linker script has been modified, it is called a "custom linker script." In practice, it is often simpler to use section attributes in source code to locate user-defined sections in data memory. See **Chapter 11. "Linker Examples"** for more information.

```
/*
** User-Defined Section in Data Memory
**
** note: can specify an address using
**       the following syntax:
**
**       userdata 0x1234 :
**         {
**           *(userdata);
**         } >data
*/
userdata :
  {
        *(userdata);
  } >data
```

An exact, absolute starting address can be specified, if necessary. If the address is greater than the current location counter, the intervening memory space will be skipped and filled with zeros. If the address is less than the current location counter, a section overlap will occur. Whenever two output sections occupy the same address range, a link error will be reported. Overlapping sections in data memory cannot be supported.

© 2008 Microchip Technology Inc.

9.5.5 Interrupt Vector Tables

The primary and alternate interrupt vector tables are defined in a second section map, near the end of the standard linker script:

```
/*
** Section Map for Interrupt Vector Tables
*/
SECTIONS
{

/*
** Primary Interrupt Vector Table
*/
.ivt __IVT_BASE :
  {
    LONG(DEFINED(__ReservedTrap0)  ? ABSOLUTE(__ReservedTrap0)  :
         ABSOLUTE(__DefaultInterrupt));
    LONG(DEFINED(__OscillatorFail) ? ABSOLUTE(__OscillatorFail) :
         ABSOLUTE(__DefaultInterrupt));
    LONG(DEFINED(__AddressError)   ? ABSOLUTE(__AddressError)   :
         ABSOLUTE(__DefaultInterrupt));
    :
    :
    LONG(DEFINED(__Interrupt53)    ? ABSOLUTE(__Interrupt53)    :
         ABSOLUTE(__DefaultInterrupt));
  } >ivt
```

The vector table is defined as a series of `LONG()` data commands. Each vector table entry is 4 bytes in length (3 bytes for a program memory address plus an unused phantom byte). The data commands include an expression using the `DEFINED()` function and the ? operator. A typical entry may be interpreted as follows:

If symbol "__OscillatorFail" is defined, insert the absolute address of that symbol. Otherwise, insert the absolute address of symbol "__DefaultInterrupt".

By convention, a function that will be installed as the second interrupt vector should have the name `__OscillatorFail`. If such a function is included in the link, its address is loaded into the entry. If the function is not included, the address of the default interrupt handler is loaded instead. If the application has not provided a default interrupt handler (i.e., a function with the name `__DefaultInterrupt`), the linker will generate one automatically. The simplest default interrupt handler is a reset instruction.

Note: The programmer must insure that functions installed in interrupt vector tables conform to the architectural requirements of interrupt service routines.

The contents of the alternate interrupt vector table are defined as follows:

```
/*
** Alternate Interrupt Vector Table
*/
.aivt __AIVT_BASE :
  {
    LONG(DEFINED(__AltReservedTrap0)  ? ABSOLUTE(__AltReservedTrap0)
:
        (DEFINED(__ReservedTrap0)     ? ABSOLUTE(__ReservedTrap0)
:
         ABSOLUTE(__DefaultInterrupt)));
    LONG(DEFINED(__AltOscillatorFail) ? ABSOLUTE(__AltOscillatorFail)
:
        (DEFINED(__OscillatorFail)    ? ABSOLUTE(__OscillatorFail)
:
         ABSOLUTE(__DefaultInterrupt)));
    LONG(DEFINED(__AltAddressError)   ? ABSOLUTE(__AltAddressError)
:
        (DEFINED(__AddressError)      ? ABSOLUTE(__AddressError)
:
         ABSOLUTE(__DefaultInterrupt)));
    :
    :
    LONG(DEFINED(__AltInterrupt53)    ? ABSOLUTE(__AltInterrupt53)
:
        (DEFINED(__Interrupt53)       ? ABSOLUTE(__Interrupt53)
:
         ABSOLUTE(__DefaultInterrupt)));
  } >aivt
```

The syntax of the alternate interrupt vector table is similar to the primary, except for an additional expression that causes each alternate table entry to default to the corresponding primary table entry.

9.5.6 SFR Addresses

Absolute addresses for the Special Function Registers (SFRs) are defined as a series of symbol definitions:

```
**====================================================================
=
**
**       dsPIC Core Register Definitions
**
**====================================================================*
/
 WREG0 = 0x0000;
_WREG0 = 0x0000;
 WREG1 = 0x0002;
_WREG1 = 0x0002;
:
:
```

Note: If identifiers in a C or assembly program are defined with the same names as SFRs, multiple definition linker errors will result.

Two versions of each SFR address are included, with and without a leading underscore. This is to enable both C and assembly language programmers to refer to the SFR using the same name. By convention, the C compiler adds a leading underscore to every identifier.

© 2008 Microchip Technology Inc.

9.6 CREATING A CUSTOM LINKER SCRIPT

The standard 16-bit linker scripts are general purpose and will satisfy the demands of most applications. However, occasions may arise where a custom linker script is required.

To create a custom linker script, start with a copy of the standard linker script that is appropriate for the target device. For example, to customize a linker script for the dsPIC30F3014 device, start with a copy of `p30f3014.gld`.

Customizing a standard linker script will usually involve editing sections or commands that are already present. For example, stubs for user-defined sections in both data memory and program memory are included. These stubs may be renamed and/or customized with absolute addresses if required.

It is recommended that unused sections be retained in a custom linker script, since unused sections will not impact application memory usage. If a section must be removed for a custom script, C style comments can be used to disable it.

9.7 LINKER SCRIPT COMMAND LANGUAGE

Linker scripts are text files that contain a series of commands. Each command is either a keyword, possibly followed by arguments, or an assignment to a symbol. Multiple commands may be separated using semicolons. White space is generally ignored.

Strings such as file or format names can normally be entered directly. If the file name contains a character such as a comma which would otherwise serve to separate file names, the file name may be specified in double quotes. There is no way to use a double quote character in a file name.

Comments may be included just as in C, delimited by `/*` and `*/`. As in C, comments are syntactically equivalent to white space.

- Basic Linker Script Concepts
- Commands Dealing with Files
- Assigning Values to Symbols
- MEMORY Command
- SECTIONS Command
- Other Linker Script Commands

9.7.1 Basic Linker Script Concepts

The linker combines input files into a single output file. The output file and each input file are in a special data format known as an object file format. Each file is called an object file. Each object file has, among other things, a list of sections. A section in an input file is called an input section; similarly, a section in the output file is an output section.

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the section contents. A section may be marked as loadable, which mean that the contents should be loaded into memory when the output file is run. A section with no contents may be allocatable, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out).

Every loadable or allocatable output section has two addresses. The first is the VMA, or virtual memory address. This is the address the section will have when the output file is run. The second is the LMA, or load memory address. This is the address at which the section will be loaded. In most cases, the two addresses will be the same. An example of when they might be different is when a section is intended for use in the Program Space Visibility (PSV) window. In this case, the program memory address would be the LMA, and the data memory address would be the VMA.

The sections in an object file can be viewed by using the `pic30-objdump` program with the `-h` option.

Every object file also has a list of symbols, known as the symbol table. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If a C or C++ program is compiled into an object file, a defined symbol will be created for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

Symbols in an object file can be viewed by using the `pic30-nm` program, or by using the `pic30-objdump` program with the `-t` option.

9.7.2 Commands Dealing with Files

Several linker script commands deal with files.

`INCLUDE filename`

Include the linker script filename at this point. The file will be searched for in the current directory, and in any directory specified with the `-L` option. Calls to `INCLUDE` may be nested up to 10 levels deep.

```
INPUT(file, file, ...)
INPUT(file file ...)
```

The `INPUT` command directs the linker to include the named files in the link, as though they were named on the command line. The linker will first try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of `-L` in **Section 8.4.14 "`--library-path <dir> (-L <dir>)`"**.

If `INPUT (-lfile)` is used, `pic30-ld` will transform the name to `libfile.a`, as with the command line argument `-l`.

When the `INPUT` command appears in an implicit linker script, the files will be included in the link at the point at which the linker script file is included. This can affect archive searching.

```
GROUP(file, file, ...)
GROUP(file file ...)
```

The `GROUP` command is like `INPUT`, except that the named files should all be archives, and they are searched repeatedly until no new undefined references are created. See the description of `-(` in **Section 8.4.2 "`-( archives -), --start-group archives, --end-group`"**.

```
OPTIONAL(file, file, ...)
OPTIONAL(file file ...)
```

The `OPTIONAL` command is analogous to the `INPUT` command, except that the named files are not required for the link to succeed. This is particularly useful for specifying archives (or libraries) that may or may not be installed with the compiler.

© 2008 Microchip Technology Inc.

OUTPUT(filename)

The OUTPUT command names the output file. Using OUTPUT(filename) in the linker script is exactly like using -o filename on the command line (see **Section 8.4.18 "--output file (-o file)"**). If both are used, the command line option takes precedence.

SEARCH_DIR(path)

The SEARCH_DIR command adds path to the list of paths where the linker looks for archive libraries. Using SEARCH_DIR(path) is exactly like using -L path on the command line (see **Section 8.4.14 "--library-path <dir> (-L <dir>)"**). If both are used, then the linker will search both paths. Paths specified using the command line option are searched first.

STARTUP(filename)

The STARTUP command is just like the INPUT command, except that filename will become the first input file to be linked, as though it were specified first on the command line.

9.7.3 Assigning Values to Symbols

A value may be assigned to a symbol in a linker script. This will define the symbol as a global symbol.

- Simple Assignments
- PROVIDE

9.7.3.1 SIMPLE ASSIGNMENTS

A symbol may be assigned using any of the C assignment operators:

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

The first case will define symbol to the value of expression. In the other cases, symbol must already be defined, and the value will be adjusted accordingly.

The special symbol name '.' indicates the location counter. This symbol may only be used within a SECTIONS command.

The semicolon after expression is required.

Expressions are defined in **Section 9.8 "Expressions in Linker Scripts"**.

Symbol assignments may appear as commands in their own right, or as statements within a SECTIONS command, or as part of an output section description in a SECTIONS command.

The section of the symbol will be set from the section of the expression; for more information, see **Section 9.8.6 "The Section of an Expression"**.

Here is an example showing the three different places that symbol assignments may be used:

```
   floating_point = 0;
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
    }
  _bdata = (. + 3) & ~ 4;
  .data : { *(.data) }
}
```

In this example, the symbol `floating_point` will be defined as zero. The symbol `_etext` will be defined as the address following the last `.text` input section. The symbol `_bdata` will be defined as the address following the `.text` output section aligned upward to a 4-byte boundary.

9.7.3.2 PROVIDE

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and is not defined by any object included in the link. For example, traditional linkers defined the symbol `etext`. However, ANSI C requires that `etext` may be used as a function name without encountering an error. The `PROVIDE` keyword may be used to define a symbol, such as `etext`, only if it is referenced but not defined. The syntax is `PROVIDE(symbol = expression)`.

Here is an example of using `PROVIDE` to define `etext`:

```
   SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
      PROVIDE(etext = .);
    }
}
```

In this example, if the program defines `_etext` (with a leading underscore), the linker will give a multiple definition error. If, on the other hand, the program defines `etext` (with no leading underscore), the linker will silently use the definition in the program. If the program references `etext` but does not define it, the linker will use the definition in the linker script.

9.7.4 MEMORY Command

The linker's default configuration permits allocation of all available memory. This can be overridden by using the `MEMORY` command.

The `MEMORY` command describes the location and size of blocks of memory in the target. It can be used to describe which memory regions may be used by the linker and which memory regions it must avoid. Sections may then be assigned to particular memory regions. The linker will set section addresses based on the memory regions and will warn about regions that become too full. The linker will not shuffle sections around to fit into the available regions.

© 2008 Microchip Technology Inc.

The syntax of the MEMORY command is:

```
 MEMORY
{
  name [(attr)] : ORIGIN = origin, LENGTH = len
  ...
}
```

The name is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names or section names. Each memory region must have a distinct name.

The *attr* string is an optional list of attributes that specify whether to use a particular memory region for an input section which is not explicitly mapped in the linker script. As described in **Section 9.7.5 "SECTIONS Command"**, if an output section is not specified for some input section, the linker will create an output section with the same name as the input section. If region attributes are defined, the linker will use them to select the memory region for the output section that it creates.

The *attr* string must consist only of the following characters:

R	Read-only section
W	Read/write section
X	Executable section
A	Allocatable section
I	Initialized section
L	Same as I
!	Invert the sense of any of the following attributes

If an unmapped section matches any of the listed attributes other than !, it will be placed in the memory region. The ! attribute reverses this test, so that an unmapped section will be placed in the memory region only if it does not match any of the listed attributes.

The origin is an expression for the start address of the memory region. The expression must evaluate to a constant before memory allocation is performed, which means that section relative symbols may not be used. The keyword ORIGIN may be abbreviated to org or o (but not, for example, ORG).

The len is an expression for the size in bytes of the memory region. As with the origin expression, the expression must evaluate to a constant before memory allocation is performed. The keyword LENGTH may be abbreviated to len or l.

In the following example, we specify that there are two memory regions available for allocation: one starting at 0 for 48 kilobytes, and the other starting at 0x800 for two kilobytes. The linker will place into the rom memory region every section which is not explicitly mapped into a memory region, and is either read-only or executable. The linker will place other sections which are not explicitly mapped into a memory region into the ram memory region.

```
 MEMORY
{
  rom (rx)  : ORIGIN = 0, LENGTH = 48K
  ram (!rx) : org = 0x800, l = 2K
}
```

Once a memory region is defined, the linker can be directed to place specific output sections into that memory region by using the >region output section attribute. For example, to specify a memory region named mem, use >mem in the output section definition. If no address was specified for the output section, the linker will set the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker will issue an error message.

9.7.5 SECTIONS Command

The SECTIONS command tells the linker how to map input sections into output sections and how to place the output sections in memory.

The format of the SECTIONS command is:

```
   SECTIONS
{
  sections-command
  sections-command
  ...
}
```

Each SECTIONS command may be one of the following:

- an ENTRY command (see **Section 9.7.6 "Other Linker Script Commands"**)
- a symbol assignment (see **Section 9.7.3 "Assigning Values to Symbols"**)
- an output section description
- an overlay description

The ENTRY command and symbol assignments are permitted inside the SECTIONS command for convenience in using the location counter in those commands. This can also make the linker script easier to understand because those commands can be used at meaningful points in the layout of the output file.

Output section descriptions and overlay descriptions are described below.

If a SECTIONS command does not appear in the linker script, the linker will place each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file. The first section will be at address zero.

- Input Section Description
- Input Section Wildcard Patterns
- Input Section for Common Symbols
- Input Section Example
- Output Section Description
- Output Section Address
- Output Section Data
- Output Section Discarding
- Output Section Attributes
- Output Section LMA
- Output Section Region
- Output Section Fill
- Overlay Description

© 2008 Microchip Technology Inc.

9.7.5.1 INPUT SECTION DESCRIPTION

The most common output section command is an input section description.

The input section description is the most basic linker script operation. Output sections tell the linker how to lay out the program in memory. Input section descriptions tell the linker how to map the input files into the memory layout.

An input section description consists of a file name optionally followed by a list of section names in parentheses.

The file name and the section name may be wildcard patterns, which are described further below.

The most common input section description is to include all input sections with a particular name in the output section. For example, to include all input `.text` sections, one would write:

```
*(.text)
```

Here the `*` is a wildcard which matches any file name. To exclude a list of files from matching the file name wildcard, `EXCLUDE_FILE` may be used to match all files except the ones specified in the `EXCLUDE_FILE` list. For example:

```
*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
```

will cause all `.ctors` sections from all files except `crtend.o` and `otherfile.o` to be included.

There are two ways to include more than one section:

```
*(.text .rdata)
*(.text) *(.rdata)
```

The difference between these is the order in which the `.text` and `.rdata` input sections will appear in the output section. In the first example, they will be intermingled. In the second example, all `.text` input sections will appear first, followed by all `.rdata` input sections.

A file name can be specified to include sections from a particular file. This would be useful if one of the files contain special data that needs to be at a particular location in memory. For example:

```
data.o(.data)
```

If a file name is specified without a list of sections, then all sections in the input file will be included in the output section. This is not commonly done, but it may be useful on occasion. For example:

```
data.o
```

When a file name is specified which does not contain any wild card characters, the linker will first see if the file name was also specified on the linker command line or in an `INPUT` command. If not, the linker will attempt to open the file as an input file, as though it appeared on the command line. This differs from an `INPUT` command because the linker will not search for the file in the archive search path.

9.7.5.2 INPUT SECTION WILDCARD PATTERNS

In an input section description, either the file name or the section name or both may be wildcard patterns.

The file name of * seen in many examples is a simple wildcard pattern for the file name.

The wildcard patterns are like those used by the UNIX shell.

*	matches any number of characters
?	matches any single character
[*chars*]	matches a single instance of any of the *chars*; the - character may be used to specify a range of characters, as in [a-z] to match any lower case letter
\	quotes the following character

When a file name is matched with a wildcard, the wildcard characters will not match a / character (used to separate directory names on UNIX). A pattern consisting of a single * character is an exception; it will always match any file name, whether it contains a / or not. In a section name, the wildcard characters will match a / character.

File name wildcard patterns only match files which are explicitly specified on the command line or in an INPUT command. The linker does not search directories to expand wild cards.

If a file name matches more than one wildcard pattern, or if a file name appears explicitly and is also matched by a wildcard pattern, the linker will use the first match in the linker script. For example, this sequence of input section descriptions is probably in error, because the data.o rule will not be used:

```
.data  : { *(.data) }
.data1 : { data.o(.data) }
```

Normally, the linker will place files and sections matched by wild cards in the order in which they are seen during the link. This can be changed by using the SORT keyword, which appears before a wildcard pattern in parentheses (e.g., SORT(.text*)). When the SORT keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.

To verify where the input sections are going, use the -M linker option to generate a map file. The map file shows precisely how input sections are mapped to output sections.

This example shows how wildcard patterns might be used to partition files. This linker script directs the linker to place all .text sections in .text and all .bss sections in .bss. The linker will place the .data section from all files beginning with an upper case character in .DATA; for all other files, the linker will place the .data section in .data.

```
   SECTIONS {
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```


© 2008 Microchip Technology Inc.

9.7.5.3 INPUT SECTION FOR COMMON SYMBOLS

A special notation is needed for common symbols, because common symbols do not have a particular input section. The linker treats common symbols as though they are in an input section named COMMON.

File names may be used with the COMMON section just as with any other input sections. This will place common symbols from a particular input file in one section, while common symbols from other input files are placed in another section.

In most cases, common symbols in input files will be placed in the .bss section in the output file. For example:

```
.bss { *(.bss) *(COMMON) }
```

If not otherwise specified, common symbols will be assigned to section .bss.

9.7.5.4 INPUT SECTION EXAMPLE

The following example is a complete linker script. It tells the linker to read all of the sections from file all.o and place them at the start of output section outputa which starts at location 0x10000. All of section .input1 from file foo.o follows immediately, in the same output section. All of section .input2 from foo.o goes into output section outputb, followed by section .input1 from foo1.o. All of the remaining .input1 and .input2 sections from any files are written to output section outputc.

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
}
```

9.7.5.5 OUTPUT SECTION DESCRIPTION

The full description of an output section looks like this:

```
   name [address] [(type)] : [AT(lma)]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [=fillexp]
```

Most output sections do not use most of the optional section attributes.

The white space around *name* and *address* is required. The colon and the curly braces are also required. The line breaks and other white space are optional.

A section name may consist of any sequence of characters, but a name which contains any unusual characters such as commas must be quoted.

Each output-section-command may be one of the following:

- a symbol assignment (see **Section 9.7.3 "Assigning Values to Symbols"**)
- an input section description (see **Section 9.7.5.1 "Input Section Description"**)
- data values to include directly (see **Section 9.7.5.7 "Output Section Data"**)

9.7.5.6 OUTPUT SECTION ADDRESS

The *address* is an expression for the VMA (the virtual memory address) of the output section. If address is not provided, the linker will set it based on region if present, or otherwise based on the current value of the location counter.

If *address* is provided, the address of the output section will be set to precisely that. If neither *address* nor *region* is provided, then the address of the output section will be set to the current value of the location counter aligned to the alignment requirements of the output section. The alignment requirement of the output section is the strictest alignment of any input section contained within the output section.

For example,

```
.text . : { *(.text) }
```

and

```
.text : { *(.text) }
```

are subtly different. The first will set the address of the `.text` output section to the current value of the location counter. The second will set it to the current value of the location counter aligned to the strictest alignment of a `.text` input section.

The address may be an arbitrary expression (see **Section 9.8 "Expressions in Linker Scripts"**). For example, to align the section on a `0x10` byte boundary, so that the lowest four bits of the section address are zero, the command could look like this:

```
.text ALIGN(0x10) : { *(.text) }
```

This works because `ALIGN` returns the current location counter aligned upward to the specified value.

Specifying *address* for a section will change the value of the location counter.

9.7.5.7 OUTPUT SECTION DATA

Explicit bytes of data may be inserted into an output section by using `BYTE`, `SHORT`, `LONG` or `QUAD` as an output section command. Each keyword is followed by an expression in parentheses providing the value to store. The value of the expression is stored at the current value of the location counter.

The `BYTE`, `SHORT`, `LONG` and `QUAD` commands store one, two, four and eight bytes (respectively). For example, this command will store the four byte value of the symbol `addr`:

```
LONG(addr)
```

After storing the bytes, the location counter is incremented by the number of bytes stored. When using data commands in a program memory section, it is important to note that the linker considers program memory to be 32-bits wide, even though only 24 bits are physically implemented. Therefore, the most significant 8 bits of a `LONG` data value are not loaded into device memory.

Data commands only work inside a section description and not between them, so the following will produce an error from the linker:

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

whereas this will work:

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```


© 2008 Microchip Technology Inc.

The FILL command may be used to set the fill pattern for the current section. It is followed by an expression in parentheses. Any otherwise unspecified regions of memory within the section (for example, gaps left due to the required alignment of input sections) are filled with the two least significant bytes of the expression, repeated as necessary. A FILL statement covers memory locations after the point at which it occurs in the section definition; by including more than one FILL statement, different fill patterns may be used in different parts of an output section.

This example shows how to fill unspecified regions of memory with the value 0x9090:

```
FILL(0x9090)
```

The FILL command is similar to the =fillexp output section attribute (see **Section 9.7.5.9 "Output Section Attributes"**), but it only affects the part of the section following the FILL command, rather than the entire section. If both are used, the FILL command takes precedence.

9.7.5.8 OUTPUT SECTION DISCARDING

The linker will not create an output section which does not have any contents. This is for convenience when referring to input sections that may or may not be present in any of the input files. For example:

```
.foo { *(.foo) }
```

will only create a .foo section in the output file if there is a .foo section in at least one input file.

If anything other than an input section description is used as an output section command, such as a symbol assignment, then the output section will always be created, even if there are no matching input sections.

The special output section name /DISCARD/ may be used to discard input sections. Any input sections which are assigned to an output section named /DISCARD/ are not included in the output file.

9.7.5.9 OUTPUT SECTION ATTRIBUTES

To review, the full description of an output section is:

```
name [address] [(type)] : [AT(lma)]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

name, *address* and output-section-command have already been described. In the following sections, the remaining section attributes will be described.

9.7.5.10 OUTPUT SECTION TYPE

Each output section may have a type. The type is a keyword in parentheses. The following types are defined:

NOLOAD

The section should be marked as not loadable, so that it will not be loaded into memory when the program is run.

DSECT, COPY, INFO, OVERLAY

These type names are supported for backward compatibility, and are rarely used. They all have the same effect: the section should be marked as not allocatable, so that no memory is allocated for the section when the program is run.

The linker normally sets the attributes of an output section based on the input sections which map into it. This can be overridden by using the section type. For example, in the script sample below, the `ROM` section is addressed at memory location `0` and does not need to be loaded when the program is run. The contents of the `ROM` section will appear in the linker output file as usual.

```
SECTIONS {
  ROM 0 (NOLOAD) : { ... }
  ...
}
```

9.7.5.11 OUTPUT SECTION LMA

Every section has a virtual address (VMA) and a load address (LMA). The address expression which may appear in an output section description sets the VMA.

The linker will normally set the LMA equal to the VMA. This can be changed by using the `AT` keyword. The expression lma that follows the `AT` keyword specifies the load address of the section. Alternatively, with `AT>lma_region` expression, a memory region may be specified for the section's load address. See **Section 9.7.4 "MEMORY Command"**.

This feature is designed to make it easy to build a ROM image. For example, the following linker script creates three output sections: one called `.text`, which starts at `0x1000`, one called `.mdata`, which is loaded at the end of the `.text` section even though its VMA is `0x2000`, and one called `.bss` to hold uninitialized data at address `0x3000`. The `symbol _data` is defined with the value `0x2000`, which shows that the location counter holds the VMA value, not the LMA value.

```
SECTIONS
  {
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR (.text) + SIZEOF (.text) )
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0x3000 :
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

The run-time initialization code for use with a program generated with this linker script would include a function to copy the initialized data from the ROM image to its run-time address. The initialization function could take advantage of the symbols defined by the linker script.

It would rarely be necessary to write such a function, however. The 16-bit linker includes automatic support for the initialization of bss-type and data-type sections. Instead of mapping a data section into both program memory and data memory (as this example implies), the linker creates a special template in program memory which includes all of the relevant information. See **Section 10.8 "Initialized Data"** for details.

© 2008 Microchip Technology Inc.

9.7.5.12 OUTPUT SECTION REGION

A section can be assigned to a previously defined region of memory by using `>region`. See **Section 9.7.4 "MEMORY Command"**.

Here is a simple example:

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

9.7.5.13 OUTPUT SECTION FILL

A fill pattern can be set for an entire section by using `=fillexp`. `fillexp` as an expression. Any otherwise unspecified regions of memory within the output section (for example, gaps left due to the required alignment of input sections) will be filled with the two least significant bytes of the value, repeated as necessary.

The fill value can also be changed with a `FILL` command in the output section commands; see **Section 9.7.5.7 "Output Section Data"**.

Here is a simple example:

```
SECTIONS { .text : { *(.text) } =0x9090 }
```

9.7.5.14 OVERLAY DESCRIPTION

An overlay description provides an easy way to describe sections which are to be loaded as part of a single memory image but are to be run at the same memory address. At run time, some sort of overlay manager will copy the overlaid sections in and out of the run-time memory address as required, perhaps by simply manipulating addressing bits.

This approach is not suitable for defining sections that will be used with the Program Space Visibility (PSV) window, because the `OVERLAY` command does not permit individual load addresses to be specified for each section. Instead, the 16-bit linker provides automatic support for read-only sections in the PSV window. See **Section 10.9 "Read-only Data"** for details.

Overlays are described using the `OVERLAY` command. The `OVERLAY` command is used within a `SECTIONS` command, like an output section description. The full syntax of the `OVERLAY` command is as follows:

```
 OVERLAY [start] : [NOCROSSREFS] [AT ( ldaddr )]
{
  secname1
    {
      output-section-command
      output-section-command
      ...
    } [:phdr...] [=fill]
  secname2
    {
      output-section-command
      output-section-command
      ...
    } [:phdr...] [=fill]
  ...
} [>region] [:phdr...] [=fill]
```

Everything is optional except `OVERLAY` (a keyword), and each section must have a name (`secname1` and `secname2` above). The section definitions within the `OVERLAY` construct are identical to those within the general `SECTIONS` construct, except that no addresses and no memory regions may be defined for sections within an `OVERLAY`.

The sections are all defined with the same starting address. The load addresses of the sections are arranged such that they are consecutive in memory starting at the load address used for the OVERLAY as a whole (as with normal section definitions, the load address is optional, and defaults to the start address; the start address is also optional, and defaults to the current value of the location counter).

If the NOCROSSREFS keyword is used, and there are any references among the sections, the linker will report an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another.

For each section within the OVERLAY, the linker automatically defines two symbols. The symbol __load_start_secname is defined as the starting load address of the section. The symbol __load_stop_secname is defined as the final load address of the section. Any characters within secname which are not legal within C identifiers are removed. C (or assembler) code may use these symbols to move the overlaid sections around as necessary.

At the end of the overlay, the value of the location counter is set to the start address of the overlay plus the size of the largest section.

Here is an example. Remember that this would appear inside a SECTIONS construct.

```
    OVERLAY 0x1000 : AT (0x4000)
  {
    .text0 { o1/*.o(.text) }
    .text1 { o2/*.o(.text) }
  }
```

This will define both .text0 and .text1 to start at address 0x1000. .text0 will be loaded at address 0x4000, and .text1 will be loaded immediately after .text0. The following symbols will be defined: __load_start_text0, __load_stop_text0, __load_start_text1, __load_stop_text1.

C code to copy overlay .text1 into the overlay area might look like the following:

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1,
        &__load_stop_text1 - &__load_start_text1);
```

The OVERLAY command is a convenience, since everything it does can be done using the more basic commands. The above example could have been written identically as follows.

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
__load_start_text1 = LOADADDR (.text1);
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```


© 2008 Microchip Technology Inc.

9.7.6 Other Linker Script Commands

There are several other linker script commands, which are described briefly:

ASSERT(*exp, message*)

Ensure that *exp* is non-zero. If it is zero, then exit the linker with an error code, and print *message*.

ENTRY(*symbol*)

Specify *symbol* as the first instruction to execute in the program. The linker will record the address of this symbol in the output object file header. This does not affect the Reset instruction at address zero, which must be generated in some other way. By convention, the 16-bit linker scripts construct a GOTO __reset instruction at address zero.

EXTERN(*symbol symbol ...*)

Force *symbol* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. Several symbols may be listed for each EXTERN, and EXTERN may appear multiple times. This command has the same effect as the -u command line option.

FORCE_COMMON_ALLOCATION

This command has the same effect as the -d command line option: to make 16-bit linker assign space to common symbols even if a relocatable output file is specified (-r).

NOCROSSREFS(*section section ...*)

This command may be used to tell 16-bit linker to issue an error about any references among certain output sections. In certain types of programs, when one section is loaded into memory, another section will not be. Any direct references between the two sections would be errors.

The NOCROSSREFS command takes a list of output section names. If the linker detects any cross references between the sections, it reports an error and returns a non-zero exit status. The NOCROSSREFS command uses output section names, not input section names.

OUTPUT_ARCH(*processor_name*)

Specify a target processor for the link. This command has the same effect as the -p,--processor command line option. If both are specified, the command line option takes precedence. The processor name should appear in quotes; for example "30F6014", "24FJ128GA010", or "33FJ128GP706".

OUTPUT_FORMAT(*format_name*)

The OUTPUT_FORMAT command names the object file format to use for the output file.

TARGET(*bfdname*)

The TARGET command names the object file format to use when reading input files. It affects subsequent INPUT and GROUP commands.

9.8 EXPRESSIONS IN LINKER SCRIPTS

The syntax for expressions in the linker script language is identical to that of C expressions. All expressions are evaluated as 32-bit integers.

You can use and set symbol values in expressions.

The linker defines several special purpose built-in functions for use in expressions.

- Constants
- Symbol Names
- The Location Counter
- Operators
- Evaluation
- The Section of an Expression
- Built-in Functions

9.8.1 Constants

All constants are integers.

As in C, the linker considers an integer beginning with `0` to be octal, and an integer beginning with `0x` or `0X` to be hexadecimal. The linker considers other integers to be decimal.

In addition, you can use the suffixes `K` and `M` to scale a constant by 1024 or 1024*1024 respectively. For example, the following all refer to the same quantity:

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

9.8.2 Symbol Names

Unless quoted, symbol names start with a letter, underscore, or period and may include letters, digits, underscores, periods and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol which contains odd characters or has the same name as a keyword by surrounding the symbol name in double quotes:

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

Since symbols can contain many non-alphabetic characters, it is safest to delimit symbols with spaces. For example, `A-B` is one symbol, whereas `A - B` is an expression involving subtraction.

© 2008 Microchip Technology Inc.

9.8.3 The Location Counter

The special linker variable dot '.' always contains the current output location counter. Since the . always refers to a location in an output section, it may only appear in an expression within a SECTIONS command. The '.' symbol may appear anywhere that an ordinary symbol is allowed in an expression.

Assigning a value to '.' will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may never be moved backwards.

```
SECTIONS
{
  output :
    {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . += 1000;
      file3(.text)
    } = 0x1234;
}
```

In the previous example, the `.text` section from `file1` is located at the beginning of the output section `output`. It is followed by a 1000 byte gap. Then the `.text` section from `file2` appears, also with a 1000 byte gap following before the `.text` section from `file3`. The notation `= 0x1234` specifies what data to write in the gaps.

'.' actually refers to the byte offset from the start of the current containing object. Normally this is the `SECTIONS` statement, whose start address is `0`, hence '.' can be used as an absolute address. If '.' is used inside a section description, however, it refers to the byte offset from the start of that section, not an absolute address. Thus in a script like this:

```
SECTIONS
{
    . = 0x100
    .text: {
      *(.text)
      . = 0x200
    }
    . = 0x500
    .data: {
      *(.data)
      . += 0x600
    }
}
```

The `.text` section will be assigned a starting address of `0x100` and a size of exactly `0x200` bytes, even if there is not enough data in the `.text` input sections to fill this area. (If there is too much data, an error will be produced because this would be an attempt to move '.' backwards). The `.data` section will start at `0x500` and it will have an extra `0x600` bytes worth of space after the end of the values from the `.data` input sections and before the end of the `.data` output section itself.

9.8.4 Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels:

TABLE 9-1: PRECEDENCE OF OPERATORS

Precedence	Associativity	Operators	Description
1 (highest)	left	! - ~	Prefix operators
2	left	* / %	multiply, divide, modulo
3	left	+ -	add, subtract
4	left	>> <<	bit shift right, left
5	left	== != > < <= >=	Relational
6	left	&	bitwise and
7	left	|	bitwise or
8	left	&&	logical and
9	left	||	logical or
10	right	? :	Conditional
11 (lowest)	right	&= += -= *= /=	Symbol assignments

9.8.5 Evaluation

The linker evaluates expressions lazily. It only computes the value of an expression when absolutely necessary.

The linker needs some information, such as the value of the start address of the first section, and the origins and lengths of memory regions, in order to do any linking at all. These values are computed as soon as possible when the linker reads in the linker script.

However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

The sizes of sections cannot be known until after allocation, so assignments dependent upon these are not performed until after allocation.

Some expressions, such as those depending upon the location counter '.', must be evaluated during section allocation.

If the result of an expression is required, but the value is not available, then an error results. For example, a script like the following:

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

will cause the error message "non-constant expression for initial address".

9.8.6 The Section of an Expression

When the linker evaluates an expression, the result is either absolute or relative to some section. A relative expression is expressed as a fixed offset from the base of a section.

The position of the expression within the linker script determines whether it is absolute or relative. An expression which appears within an output section definition is relative to the base of the output section. An expression which appears elsewhere will be absolute.

© 2008 Microchip Technology Inc.

A symbol set to a relative expression will be relocatable if you request relocatable output using the `-r` option. That means that a further link operation may change the value of the symbol. The symbol's section will be the section of the relative expression.

A symbol set to an absolute expression will retain the same value through any further link operation. The symbol will be absolute, and will not have any particular associated section.

You can use the built-in function `ABSOLUTE` to force an expression to be absolute when it would otherwise be relative. For example, to create an absolute symbol set to the address of the end of the output section `.data`:

```
SECTIONS
  {
    .data : { *(.data) _edata = ABSOLUTE(.); }
  }
```

If `ABSOLUTE` were not used, `_edata` would be relative to the `.data` section.

9.8.7 Built-in Functions

The linker script language includes a number of built-in functions for use in linker script expressions.

- ABSOLUTE(exp)
- ADDR(section)
- ALIGN(exp)
- BLOCK(exp)
- DEFINED(symbol)
- LOADADDR(section)
- MAX(exp1, exp2)
- MIN(exp1, exp2)
- NEXT(exp)
- SIZEOF(section)

9.8.7.1 ABSOLUTE(*EXP*)

Return the absolute (non-relocatable, as opposed to non-negative) value of the expression *exp*. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section relative. See **Section 9.8.6 "The Section of an Expression"**.

9.8.7.2 ADDR(*SECTION*)

Return the absolute address (the VMA) of the named section. Your script must previously have defined the location of that section. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS { ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
  .output :
    {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
    }
...
}
```

9.8.7.3 ALIGN(*EXP*)

Return the location counter (.) aligned to the next exp boundary. *exp* must be an expression whose value is a power of two. This is equivalent to

```
(. + exp - 1) & ~(exp - 1)
```

ALIGN doesn't change the value of the location counter; it just does arithmetic on it. Here is an example which aligns the output .data section to the next 0x2000 byte boundary after the preceding section and sets a variable within the section to the next 0x8000 boundary after the input sections:

```
SECTIONS { ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
...
}
```

The first use of ALIGN in this example specifies the location of a section because it is used as the optional address attribute of a section definition (see **Section 9.7.5 "SECTIONS Command"**). The second use of ALIGN is used to define the value of a symbol.

The built-in function NEXT is closely related to ALIGN.

9.8.7.4 BLOCK(*EXP*)

This is a synonym for ALIGN, for compatibility with older linker scripts. It is most often seen when setting the address of an output section.

9.8.7.5 DEFINED(*SYMBOL*)

Return 1 if symbol is in the linker global symbol table and is defined; otherwise return 0. You can use this function to provide default values for symbols. For example, the following script fragment shows how to set a global symbol begin to the first location in the .text section, but if a symbol called begin already existed, its value is preserved:

```
SECTIONS { ...
  .text : {
    begin = DEFINED(begin) ? begin : . ;
    ...
  }
  ...
}
```

9.8.7.6 LOADADDR(*SECTION*)

Return the absolute LMA of the named section. This is normally the same as ADDR, but it may be different if the AT attribute is used in the output section definition (see **Section 9.7.5 "SECTIONS Command"**).

© 2008 Microchip Technology Inc.

9.8.7.7 MAX(*EXP1*, *EXP2*)

Returns the maximum of *exp1* and *exp2*.

9.8.7.8 MIN(*EXP1*, *EXP2*)

Returns the minimum of *exp1* and *exp2*.

9.8.7.9 NEXT(*EXP*)

Return the next unallocated address that is a multiple of *exp*. This function is equivalent to ALIGN(*exp*).

9.8.7.10 SIZEOF(*SECTION*)

Return the size in bytes of the named section, if that section has been allocated. If the section has not been allocated when this is evaluated, the linker will report an error. In the following example, symbol_1 and symbol_2 are assigned identical values:

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
    }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
...
}
```

NOTES:

© 2008 Microchip Technology Inc.

Chapter 10. Linker Processing

10.1 INTRODUCTION

How the MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB LINK30) builds an application from input files is discussed here.

10.2 HIGHLIGHTS

Topics covered in this chapter are:

- Overview of Linker Processing
- Memory Addressing
- Linker Allocation
- Global and Weak Symbols
- Handles
- Initialized Data
- Read-only Data
- Stack Allocation
- Heap Allocation
- Interrupt Vector Tables
- Optimizing Memory Usage
- Boot and Secure Segments
- Notable Symbols

10.3 OVERVIEW OF LINKER PROCESSING

A linker combines one or more object files, with optional archive files, into a single executable output file. The object files contain relocatable sections of code and data which the linker will allocate into target memory. The entire process is controlled by a linker script, also known as a link command file. A linker script is required for every link.

The link process may be broken down into 6 steps:

1. Loading Input Files
2. Allocating Memory
3. Resolving Symbols
4. Creating Special Sections
5. Computing Absolute Addresses
6. Building the Output File

10.3.1 Loading Input Files

The initial task of the linker is to interpret link command options and load input files. If a linker script is specified, that file is opened and interpreted. Otherwise an internal default linker script is used. In either case, the linker script provides a description of the target device, including specific memory region information and Special Function Register (SFR) addresses. See **Chapter 9. "Linker Scripts"** for more details.

Next the linker opens all of the input object files. Each input file is checked to make sure the object format is compatible. If the object format is not compatible, an error is generated. The contents of each input file are then loaded into internal data structures. Typically each input file will contain multiple sections of code or data. Each section contains a list of relocation entries which associate locations in a section's raw data with relocatable symbols.

10.3.2 Allocating Memory

After all of the input files have been loaded, the linker allocates memory. This is accomplished by assigning each input section to an output section. The relation between input and output sections is defined by a section map in the linker script. An output section may or may not have the same name as an input section. Each output section is then assigned to a memory region in the target device.

Note:	Input sections are derived from source code by the compiler or the assembler. Output sections are created by the linker.

If an input section is not explicitly assigned to an output section, the linker will allocate the unassigned section according to section attributes. For more information about linker allocation, see **Section 10.5 "Linker Allocation"**.

10.3.3 Resolving Symbols

Once memory has been allocated, the linker begins the process of resolving symbols. Symbols defined in each input section have offsets that are relative to the beginning of the section. The linker converts these values into output section offsets.

Next, the linker attempts to match all external symbol references with a corresponding symbol definition. Multiple definitions of the same external symbol result in an error. If an external symbol is not found, an attempt is made to locate the symbol definition in an archive file. If the symbol definition is found in an archive, the corresponding archive module is loaded.

Modules loaded from archives may contain additional symbol references, so the process continues until all external symbol references have matching definitions. External symbols that are defined as "weak" receive special processing, as explained in **Section 10.6 "Global and Weak Symbols"**. If any external symbol reference remains undefined, an error is generated.

References to redundant functions in archive files will be merged in order to conserve memory. For example, both integer and floating-point versions of the standard C formatted I/O functions are included in `libc.a`. The 16-bit compiler will generate references to the appropriate function, based on a static analysis of format strings. When multiple object files are combined by the linker, both versions of a particular I/O function may be referenced. In such cases the integer functions are redundant, since they represent a subset of the floating-point functionality. The linker will detect this situation, and merge the I/O functions together to conserve memory. This optimization may be disabled with the `--no-smart-io` option.

10.3.4 Creating Special Sections

After the symbols have been resolved, the linker constructs any special input or output sections that are required. For example, the compiler or assembler may have created function pointers using the `handle()` operator. The linker then builds a special input section named `.handle` to implement a jump table. For more information about handles, see **Section 10.7 "Handles"**.

© 2008 Microchip Technology Inc.

The linker also constructs a special output section named `.dinit` to support initialized data. Section `.dinit` is an initialization template that is interpreted by the C run-time library. For more information about initialized data, see **Section 10.8 "Initialized Data"**.

10.3.5 Computing Absolute Addresses

After the special sections have been created, the final sizes of all output sections are known. The linker then computes absolute addresses for all output sections and external symbols. Each output section is checked to make sure it falls within its assigned memory regions. If any section falls outside of its memory region, an error is generated. Any symbols defined in the linker script are also computed.

Boundaries of the stack and heap are calculated, based on the extent of unused data memory. If insufficient memory is available, an error is generated. For more information about the stack and heap, see **Section 10.10 "Stack Allocation"** and **Section 10.11 "Heap Allocation"**.

10.3.6 Building the Output File

Finally, the linker builds the output file. Relocation entries in each section are patched using absolute addresses. If the address computed for a symbol does not fit in the relocation entry, a link error results. This can occur, for example, if a function pointer is referenced without the `handle()` operator and its address is too large to fit in 16 bits.

A link map is also generated if requested with the appropriate option. The link map includes a memory usage report, which shows the starting address and length of all sections in data memory and program memory. For more information about the link map, see **Section 9.5.4 "Input/Output Section Map"**.

10.4 MEMORY ADDRESSING

The dsPIC30F devices use a modified Harvard architecture with separate data and program memory spaces. Data memory is both byte-oriented (8 bits wide) and word-oriented (16 bits wide). Bytes are assigned sequential addresses, starting with 0, 1, 2, 3 and so on. Words are assigned sequential even addresses, starting with 0, 2, 4, 6 and so on.

Program memory is word-oriented, where each instruction word is 24 bits wide. Instruction words are assigned sequential even addresses, starting with 0, 2, 4, 6 and so on. The Program Counter (PC) indicates the next instruction to be executed, and increments by 2 for each instruction word. Individual bytes in a program memory word are not addressable.

While a traditional Harvard architecture does not permit access to data stored in program memory, the 16-bit architecture provides two ways to accomplish this task: table access instructions and the Program Space Visibility (PSV) window.

- Table Access Instructions
- Program Space Visibility (PSV) Window

10.4.1 Table Access Instructions

The table access instructions `tblrdl`, `tblrdh`, `tblwtl` and `tblwth` can be used to access data stored in program memory. Data is addressed through a 16-bit data register pointer in combination with the 8-bit TBLPAG register. The special operators `tbloffset()` and `tblpage()` facilitate table access in assembly language. See the 16-bit assembler documentation, "Table Read/Write Instructions", for more information.

The linker resolves symbolic references to labels in program memory for use with the table access instructions. Although data in program memory can be specified one byte at a time, only the least-significant byte in each instruction word has a unique address. For example, consider the following assembly source code example:

```
     .section prog,code
L1:  .pbyte 1
L2:  .pbyte 2
L3:  .pbyte 3
L4:  .pbyte 4
     .pbyte 5
     .pbyte 6
     .pbyte 7,8,9
```

In this example, the `code` section attribute designates a section to be allocated in program memory, and the `.pbyte` directives define individual byte constants. Since labels must resolve to a valid PC address, the assembler adds padding after each of the first three constants. Subsequent constants do not require padding. The following assembly listing excerpt illustrates the organization of these constants in program memory:

```
1                       .section prog,code
2 000000  01 00 00      L1:.pbyte 1
3 000002  02 00 00      L2:.pbyte 2
4 000004  03 00 00      L3:.pbyte 3
5 000006  04            L4:.pbyte 4
6            05         .pbyte 5
7               06      .pbyte 6
8 000008  07 08 09      .pbyte 7,8,9
```

Constants 1, 2, 3 are padded out to a full instruction word and have unique PC addresses. Constants 4, 5, 6 are packed into a single instruction word and share the same address.

10.4.2 Program Space Visibility (PSV) Window

The Program Space Visibility window can be used to access data stored in the least significant 16 bits of program memory. When PSV is enabled, the upper 32K of data memory space (0x8000-0xFFFF) functions as a window into program memory. Data is addressed through a 16-bit data register pointer in combination with the 8-bit PSVPAG register. The special operators `psvoffset()` and `psvpage()` are provided to facilitate PSV access in assembly language. Built-in functions `__builtin_psvoffset()` and `__builtin_psvpage()` are provided to facilitate PSV access in C.

The linker supports PSV window operations through the use of read-only data sections. For a detailed discussion of read-only sections, see **Section 10.9 "Read-only Data"**.

© 2008 Microchip Technology Inc.

10.5 LINKER ALLOCATION

Linker allocation is controlled by the linker script, and proceeds in three steps:

1. Mapping Input Sections to Output Sections
2. Assigning Output Sections to Regions
3. Allocating Unmapped Sections

Steps 1 and 2 are performed by a sequential memory allocator. Input sections which appear in the linker script are assigned to specific memory regions in the target devices. Addresses within a memory region are allocated sequentially, beginning with the lowest address and growing upwards.

Step 3 is performed by a best-fit memory allocator. Input sections which do not appear in the linker script are assigned to memory regions according to their attributes. The best-fit allocator makes efficient use of any remaining memory, including gaps between output sections that may have been left by the sequential allocator.

If memory has been reserved for the `boot` and/or `secure` segments, it will be allocated by the best-fit allocator in step 3. The sequential allocator will avoid these segments, so sections designated with the `boot` or `secure` attributes should not appear in the linker script.

10.5.1 Mapping Input Sections to Output Sections

Input sections are grouped and mapped into output sections, according to the section map. When an output section contains several different input sections, the exact ordering of input sections may be important. For example, consider the following output section definition:

```
/*
** User Code and Library Code
*/
.text :
  {
        *(.init);
        *(.user_init);
        *(.handle);
        *(.libc) *(.libm) *(.libdsp); /* keep together in this order */
        *(.lib*);
  } >program
```

Here the output section named `.text` is defined. Notice that the contents of this section are specified within curly braces {}. After the closing brace, `>program` indicates that this output section should be assigned to memory region program.

The contents of output section `.text` may be interpreted as follows:

- Input sections named `.init` are collected and mapped into the output section. By convention, there is only one `.init` section, and it contains the startup code for an application. It appears first in the output section (i.e., at the lowest address) so that its address is readily available if necessary.
- Input sections named `.user_init` are collected and mapped into the output section. These sections are created by the compiler and refer to functions that have been decorated with the `user_init` attribute. Their position within the output section is not critical, but since they are associated with section `.init`, they are located immediately after.
- All input sections named `.handle` are collected and mapped into the output section. `.handle` sections occupy a relatively low address range, which is a requirement for code handles.

- Input sections named .libc, .libm and .libdsp are collected and mapped into the output section. Grouping these sections ensures locality of reference for the run-time library functions, so that PC-relative instructions can be used for maximum efficiency.
- Input sections which match the wildcard pattern .lib* are collected and mapped into the output section. This includes libraries such as the peripheral libraries (which are allocated in section .libperi).

10.5.2 Assigning Output Sections to Regions

Once the sizes of all output sections are known, they are assigned to memory regions. Normally a region is specified in the output section definition. If a region is not specified, the first defined memory region will be used.

Memory regions are filled sequentially, from lower to higher addresses, in the same order that sections appear in the section map. Memory reserved for boot or secure segments will be avoided, as well as sections that have been marked with the address attribute in source code. A location counter, unique to each region, keeps track of the next available memory location. There are two conditions which may cause gaps in the allocation of memory within a region:

1. The section map specifies an absolute address for an output section, or
2. The output section has a particular alignment requirement.

In either case, any intervening memory between the current location counter and the absolute (or aligned) address is skipped. Once a range of memory has been skipped, it is available for use by the best-fit allocator. The exact address of all items allocated in memory may be determined from the link map file.

Section alignment requirements typically arise in DSP programming. To utilize modulo addressing, it is necessary to align a block of memory to a particular storage boundary. This can be accomplished with the aligned attribute in C, or with the .align directive in assembly language. The section containing an aligned memory block must also be aligned, to the same (or greater) power of 2. If two or more input sections have different alignment requirements, the largest alignment is used for the output section.

Another restriction on memory allocation is associated with read-only data sections. Read-only data sections are identified with the psv section attribute and are dedicated for use in the Program Space Visibility (PSV) window. The C compiler creates a read-only data section named .const to store constants when the --mconst-in-code option is selected.

To allow efficient access of constant tables in the PSV window, the linker ensures that a read-only section will not cross a PSVPAG boundary. Therefore a single setting of the PSVPAG register can be used to access the entire section. If necessary, output sections in program memory will be re-sorted after the sequential allocation pass to accommodate this restriction. If an absolute address has been specified in the linker script for a particular section, it will not be moved. In general, fully relocatable sections provide the most flexibility for efficient memory allocation.

Note: Sections with specific alignment requirements, such as psv sections or sections intended for modulo addressing, may be allocated most efficiently by the best-fit allocator. For best-fit allocation, these sections should not appear in the linker script.

© 2008 Microchip Technology Inc.

10.5.3 Allocating Unmapped Sections

After all sections that appear in the section map are allocated, any remaining sections are considered to be unmapped. Unmapped sections are allocated according to section attributes. The linker uses a best-fit memory allocator to determine the most efficient arrangement in memory. The primary emphasis of the best-fit allocator is the reduction or elimination of memory gaps due to address alignment restrictions.

Since data memory is limited on many 16-bit devices, and several architectural features imply address alignment restrictions, efficient allocation of data memory is particularly important. By convention, data memory sections are not explicitly mapped in linker scripts, thus providing maximum flexibility for the best-fit memory allocator.

Section attributes affect memory allocation as described below. For a general discussion of section attributes, see **Section 6.3 "Directives that Define Sections"**.

code

The code attribute specifies that a section should be allocated in program memory, as defined by region `program` in the linker script. The following attributes may be used in conjunction with code and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address
- `boot` specifies the boot segment
- `secure` specifies the secure segment

data

The data attribute specifies that a section should be allocated as initialized storage in data memory, as defined by region `data` in the linker script. The following attributes may be used in conjunction with data and will further specify the allocation:

- `address()` specifies an absolute address
- `near` specifies the first 8K of data memory
- `xmemory` specifies X address space, which includes all of region data below the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `ymemory` specifies Y address space, which includes all of region data above the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `dma` specifies dma address space, which includes the portion of region data between addresses __DMA_BASE and __DMA_END as defined in the linker script (for PIC24H MCUs and dsPIC33F DSCs only).

bss

The bss attribute specifies that a section should be allocated as uninitialized storage in data memory, as defined by region `data` in the linker script. The following attributes may be used in conjunction with bss and will further specify the allocation:

- `address()` specifies an absolute address
- `near` specifies the first 8K of data memory
- `xmemory` specifies X address space, which includes all of region data below the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)

- `ymemory` specifies Y address space, which includes all of region data above the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1

- `dma` specifies dma address space, which includes the portion of region data between addresses __DMA_BASE and __DMA_END as defined in the linker script (for PIC24H MCUs and dsPIC33F DSCs only).
- `boot` specifies the boot segment
- `secure` specifies the secure segment

persist

The persist attribute specifies that a section should be allocated as persistent storage in data memory, as defined by region `data` in the linker script. Persistent storage is not cleared or initialized by the C run-time library. The following attributes may be used in conjunction with persist and will further specify the allocation:

- `address()` specifies an absolute address
- `near` specifies the first 8K of data memory

- `xmemory` specifies X address space, which includes all of region data below the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)

- `ymemory` specifies Y address space, which includes all of region data above the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1

- `dma` specifies dma address space, which includes the portion of region data between addresses __DMA_BASE and __DMA_END as defined in the linker script (for PIC24H MCUs and dsPIC33F DSCs only).

psv

The psv attribute specifies that a section should be allocated in program memory, as defined by region `program` in the linker script. psv sections are intended for use with the Program Space Visibility window, and will be located so that the entire contents may be accessed using a single setting of the PSVPAG register. This allocation rule implies that the total size of a psv section can not exceed 32K. The following attributes may be used in conjunction with psv and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `boot` specifies the boot segment
- `secure` specifies the secure segment

© 2008 Microchip Technology Inc.

memory

The memory attribute specifies that a section should be allocated in external or user-defined memory. The following attributes may be used in conjunction with memory and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `noload` specifies that the section should not be loaded with the primary application

eedata – dsPIC30F DSCs only

The eedata attribute specifies that a section should be allocated in data EEPROM memory, as defined by region `eedata` in the linker script. The following attributes may be used in conjunction with eedata and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `boot` specifies the boot segment
- `secure` specifies the secure segment

10.6 GLOBAL AND WEAK SYMBOLS

When a symbol reference appears in an object file without a corresponding definition, the symbol is declared external. By default, external symbols have global binding and are referred to as global symbols. External symbols may be explicitly declared with weak binding, using the `__weak__` attribute in C or the `.weak` directive in assembly language.

As the name implies, global symbols are visible to all input files involved in the link. There must be one (and only one) definition for every global symbol referenced. If a global definition is not found among the input files, archives will be searched and the first archive module found that contains the needed definition will be loaded. If no definition is found for a global symbol a link error is reported.

Weak symbols share the same name space as global symbols, but are handled differently. Multiple definitions of a weak symbol are permitted. If a weak definition is not found among the input files, archives are not searched and a value of 0 is assumed for all references to the weak symbol. A global symbol definition of the same name will take precedence over a weak definition (or the lack of one). In essence, weak symbols are considered optional and may be replaced by global symbols, or ignored entirely.

10.7 HANDLES

The modified Harvard architecture of dsPIC30F devices supports two memory spaces of unequal size. Data memory space can be fully addressed with 16 bits while program memory space requires 24 bits. Since the native integer data type (register width) is only 16 bits, there is an inherent difficulty in the allocation and manipulation of function pointers that require a full 24 bits. Reserving a pair of 16-bit registers to represent every function pointer is inefficient in terms of code space and execution speed, since many programs will fit in 64K words of program space or less. However, the linker must accommodate function pointers throughout the full 24-bit range of addressable program memory.

In order to ensure a valid 16-bit pointer for any function in the full program memory address space, the 16-bit assembler and linker support the `handle()` operator. The C compiler uses this operator whenever a function address is taken. Assembly programmers can use this operator three different ways:

```
mov    #handle(func),w0 ; handle() used in an instruction
.word  handle(func)     ; handle() used with a data word directive
.pword handle(func)     ; handle() used with a instruction word
                        ;directive
```

The linker searches all input files for handle operators and constructs a jump table in a section named `.handle`. For each function that is referenced by one or more handle operators, a single entry is made in the jump table. Each entry is a `GOTO` instruction. Note that `GOTO` is capable of reaching any function in the full 24- bit address space. Section `.handle` is allocated low in program memory, well within the range of a 16-bit pointer.

When the output file is built, the absolute addresses of all functions are known. Each handle relocation entry is filled with an absolute address. If the address of the target function fits in 16 bits, it is inserted directly into the object code. If the absolute address of the target function exceeds 16 bits, the address of the corresponding entry in the jump table is used instead. Only functions located beyond the range of 16-bit addressing suffer any performance penalty with this technique. However, there is a code space penalty for each unused entry in the jump table.

In order to conserve program memory, the handle jump table can be suppressed for certain devices, or whenever the application programmer is sure that all function pointers will fit in 16 bits. One way is to specify the `--no-handles` link option on the command line or in the IDE. Another way is to define a symbol named __NO_HANDLES in the linker script:

```
__NO_HANDLES = 1;
```

Linker scripts for 16-bit devices with 32K instruction words or less all contain the `__NO_HANDLES` definition to suppress the handle jump table.

> **Note:** If the handle jump table is suppressed, and the target address of a function pointer does not fit in 16 bits, a "relocation truncated" link error will be generated.

10.8 INITIALIZED DATA

The linker provides automatic support for initialized variables in data memory. Variables are allocated in sections. Each data section is declared with a flag that indicates whether it is initialized, or not initialized.

To control the initialization of the various data sections, the linker constructs a data initialization template. The template is allocated in program memory, and is processed at start-up by the run-time library. When the application main program takes control, all variables in data memory have been initialized.

- Standard Data Section Names
- Data Initialization Template
- Run-Time Library Support

© 2008 Microchip Technology Inc.

10.8.1 Standard Data Section Names

Traditionally, linkers based on the GNU technology support three sections in the linked binary file:

TABLE 10-1: TRADITIONAL SECTION NAMES

Section Name	Description	Attribute
.text	executable code	code
.data	data memory that receives initial values	data
.bss	data memory that is not initialized	bss

The name “bss” dates back several decades, and means memory “Block Started by Symbol”. By convention, bss memory is filled with zeros during program start-up.

The traditional section names are considered to have implied attributes as listed in Table 10-1. The `code` attribute indicates that the section contains executable code and should be loaded in program memory. The `bss` attribute indicates that the section contains data storage that is not initialized, but will be filled with zeros at program start-up. The `data` attribute indicates that the section contains data storage that receives initial values at start-up.

Assembly applications may define additional sections with explicit attributes using the section directive described in **Section 6.3 “Directives that Define Sections”**. For C applications, the 16-bit compiler will automatically define sections to contain variables and functions as needed. For more information on the attributes of variables and functions that may result in automatic section definition, see the “*MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs User's Guide”* (DS51284).

> **Note:** Whenever a section directive is used, all declarations that follow are assembled into the named section. This continues until another section directive appears, or the end of file. For more information on defining sections and section attributes, see **Section 6.3 “Directives that Define Sections”**.

10.8.2 Data Initialization Template

As noted in **Section 10.8.1 “Standard Data Section Names”**, the 16-bit Language Tools support bss-type sections (memory that is not initialized) as well as data-type sections (memory that receives initial values). The data-type sections receive initial values at start-up, and the bss-type sections are filled with zeros.

A generic data initialization template is used that supports any number of arbitrary bss-type sections or data-type sections. The data initialization template is created by the linker and is loaded into an output section named `.dinit` in program memory. Start-up code in the run-time library interprets the template and initializes data memory accordingly.

The data initialization template contains one record for each output section in data memory. The template is terminated by a null instruction word. The format of a data initialization record is:

```
/* data init record */
struct data_record {
  char *dst;        /* destination address  */
  int  len;         /* length in bytes      */
  int  format;      /* format code          */
  char dat[0];      /* variable length data */
};
```

The first element of the record is a pointer to the section in data memory. The next two elements are the section length and format code, respectively. The fourth element is an optional array of data bytes. For bss-type sections, no data bytes are required.

The format code has three possible values.

TABLE 10-2: FORMAT CODE VALUES

Format Code	Description
0	Fill the output section with zeros
1	Copy 2 bytes of data from each instruction word in the data array
2	Copy 3 bytes of data from each instruction word in the data array

By default, data records are created using format 2. Format 2 conserves program memory by using the entire 24-bit instruction word to store initial values. Note that this format causes the encoded instruction words to appear as random and possibly invalid instructions if viewed in the disassembler.

Format 1 data records may be created by specifying the `--no-pack-data` option. Format 1 uses only the lower 16 bits of each 24-bit instruction word to store initial values. The upper byte of each instruction word is filled with 0x0 by default and causes the template to appear as `NOP` instructions if viewed in the disassembler (and will be executed as such by the 16-bit device). A different value may be specified for the upper byte of the data template with the `--fill-data` option.

10.8.3 Run-Time Library Support

In order to initialize variables in data memory, the data initialization template must be processed at start-up, before the proper application takes control. For C programs, this function is performed by the start-up modules in `libpic30.a`. Assembly language programs can utilize these modules directly by linking with the file `crt0.o` or `crt1.o`. The source code for the start-up modules is provided in file `crt0.s` and `crt1.s`.

To utilize a start-up module, the application must allow the run-time library to take control at device reset. This happens automatically for C programs. The application's `main()` function is invoked after the start-up module has completed its work. Assembly language programs should use the following naming conventions to specify which routine takes control at device reset.

TABLE 10-3: MAIN ENTRY POINTS

Main Entry Name	Description
`__reset`	Takes control immediately after device reset
`_main`	Takes control after the start-up module completes its work

Note that the first entry name (`__reset`) includes two leading underscore characters. The second entry name (`_main`) includes only one leading underscore character. The linker scripts construct a `GOTO __reset` instruction at location 0 in program memory, which transfers control upon device reset.

The primary start-up module (`crt0.o`) is linked by default and performs the following:

1. The stack pointer (W15) and stack pointer limit register (SPLIM) are initialized, using values provided by the linker or a custom linker script. For more information, see **Section 10.10 "Stack Allocation"**.
2. If a `.const` section is defined, it is mapped into the Program Space Visibility (PSV) window by initializing the PSVPAG and CORCON registers. Note that a `.const` section is defined when the "Constants in code space" option is selected in MPLAB IDE, or the `-mconst-in-code` option is specified on the compiler command line.

© 2008 Microchip Technology Inc.

3. The data initialization template in section .dinit is read, causing all uninitialized sections to be cleared, and all initialized sections to be initialized with values read from program memory.
4. If the application has defined user_init functions, section .user_init is called.
5. The function main is called with no parameters.
6. If main returns, the processor will reset.

The alternate start-up module (crt1.o) is linked when the --no-data-init option is specified. It performs the same operations, except for step (3), which is omitted. The alternate start-up module is much smaller than the primary module, and can be selected to conserve program memory if data initialization is not required.

Source code (in 16-bit assembly language) for both modules is provided in the c:\Program Files\Microchip\MPLAB C30\src directory. The start-up modules may be modified if necessary. For example, if an application requires main to be called with parameters, a conditional assembly directive may be switched to provide this support.

10.9 READ-ONLY DATA

Read-only data sections are located in program memory, but are defined and accessed just like data memory. They are useful for storing constant tables that are too large for available data memory. The C compiler creates a read-only section named .const when the -mconst-in-code option is specified.

The psv section attribute is used to designate read-only data sections. The contents of read-only data sections may be specified with data directives, as shown in the following assembly source example:

```
.section rdonly,psv
L1:  .byte 1
L2:  .byte 2
```

In this example, section rdonly will be allocated in program memory. Both byte constants will be located in the same program memory word, followed by a pad byte. Unlike other sections in program memory, read-only sections are byte addressable. Each label is resolved to a unique address that lies with the PSV address range.

The linker allocates read-only sections such that they do not cross a PSV page boundary. Therefore, a single setting of the PSVPAG register will access the entire section. A maximum length restriction is implied; the linker will issue an error message if any read-only data section exceeds 32 Kbytes. Only the least significant 16 bits of each instruction word are available for data storage (bits 16-23). The upper byte of each program word is filled with 0x0 or another value specified with the --fill-upper option. None of the p-variant assembler directives (including .pbyte and .pword) are permitted in read-only data sections.

The following examples illustrate how bytes in read-only sections may be accessed:

```
; example 1
mov    #psvoffset(L1),w0  ; PSVPAG already set
mov    #psvoffset(L2),w1
mov.b  [w0],w2            ; load the byte at L1
mov.b  [w1],w3            ; load the byte at L2

; example 2
mov    #L1,w0   ; PSVPAG already set
mov    #L2,w1
mov.b  [w0],w2  ; load the byte at L1
mov.b  [w1],w3  ; load the byte at L2
```

Use of the `psvoffset()` operator is optional in this example. This is possible because read-only sections are dedicated for use in the PSV window. The generic form of example 2 will work whether `L1` or `L2` are defined in a read-only section or in an ordinary data section.

User-defined read-only sections do not require a custom linker script. Based on the `psv` section attribute, the linker will locate the section in program memory and map its labels into the PSV window. If the programmer wishes to declare a read-only section in a custom linker script, the same syntax may be used as for other sections in program memory:

```
/*
** User-Defined Constants in Program Memory
**
** This section is identified as a read-only section
** by use of the psv section attribute. It will be
** loaded into program memory and mapped into data
** memory using the PSV window.
*/
userconstants ADDR : AT (LOADADDR)
  {
        *(userconstants);
  } >program
```

In this example, `ADDR` specifies a data memory address in the range 0x8000 to 0xFFFE. `LOADADDR` specifies the corresponding address in program memory. The least significant 15 bits of each address should be the same.

Any number of read-only sections may share the PSV window. By default, only one read-only section is ensured to be visible for any one setting of the PSVPAG register. To make a read-only section visible, the following assembly code can be used:

```
mov  #psvpage(L1),w0   ; L1 is a label in the desired section
mov  w0,PSVPAG
```

If an application requires multiple read-only sections to be visible at the same time, the following linker script syntax will create a single output section from multiple input sections:

```
/*
** Multiple read-only sections may be joined into a single
** output section. In this case all of the input sections
** will be visible in the PSV window at the same time.
**
** Total size of the output section is limited to 32K bytes.
*/
psv_set :
{
    *(rdonly1);
    *(rdonly2);
} >program
```

In this example, any label from `rdonly1` or `rdonly2` may be used to determine the correct PSVPAG setting so that both sections are visible at the same time.

© 2008 Microchip Technology Inc.

10.10 STACK ALLOCATION

The 16-bit device dedicates register W15 for use as a software stack pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. Upon power-on or reset, register W15 is initialized to point to a region of memory reserved for the stack. The stack grows upward, towards higher memory addresses.

The 16-bit device also supports stack overflow detection. If the stack limit register SPLIM is initialized, the device will test for overflow on all stack operations. If an overflow should occur, the processor will initiate a stack error exception. By default, this will result in a processor reset. Applications may also install a stack error exception handler by defining an interrupt function named __StackError. See **Section 10.12 "Interrupt Vector Tables"** for details.

By default, 16-bit linker allocates the largest stack possible from unused data memory. The location and size of the stack is reported in the link map output file, under the heading Dynamic Memory Usage. Applications can ensure that at least a minimum sized stack is available by using the `--stack` command option. For example:

```
pic30-ld -o t.exe t1.o --stack=0x100
```

Alternatively, the minimum stack size can be specified in assembly source code:

```
.global STACKSIZE
.equiv STACKSIZE,0x100
```

While performing automatic stack allocation, 16-bit linker increases the minimum required size by a small amount to accomodate the processing of stack overflow exceptions. The stack limit register SPLIM is initialized to point just below this extra space, which acts as a stack overflow guardband. If not enough memory is available for the minimum size stack plus guardband, the linker will report an error.

As an alternative to automatic stack allocation, the stack may be allocated directly with a user-defined section in a custom linker script. In the following example, 0x100 bytes of data memory are reserved for the stack:

```
.stack :
  {
    __SP_init = .;
    . += 0x100;
    __SPLIM_init = .;
    . += 8;
  } > data
```

In the user-defined section, two symbols are declared `__SP_init` and `__SPLIM_init` for use by the start-up module. `__SP_init` defines the initial value for the stack pointer (w15) and `__SPLIM_init` defines the initial value for the stack pointer limit register (SPLIM). The value of `__SPLIM_init` should be at least 8 bytes less than the physical stack limit to allow for stack error exception processing. Note the use of the special symbol '.' in this example. This so-called "dot variable" always contains the current location counter for a given section. For more information, see **Section 9.7.5 "SECTIONS Command"**.

The start-up module uses these symbols to initialize the stack pointer and stack pointer limit register. Normally the start-up module is provided by `libpic30.a` (for C programs) or `crt0.o` (for assembly programs). In special cases, the application may provide its own start-up code. The following stack initialization sequence may be used:

```
mov      #__SP_init,w15    ; initialize w15
mov      #__SPLIM_init,w0  ;
mov      w0,_SPLIM         ; initialize SPLIM
```

10.11 HEAP ALLOCATION

The 16-bit compiler standard C library, `libc.a`, supports dynamic memory allocation functions such as `malloc()` and `free()`. Applications which utilize these functions must instruct the linker to reserve a portion of 16-bit data memory for this purpose. The reserved memory is called a heap.

Applications can specify the heap size by using the `--heap` command option. For example:

```
pic30-ld -o t.exe t1.o --heap=0x100
```

Alternatively, the heap size can be specified in assembly source code:

```
.global HEAPSIZE
.equiv HEAPSIZE,0x100
```

The linker allocates the heap from unused data memory. The heap size is always specified by the programmer. In contrast, the linker sets the stack size to a maximum value, utilizing all remaining data memory.

The location and size of the heap are reported in the link map output file, under the heading Dynamic Memory Usage. If the requested size is not available, the linker reports an error.

10.12 INTERRUPT VECTOR TABLES

dsPIC30F/33F DSC and PIC24F/H MCU devices have two interrupt vector tables - a primary and an alternate table, each containing exception vectors, as well as a `RESET` instruction at location zero. By convention, the linker initializes the `RESET` instruction and interrupt vector tables automatically, using information provided in the standard linker scripts.

The 16-bit compiler provides a special syntax for writing interrupt handlers. See the "*MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User's Guide"* (DS51284) for more information.

Assembly language programmers can install interrupt handlers simply by following the standard naming conventions. Interrupt handlers declared with the standard names are automatically installed into the vector tables.

By convention, the entry point named `__reset` takes control at device reset. All applications written in assembly language must include a reset function with this name. For C programs, the reset function is provided in `libpic30`, which initializes the C run-time environment.

Applications may provide a default interrupt handler, which will be installed into any unused vector table entries. In assembly language, the name of the default interrupt handler is `__DefaultInterrupt`. In C the name is `_DefaultInterrupt`.

If the application does not provide a default interrupt handler, the linker will create one in section `.isr` that contains a `reset` instruction. Creation of a default interrupt handler by the linker may be suppressed with the `--no-isr` option. In that case unused slots in the interrupt vector tables will be filled with zeros.

© 2008 Microchip Technology Inc.

The following example provides a reset function and a default interrupt handler in assembly language. The default interrupt handler uses persistent data storage to keep a count of unexpected interrupts and/or error traps.

```
        .include "p30f6014.inc"
        .text

        .global __reset
__reset:
        ;; takes control at device reset/power-on
        mov   #__SP_init,w15      ; initialize stack pointer
        mov   #__SPLIM_init,w0    ;  and stack limit register
        mov   w0,SPLIM            ;

        btst  RCON,#POR           ; was this a power-on reset?
        bra   z,start             ;  branch if not

        clr   FaultCount          ; else clear fault counter
        bclr  RCON,#POR           ;  and power-on bit
start:
        goto  main                ; start application

        .global __T1Interrupt
__T1Interrupt:
        ;; services timer 1 interrupts
        bclr  IFS0,#T1IF          ; clear the interrupt flag
        retfie                    ;  and return from interrupt

        .global __DefaultInterrupt
__DefaultInterrupt:
        ;; services all other interrupts & traps
        inc  FaultCount           ; increment the fault counter
        reset                     ;  and reset the device

        .section .pbss,persist    ; persistent data storage
        .global FaultCount        ;  is not affected by reset
FaultCount:
        .space 2                  ; count of unexpected interrupts
```

The standard naming convention for interrupt handlers are described in the sections below.

Note: The compiler requires only one leading underscore before any of the interrupt handler names. The assembler requires two leading underscores before any of the interrupt handler names. The compiler format is shown in tables in the following sections.

10.12.1 dsPIC30F DSCs (non-SMPS) Interrupt Vectors

The dsPIC30F SMPS devices are currently dsPIC30F1010, dsPIC30F2020 and dsPIC30F2023. All other dsPIC30F devices are non-SMPS.

TABLE 10-4: INTERRUPT VECTORS – dsPIC30F DSCs (NON-SMPS)

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_ReservedTrap5	_AltReservedTrap5	Reserved
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_IC2Interrupt	_AltIC2Interrupt	IC2 Input capture 2
5	_OC2Interrupt	_AltOC2Interrupt	OC2 Output compare 2
6	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 expired
7	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 expired
8	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 Serial peripheral interface 1
9	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 Receiver
10	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 Transmitter
11	_ADCInterrupt	_AltADCInterrupt	ADC convert completed
12	_NVMInterrupt	_AltNVMInterrupt	NMM NVM write completed
13	_SI2CInterrupt	_AltSI2CInterrupt	Slave I^2C™ interrupt
14	_MI2CInterrupt	_AltMI2CInterrupt	Master I^2C interrupt
15	_CNInterrupt	_AltCNInterrupt	CN Input change interrupt
16	_INT1Interrupt	_AltINT1Interrupt	INT1 External interrupt 0
17	_IC7Interrupt	_AltIC7Interrupt	IC7 Input capture 7
18	_IC8Interrupt	_AltIC8Interrupt	IC8 Input capture 8
19	_OC3Interrupt	_AltOC3Interrupt	OC3 Output compare 3
20	_OC4Interrupt	_AltOC4Interrupt	OC4 Output compare 4
21	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 expired
22	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 expired
23	_INT2Interrupt	_AltINT2Interrupt	INT2 External interrupt 2
24	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 Receiver
25	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 Transmitter
26	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 Serial peripheral interface 2
27	_C1Interrupt	_AltC1Interrupt	CAN1 combined IRQ
28	_IC3Interrupt	_AltIC3Interrupt	IC3 Input capture 3
29	_IC4Interrupt	_AltIC4Interrupt	IC4 Input capture 4
30	_IC5Interrupt	_AltIC5Interrupt	IC5 Input capture 5
31	_IC6Interrupt	_AltIC6Interrupt	IC6 Input capture 6
32	_OC5Interrupt	_AltOC5Interrupt	OC5 Output compare 5
33	_OC6Interrupt	_AltOC6Interrupt	OC6 Output compare 6

© 2008 Microchip Technology Inc.

TABLE 10-4: INTERRUPT VECTORS – dsPIC30F DSCs (NON-SMPS)

IRQ#	Primary Name	Alternate Name	Vector Function
34	_OC7Interrupt	_AltOC7Interrupt	OC7 Output compare 7
35	_OC8Interrupt	_AltOC8Interrupt	OC8 Output compare 8
36	_INT3Interrupt	_AltINT3Interrupt	INT3 External interrupt 3
37	_INT4Interrupt	_AltINT4Interrupt	INT4 External interrupt 4
38	_C2Interrupt	_AltC2Interrupt	CAN2 combined IRQ
39	_PWMInterrupt	_AltPWMInterrupt	PWM period match
40	_QEIInterrupt	_AltQEIInterrupt	QEI position counter compare
41	_DCIInterrupt	_AltDCIInterrupt	DCI CODEC transfer completed
42	_LVDInterrupt	_AltLVDInterrupt	PLVD low voltage detected
43	_FLTAInterrupt	_AltFLTAInterrupt	FLTA MCPWM fault A
44	_FLTBInterrupt	_AltFLTBInterrupt	FLTB MCPWM fault B
45	_Interrupt45	_AltInterrupt45	Reserved
46	_Interrupt46	_AltInterrupt46	Reserved
47	_Interrupt47	_AltInterrupt47	Reserved
48	_Interrupt48	_AltInterrupt48	Reserved
49	_Interrupt49	_AltInterrupt49	Reserved
50	_Interrupt50	_AltInterrupt50	Reserved
51	_Interrupt51	_AltInterrupt51	Reserved
52	_Interrupt52	_AltInterrupt52	Reserved
53	_Interrupt53	_AltInterrupt53	Reserved

10.12.2 dsPIC30F DSCs (SMPS) Interrupt Vectors

The dsPIC30F SMPS devices are currently dsPIC30F1010, dsPIC30F2020 and dsPIC30F2023. All other dsPIC30F devices are non-SMPS.

TABLE 10-5: INTERRUPT VECTORS – dsPIC30F DSCs (SMPS)

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_ReservedTrap5	_AltReservedTrap5	Reserved
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_Interrupt4	_AltInterrupt4	Reserved
5	_OC2Interrupt	_AltOC2Interrupt	OC2 Output compare 2
6	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 expired
7	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 expired
8	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 Serial peripheral interface 1
9	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 Receiver
10	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 Transmitter

TABLE 10-5: INTERRUPT VECTORS – dsPIC30F DSCs (SMPS) (CONTINUED)

IRQ#	Primary Name	Alternate Name	Vector Function
11	_ADCInterrupt	_AltADCInterrupt	ADC Convert completed
12	_NVMInterrupt	_AltNVMInterrupt	NVM write completed
13	_SI2CInterrupt	_AltSI2CInterrupt	Slave I^2C™ interrupt
14	_MI2CInterrupt	_AltMI2CInterrupt	Master I^2C interrupt
15	_Interrupt15	_AltInterrupt15	Reserved
16	_INT1Interrupt	_AltINT1Interrupt	INT1 External interrupt 1
17	_INT2Interrupt	_AltINT2Interrupt	INT2 External interrupt 2
18	_PWMSpEvent MatchInterrupt	_AltPWMSpEvent MatchInterrupt	PWM special event interrupt
19	_PWM1Interrupt	_AltPWM1Interrupt	PWM period match 1
20	_PWM2Interrupt	_AltPWM2Interrupt	PWM period match 2
21	_PWM3Interrupt	_AltPWM3Interrupt	PWM period match 3
22	_PWM4Interrupt	_AltPWM4Interrupt	PWM period match 4
23	_Interrupt23	_AltInterrupt23	Reserved
24	_Interrupt24	_AltInterrupt24	Reserved
25	_Interrupt25	_AltInterrupt25	Reserved
26	_Interrupt26	_AltInterrupt26	Reserved
27	_CNInterrupt	_AltCNInterrupt	Input Change Notification
28	_Interrupt28	_AltInterrupt28	Reserved
29	_CMP1Interrupt	_AltCMP1Interrupt	Analog comparator interrupt 1
30	_CMP2Interrupt	_AltCMP2Interrupt	Analog comparator interrupt 2
31	_CMP3Interrupt	_AltCMP3Interrupt	Analog comparator interrupt 3
32	_CMP4Interrupt	_AltCMP4Interrupt	Analog comparator interrupt 4
33	_Interrupt33	_AltInterrupt33	Reserved
34	_Interrupt34	_AltInterrupt34	Reserved
35	_Interrupt35	_AltInterrupt35	Reserved
36	_Interrupt36	_AltInterrupt36	Reserved
37	_ADCP0Interrupt	_AltADCP0Interrupt	ADC Pair 0 conversion complete
38	_ADCP1Interrupt	_AltADCP1Interrupt	ADC Pair 1 conversion complete
39	_ADCP2Interrupt	_AltADCP2Interrupt	ADC Pair 2 conversion complete
40	_ADCP3Interrupt	_AltADCP3Interrupt	ADC Pair 3 conversion complete
41	_ADCP4Interrupt	_AltADCP4Interrupt	ADC Pair 4 conversion complete
42	_ADCP5Interrupt	_AltADCP5Interrupt	ADC Pair 5 conversion complete
43	_Interrupt43	_AltInterrupt43	Reserved
44	_Interrupt44	_AltInterrupt44	Reserved
45	_Interrupt45	_AltInterrupt45	Reserved
46	_Interrupt46	_AltInterrupt46	Reserved
47	_Interrupt47	_AltInterrupt47	Reserved
48	_Interrupt48	_AltInterrupt48	Reserved
49	_Interrupt49	_AltInterrupt49	Reserved
50	_Interrupt50	_AltInterrupt50	Reserved
51	_Interrupt51	_AltInterrupt51	Reserved
52	_Interrupt52	_AltInterrupt52	Reserved
53	_Interrupt53	_AltInterrupt53	Reserved

© 2008 Microchip Technology Inc.

10.12.3 PIC24F MCUs Interrupt Vectors

The table below specifies the interrupt vectors for these 16-bit devices.

TABLE 10-6: INTERRUPT VECTORS – PIC24F MCUs

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_ReservedTrap5	_AltReservedTrap5	Reserved
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_Interrupt4	_AltInterrupt4	Reserved
5	_IC2Interrupt	_AltIC2Interrupt	IC2 Input capture 2
6	_OC2Interrupt	_AltOC2Interrupt	OC2 Output compare 2
7	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 expired
8	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 expired
9	_SPI1ErrInterrupt	_AltSPI1ErrInterrupt	SPI1 error interrupt
10	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 tranfer completed interrupt
11	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 Receiver
12	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 Transmitter
13	_ADC1Interrupt	_AltADC1Interrupt	ADC 1 convert completed
14	_Interrupt14	_AltInterrupt14	Reserved
15	_Interrupt15	_AltInterrupt15	Reserved
16	_SI2C1Interrupt	_AltSI2C1Interrupt	Slave I2C interrupt 1
17	_MI2C1Interrupt	_AltMI2C1Interrupt	Slave I2C interrupt 1
18	_CompInterrupt	_AltCompInterrupt	Comparator interrupt
19	_CNInterrupt	_AltCNInterrupt	CN Input change interrupt
20	_INT1Interrupt	_AltINT1Interrupt	INT1 External interrupt 1
21	_Interrupt21	_AltInterrupt21	Reserved
22	_Interrupt22	_AltInterrupt22	Reserved
23	_Interrupt23	_AltInterrupt23	Reserved
24	_Interrupt24	_AltInterrupt24	Reserved
25	_OC3Interrupt	_AltOC3Interrupt	OC3 Output compare 3
26	_OC4Interrupt	_AltOC4Interrupt	OC4 Output compare 4
27	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 expired
28	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 expired
29	_INT2Interrupt	_AltINT2Interrupt	INT2 External interrupt 2
30	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 Receiver
31	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 Transmitter
32	_SPI2ErrInterrupt	_AltSPI2ErrInterrupt	SPI2 error interrupt
33	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 tranfer completed interrupt
34	_Interrupt34	_AltInterrupt34	Reserved

TABLE 10-6: INTERRUPT VECTORS – PIC24F MCUs (CONTINUED)

IRQ#	Primary Name	Alternate Name	Vector Function
35	_Interrupt35	_AltInterrupt35	Reserved
36	_Interrupt36	_AltInterrupt36	Reserved
37	_IC3Interrupt	_AltIC3Interrupt	IC3 Input capture 3
38	_IC4Interrupt	_AltIC4Interrupt	IC4 Input capture 4
39	_IC5Interrupt	_AltIC5Interrupt	IC5 Input capture 5
40	_Interrupt40	_AltInterrupt40	Reserved
41	_OC5Interrupt	_AltOC5Interrupt	OC5 Output compare 5
42	_Interrupt42	_AltInterrupt42	Reserved
43	_Interrupt43	_AltInterrupt43	Reserved
44	_Interrupt44	_AltInterrupt44	Reserved
45	_PMPInterrupt	_AltPMPInterrupt	Parallel master port interrupt
46	_Interrupt46	_AltInterrupt46	Reserved
47	_Interrupt47	_AltInterrupt47	Reserved
48	_Interrupt48	_AltInterrupt48	Reserved
49	_SI2C2Interrupt	_AltSI2C2Interrupt	Slave I^2C™ interrupt 2
50	_MI2C2Interrupt	_AltMI2C2Interrupt	Slave I^2C interrupt 2
51	_Interrupt51	_AltInterrupt51	Reserved
52	_Interrupt52	_AltInterrupt52	Reserved
53	_INT3Interrupt	_AltINT3Interrupt	INT3 External interrupt 3
54	_INT4Interrupt	_AltINT4Interrupt	INT4 External interrupt 4
55	_Interrupt55	_AltInterrupt55	Reserved
56	_Interrupt56	_AltInterrupt56	Reserved
57	_Interrupt57	_AltInterrupt57	Reserved
58	_Interrupt58	_AltInterrupt58	Reserved
59	_Interrupt59	_AltInterrupt59	Reserved
60	_Interrupt60	_AltInterrupt60	Reserved
61	_Interrupt61	_AltInterrupt61	Reserved
62	_RTCCInterrupt	_AltRTCCInterrupt	Real-time clock and calender
63	_Interrupt63	_AltInterrupt63	Reserved
64	_Interrupt64	_AltInterrupt64	Reserved
65	_U1EInterrupt	_AltU1EInterrupt	UART1 error interrupt
66	_U2EInterrupt	_AltU2EInterrupt	UART2 error interrupt
67	_CRCInterrupt	_AltCRCInterrupt	Cyclic Redundancy Check
68	_Interrupt68	_AltInterrupt68	Reserved
69	_Interrupt69	_AltInterrupt69	Reserved
70	_Interrupt70	_AltInterrupt70	Reserved
71	_Interrupt71	_AltInterrupt71	Reserved
72	_Interrupt72	_AltInterrupt72	Reserved
73	_Interrupt73	_AltInterrupt73	Reserved
74	_Interrupt74	_AltInterrupt74	Reserved
75	_Interrupt75	_AltInterrupt75	Reserved
76	_Interrupt76	_AltInterrupt76	Reserved
77	_Interrupt77	_AltInterrupt77	Reserved
78	_Interrupt78	_AltInterrupt78	Reserved
79	_Interrupt79	_AltInterrupt79	Reserved

© 2008 Microchip Technology Inc.

TABLE 10-6: INTERRUPT VECTORS – PIC24F MCUs (CONTINUED)

IRQ#	Primary Name	Alternate Name	Vector Function
80	_Interrupt80	_AltInterrupt80	Reserved
81	_Interrupt81	_AltInterrupt81	Reserved
82	_Interrupt82	_AltInterrupt82	Reserved
83	_Interrupt83	_AltInterrupt83	Reserved
84	_Interrupt84	_AltInterrupt84	Reserved
85	_Interrupt85	_AltInterrupt85	Reserved
86	_Interrupt86	_AltInterrupt86	Reserved
87	_Interrupt87	_AltInterrupt87	Reserved
88	_Interrupt88	_AltInterrupt88	Reserved
89	_Interrupt89	_AltInterrupt89	Reserved
90	_Interrupt90	_AltInterrupt90	Reserved
91	_Interrupt91	_AltInterrupt91	Reserved
92	_Interrupt92	_AltInterrupt92	Reserved
93	_Interrupt93	_AltInterrupt93	Reserved
94	_Interrupt94	_AltInterrupt94	Reserved
95	_Interrupt95	_AltInterrupt95	Reserved
96	_Interrupt96	_AltInterrupt96	Reserved
97	_Interrupt97	_AltInterrupt97	Reserved
98	_Interrupt98	_AltInterrupt98	Reserved
99	_Interrupt99	_AltInterrupt99	Reserved
100	_Interrupt100	_AltInterrupt100	Reserved
101	_Interrupt101	_AltInterrupt101	Reserved
102	_Interrupt102	_AltInterrupt102	Reserved
103	_Interrupt103	_AltInterrupt103	Reserved
104	_Interrupt104	_AltInterrupt104	Reserved
105	_Interrupt105	_AltInterrupt105	Reserved
106	_Interrupt106	_AltInterrupt106	Reserved
107	_Interrupt107	_AltInterrupt107	Reserved
108	_Interrupt108	_AltInterrupt108	Reserved
109	_Interrupt109	_AltInterrupt109	Reserved
110	_Interrupt110	_AltInterrupt110	Reserved
111	_Interrupt111	_AltInterrupt111	Reserved
112	_Interrupt112	_AltInterrupt112	Reserved
113	_Interrupt113	_AltInterrupt113	Reserved
114	_Interrupt114	_AltInterrupt114	Reserved
115	_Interrupt115	_AltInterrupt115	Reserved
116	_Interrupt116	_AltInterrupt116	Reserved
117	_Interrupt117	_AltInterrupt117	Reserved

10.12.4 dsPIC33F DSCs / PIC24H MCUs Interrupt Vectors

The table below specifies the interrupt vectors for these 16-bit devices.

TABLE 10-7: INTERRUPT VECTORS – dsPIC33F DSCs/PIC24H MCUs

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_DMACError	_AltDMACError	DMA conflict error trap
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_DMA0Interrupt	_AltDMA0Interrupt	DMA 0 interrupt
5	_IC2Interrupt	_AltIC2Interrupt	IC2 Input capture 2
6	_OC2Interrupt	_AltOC2Interrupt	OC2 Output compare 2
7	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 expired
8	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 expired
9	_SPI1ErrInterrupt	_AltSPI1ErrInterrupt	SPI1 error interrupt
10	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 tranfer completed interrupt
11	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 Receiver
12	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 Transmitter
13	_ADC1Interrupt	_AltADC1Interrupt	ADC 1 convert completed
14	_DMA1Interrupt	_AltDMA1Interrupt	DMA 1 interrupt
15	_Interrupt15	_AltInterrupt15	Reserved
16	_SI2C1Interrupt	_AltSI2C1Interrupt	Slave I2C interrupt 1
17	_MI2C1Interrupt	_AltMI2C1Interrupt	Master I2C interrupt 1
18	_Interrupt18	_AltInterrupt18	Reserved
19	_CNInterrupt	_AltCNInterrupt	CN Input change interrupt
20	_INT1Interrupt	_AltINT1Interrupt	INT1 External interrupt 1
21	_ADC2Interrupt	_AltADC2Interrupt	ADC 2 convert completed
22	_IC7Interrupt	_AltIC7Interrupt	IC7 Input capture 7
23	_IC8Interrupt	_AltIC8Interrupt	IC8 Input capture 8
24	_DMA2Interrupt	_AltDMA2Interrupt	DMA 2 interrupt
25	_OC3Interrupt	_AltOC3Interrupt	OC3 Output compare 3
26	_OC4Interrupt	_AltOC4Interrupt	OC4 Output compare 4
27	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 expired
28	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 expired
29	_INT2Interrupt	_AltINT2Interrupt	INT2 External interrupt 2
30	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 Receiver
31	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 Transmitter
32	_SPI2ErrInterrupt	_AltSPI2ErrInterrupt	SPI2 error interrupt
33	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 tranfer completed interrupt
34	_C1RxRdyInterrupt	_AltC1RxRdyInterrupt	CAN1 receive data ready

© 2008 Microchip Technology Inc.

TABLE 10-7: INTERRUPT VECTORS – dsPIC33F DSCs/PIC24H MCUs

IRQ#	Primary Name	Alternate Name	Vector Function
35	_C1Interrupt	_AltC1Interrupt	CAN1 completed interrupt
36	_DMA3Interrupt	_AltDMA3Interrupt	DMA 3 interrupt
37	_IC3Interrupt	_AltIC3Interrupt	IC3 Input capture 3
38	_IC4Interrupt	_AltIC4Interrupt	IC4 Input capture 4
39	_IC5Interrupt	_AltIC5Interrupt	IC5 Input capture 5
40	_IC6Interrupt	_AltIC6Interrupt	IC6 Input capture 6
41	_OC5Interrupt	_AltOC5Interrupt	OC5 Output compare 5
42	_OC6Interrupt	_AltOC6Interrupt	OC6 Output compare 6
43	_OC7Interrupt	_AltOC7Interrupt	OC7 Output compare 7
44	_OC8Interrupt	_AltOC8Interrupt	OC8 Output compare 8
45	_Interrupt45	_AltInterrupt45	Reserved
46	_DMA4Interrupt	_AltDMA4Interrupt	DMA 4 interrupt
47	_T6Interrupt	_AltT6Interrupt	TMR6 Timer 6 expired
48	_T7Interrupt	_AltT7Interrupt	TMR7 Timer 7 expired
49	_SI2C2Interrupt	_AltSI2C2Interrupt	Slave I^2C™ interrupt 1
50	_MI2C2Interrupt	_AltMI2C2Interrupt	Master I^2C interrupt 2
51	_T8Interrupt	_AltT8Interrupt	TMR8 Timer 8 expired
52	_T9Interrupt	_AltT9Interrupt	TMR9 Timer 9 expired
53	_INT3Interrupt	_AltINT3Interrupt	INT3 External interrupt 3
54	_INT4Interrupt	_AltINT4Interrupt	INT4 External interrupt 4
55	_C2RxRdyInterrupt	_AltC2RxRdyInterrupt	CAN2 receive data ready
56	_C2Interrupt	_AltC2Interrupt	CAN2 completed interrupt
57	_PWMInterrupt	_AltPWMInterrupt	PWM period match
58	_QEIInterrupt	_AltQEIInterrupt	QEI position counter compare
59	_DCIErrInterrupt	_AltDCIErrInterrupt	DCI CODEC error interrupt
60	_DCIInterrupt	_AltDCIInterrupt	DCI CODEC tranfer done
61	_DMA5Interrupt	_AltDMA5Interrupt	DMA channel 5 interrupt
62	_Interrupt62	_AltInterrupt62	Reserved
63	_FLTAInterrupt	_AltFLTAInterrupt	FLTA MCPWM fault A
64	_FLTBInterrupt	_AltFLTBInterrupt	FLTB MCPWM fault B
65	_U1ErrInterrupt	_AltU1ErrInterrupt	UART1 error interrupt
66	_U2ErrInterrupt	_AltU2ErrInterrupt	UART2 error interrupt
67	_Interrupt67	_AltInterrupt67	Reserved
68	_DMA6Interrupt	_AltDMA6Interrupt	DMA channel 6 interrupt
69	_DMA7Interrupt	_AltDMA7Interrupt	DMA channel 7 interrupt
70	_C1TxReqInterrupt	_AltC1TxReqInterrupt	CAN1 transmit data request
71	_C2TxReqInterrupt	_AltC2TxReqInterrupt	CAN2 transmit data request
72	_Interrupt72	_AltInterrupt72	Reserved
73	_Interrupt73	_AltInterrupt73	Reserved
74	_Interrupt74	_AltInterrupt74	Reserved
75	_Interrupt75	_AltInterrupt75	Reserved
76	_Interrupt76	_AltInterrupt76	Reserved
77	_Interrupt77	_AltInterrupt77	Reserved
78	_Interrupt78	_AltInterrupt78	Reserved
79	_Interrupt79	_AltInterrupt79	Reserved

TABLE 10-7: INTERRUPT VECTORS – dsPIC33F DSCs/PIC24H MCUs

IRQ#	Primary Name	Alternate Name	Vector Function
80	_Interrupt80	_AltInterrupt80	Reserved
81	_Interrupt81	_AltInterrupt81	Reserved
82	_Interrupt82	_AltInterrupt82	Reserved
83	_Interrupt83	_AltInterrupt83	Reserved
84	_Interrupt84	_AltInterrupt84	Reserved
85	_Interrupt85	_AltInterrupt85	Reserved
86	_Interrupt86	_AltInterrupt86	Reserved
87	_Interrupt87	_AltInterrupt87	Reserved
88	_Interrupt88	_AltInterrupt88	Reserved
89	_Interrupt89	_AltInterrupt89	Reserved
90	_Interrupt90	_AltInterrupt90	Reserved
91	_Interrupt91	_AltInterrupt91	Reserved
92	_Interrupt92	_AltInterrupt92	Reserved
93	_Interrupt93	_AltInterrupt93	Reserved
94	_Interrupt94	_AltInterrupt94	Reserved
95	_Interrupt95	_AltInterrupt95	Reserved
96	_Interrupt96	_AltInterrupt96	Reserved
97	_Interrupt97	_AltInterrupt97	Reserved
98	_Interrupt98	_AltInterrupt98	Reserved
99	_Interrupt99	_AltInterrupt99	Reserved
100	_Interrupt100	_AltInterrupt100	Reserved
101	_Interrupt101	_AltInterrupt101	Reserved
102	_Interrupt102	_AltInterrupt102	Reserved
103	_Interrupt103	_AltInterrupt103	Reserved
104	_Interrupt104	_AltInterrupt104	Reserved
105	_Interrupt105	_AltInterrupt105	Reserved
106	_Interrupt106	_AltInterrupt106	Reserved
107	_Interrupt107	_AltInterrupt107	Reserved
108	_Interrupt108	_AltInterrupt108	Reserved
109	_Interrupt109	_AltInterrupt109	Reserved
110	_Interrupt110	_AltInterrupt110	Reserved
111	_Interrupt111	_AltInterrupt111	Reserved
112	_Interrupt112	_AltInterrupt112	Reserved
113	_Interrupt113	_AltInterrupt113	Reserved
114	_Interrupt114	_AltInterrupt114	Reserved
115	_Interrupt115	_AltInterrupt115	Reserved
116	_Interrupt116	_AltInterrupt116	Reserved
117	_Interrupt117	_AltInterrupt117	Reserved

© 2008 Microchip Technology Inc.

10.13 OPTIMIZING MEMORY USAGE

For memory intensive applications, it is often necessary to optimize memory usage by reducing or eliminating any unused gaps. The linker will optimize memory allocation automatically in most cases. However, certain constructs in source code and/or linker scripts may introduce gaps and should be avoided.

Memory gaps generally fall into four categories:

- Gaps Between Variables of Different Types
- Gaps Between Aligned Variables
- Gaps Between Input Sections
- Gaps Between Output Sections

10.13.1 Gaps Between Variables of Different Types

Gaps may be inserted between variables of different types to satisfy address alignment requirements. For example, the following sequence of C statements will result in a gap:

```
char c1;
int i;
char c2;
int j;
```

Because the processor requires integers to be aligned on a 16-bit boundary, a padding byte was inserted after variables `c1` and `c2`. To eliminate this padding, variables of the same type should be defined together, as shown:

```
char c1,c2;
int i,j;
```

Gaps between variables are not visible to the linker, and are not reported in the link map. To detect these gaps, an assembly listing file must be created. The following procedure can be used:

1. If the source file is written in C, specify the `-save-temps` command line option to the compiler. This will cause an assembly version of the source file to be saved in *`filename`*`.s`.

   ```
   pic30-gcc test.c -save-temps
   ```

2. Specify the `-ai` listing option to the assembler. This will cause a table of section information to be generated.

   ```
   pic30-as test.s -ai
   ```

```
SECTION INFORMATION:

Section        Length (PC units)   Length (bytes) (dec)
-------        -----------------   --------------------
.text                          0                0  (0)

TOTAL PROGRAM MEMORY USED (bytes):              0  (0)

Section           Alignment Gaps   Length (bytes) (dec)
-------           --------------   --------------------
.data                          0                0  (0)
.bss                           0                0  (0)
.nbss                        0x2              0x8  (8)

   TOTAL DATA MEMORY USED (bytes):            0x8  (8)
```

In this example, 2 bytes of unused memory were inserted into section .nbss. Gaps between ordinary C variables will not exceed 1 byte per variable.

10.13.2 Gaps Between Aligned Variables

Variables may be defined in C with the aligned attribute in order to specify special alignment requirements for modulo addressing or other purposes. Use of the aligned attribute will cause the variable to be allocated in a unique section. Since a unique section is never combined with other input sections, no alignment padding is necessary and the linker will allocate memory for the aligned variable in the most efficient way possible.

For example, the following sequence of C statements will not result in an alignment gap, because variable buf is allocated in a unique section automatically:

```
char c1,c2;
int i,j;
int __attribute__((aligned(256))) buf[128];
```

When allocating space for aligned variables in assembly language, the source code must also specify a section name. Unless the aligned variable is defined in a unique section, alignment padding may be inserted. For example, the following sequence of assembly statements would result in a large alignment gap, and should be avoided:

```
       .section my_vars,bss
       .global _var1,_var2,_buf
_var1: .space 2
_var2: .space 2
                    ; location counter is now 4
       .align 256
_buf:  .space 256
                    ; location counter is now 512
```

Re-ordering the statements so that _buf is defined first will not eliminate the gap. A named input section will be padded so that its length is a multiple of the requested alignment. This is necessary in order to guarantee correct alignment when multiple input sections with the same name are combined by the linker. Therefore reordering statements would cause the gap to move, but would not eliminate the gap.

Aligned variables in assembly must be defined in a unique section in order to avoid alignment padding. It is not sufficient to specify a section name that is used only once, because the assembler does not know if that section will be combined with others by the linker. Instead, the special section name * should be used. As explained in **Section 6.3 "Directives that Define Sections"** the section name * instructs the assembler to create a unique section that will not be combined with other sections.

To avoid alignment gaps, the previous example could be written as:

```
       .section my_vars,bss
       .global _var1,_var2
_var1: .space 2
_var2: .space 2

       .section *,bss
       .global _buf
       .align 256
_buf:  .space 256
```

The alignment requirement for _buf could also be specified in the .section directive, as shown:

```
       .section *,bss,align(256)
       .global _buf
_buf:  .space 256
```


© 2008 Microchip Technology Inc.

10.13.3 Gaps Between Input Sections

Gaps between input sections are similar to gaps between aligned variables, except that the padding is inserted by the linker, not the assembler. This type of gap can occur when variables with different alignment requirements are defined in separate source files.

A necessary condition for the insertion of alignment gaps by the linker is explicit mapping of input sections in the linker script. For example, older versions of the 16-bit compiler (prior to version 1.30) included the following definition:

```
/*
** Initialized Data and Constants
*/
.data :
  {
      *(.data);
      *(.dconst);
  } >data
```

This example maps all input sections named `.data`, and all input sections named `.dconst`, into a single output section. The various input sections will be combined sequentially. If the alignment requirement of any section exceeds that of the previous section, the linker will insert padding as needed and report an alignment gap in the link map:

```
Data Memory Usage

section         address      alignment gaps    total length  (dec)
-------         -------      --------------    --------------------
.data             0x800                0x10             0x90  (144)

             Total data memory used (bytes):             0x90  (144) <1%
```

The remedy for this type of gap is to simply eliminate the mapping of input sections in linker scripts. Unmapped sections are allocated individually by the linker, so that no special alignment padding is necessary. Newer versions of the 16-bit compiler (version 1.30 and later) do not explicitly map any input sections in data memory for this reason.

10.13.4 Gaps Between Output Sections

Gaps between output sections can occur when the alignment requirements differ and multiple sections are allocated sequentially into the same memory region.

A necessary condition for the insertion of alignment gaps between output sections is explicit mapping of output sections in the linker script. For example, older versions of the 16-bit compiler (prior to version 1.30) included the following definitions:

```
/*
** Persistent Data
*/
.pbss (NOLOAD):
  {
      *(.pbss);
  } >data

/*
** Static Data
*/
.bss (NOLOAD):
  {
      *(.bss);
  } >data
```

This example creates two output sections (`.pbss` and `.bss`) and maps them into memory region `data`. Because the output sections are allocated sequentially, any difference in alignment requirements will result in gap.

In some instances the linker will make use of this gap, depending on the availability, size, and alignment requirements of any unmapped sections. In general it is preferable to eliminate the explicit mapping of output sections in linker scripts. When all output sections are unmapped, the linker is free to perform a best-fit allocation based on section attributes.

Explicit mapping of output sections in linker scripts is recommended only when the proximity or relative ordering of sections is important, and can't be satisfied using the section attributes described in **Section 6.3 "Directives that Define Sections"**.

10.14 BOOT AND SECURE SEGMENTS

The linker supports boot, secure, and general segments as described in the "*CodeGuard™ Security Reference Manual"* (DS70180). The security model which includes segment sizes and configuration options may be specified in multiple ways. The linker allocates memory according to this security model and supports independant linking of application segments.

10.14.1 Specifying the Security Model

The application security model (including the sizes of various secure segments in FLASH, RAM, and EEDATA) can be specified in two ways:

1. In source code using macros currently defined for the FBS, FSS, FGS configuration words. See processor-specific include files for details and examples.
2. Using linker command options (see **Section 8.8 "Options that Specify CodeGuard™ Security Features"**).

If both methods are used to provide conflicting information, the linker will issue a diagnostic. Likewise, a diagnostic will be issued if a security model is specified that can not be supported by the target device. The security model will be encoded by the linker into the executable file as contents for the FBS, FSS, and FGS configuration words.

A summary of CodeGuard Security options and segment sizes is written to the link map file.

10.14.2 User-Defined Boot and Secure Segments

User-defined `boot` and `secure` segments are supported in program memory and data memory. This allows an application to take advantage of the CodeGuard Security language extensions on any device, not just CodeGuard Security-enabled devices. User-defined segments are specified with the `ram_size` and `flash_size` options (see **Section 8.8 "Options that Specify CodeGuard™ Security Features"**).

User-defined segment options should not be combined with CodeGuard Security options. They are intended for debugging and/or special bootloader applications. User-defined segment options are not encoded in the FBS, FSS, FGS configuration words.

10.14.3 Boot and Secure Segment Allocation

The linker will collect input sections designated as boot or secure and allocate them according to the security model. Diagnostics will be issued for errors such as overflow of a secure segment, or requests for a type of protected memory that does not match the security model.

© 2008 Microchip Technology Inc.

Only sections explicitly designated as `boot` or `secure` will be allocated in the `boot` and `secure` segments. For independantly linked applications, `boot` and `secure` functions must not call any library functions, or have any section dependencies that are not explictly designated as `boot` or `secure`.

If access entry points have been defined, the linker will construct branch tables as needed for the boot or secure segment. Branch tables fill the entire access area (32 instruction words), regardless of how many access entry slots are actually used. This ensures that secure segment object code can be reached only by access entry point. Unused slots in the branch table will be filled with the default entry if one has been specified.

Execution flow may reach access entry points in several different ways, using a combination of machine instructions and data directives. Each access entry consists of a single, unconditional branch instruction, which targets the actual object code for a secure function.

10.14.4 Resolving Symbols

Symbol references within CodeGuard Security segments, and between CodeGuard Security segments, will be processed normally. If access entry points have been specified in a code address reference or in a function call reference, they will be resolved to specific offsets in the access entry tables. This mechanism allows the linker to resolve references to `boot` or `secure` functions that are defined only in terms of their access entry slot number, and is the key to supporting independently-linked applications.

Interrupt service routines designated for the `boot` or `secure` segments will be installed as a vector in slot 16 of the appropriate segment. Unused slots in the access entry tables are resolved to the `unused` function handler if one has been defined.

Note:	The linker implements the security model in terms of memory allocation, but does not enforce a security policy. For example, references to a function defined in a secure segment from a lower privileged segment are permitted. Therefore it is possible to successfully link an application that fails at runtime due to CodeGuard Security hardware protection. This should be a relatively uncommon occurrence, since in practice strict CodeGuard Security protection implies independently-linked application segments.

10.15 NOTABLE SYMBOLS

The following symbols are defined by the linker and may be useful in code development.

__C30_UART

Description:	Constant that defines the default UART.
Include:	`libpic30.h`
Prototype:	`int __C30_UART;`
Remarks:	Defines the default UART that `read()` and `write()` will use for `stdin` (unless a file has been attached), `stdout`, and `stdout`.
Default Behavior:	By default, or with a value of 1, UART 1 will be used. Otherwise UART 2 will be used. `read()` and `write()` are the eventual destinations of the C standard I/O functions.
Example:	See *16-Bit Language Tool Libraries* (DS51456), “Support Functions” chapter.

_DMA_BASE

Description: A symbol defined in data memory for devices with DMA support. This symbol marks the starting address of dual-ported memory.

Include: p33Fxxx.h, p24Hxxx.h

Prototype: void *_DMA_BASE

Remarks: In C, the symbol should be referenced with the address operator (&). In assembly language, it should be referenced with an extra underbar character in the prefix.

Default Behavior: The starting address of dual-ported memory.

Examples:

C code:

```
unsigned int addr = (unsigned int) &_DMA_BASE;
```

Assembly code:

```
mov #__DMA_BASE, w0

.word __DMA_BASE
```

_PROGRAM_END

Description: A symbol defined in program memory to mark the highest address used by a CODE or PSV section.

Include: libpic30.h

Prototype: __attribute__((space(prog))) int _PROGRAM_END

Remarks: In C, the symbol should be referenced with the address operator (&), as in a built-in function call that accepts the address of an object in program memory. Also, this symbol can be used by applications as an end point for checksum calculations.
In assembly language, it should be referenced with an extra underbar character in the prefix.

Default Behavior: The highest address used by a CODE or PSV section.

Examples:

C code:

```
__builtin_tblpage(&_PROGRAM_END)
__builtin_tbloffset(&_PROGRAM_END)

_prog_addressT big_addr;
_init_prog_address(big_addr, &_PROGRAM_END)
```

Assembly code:

```
mov #tblpage(__PROGRAM_END),w0
mov #tbloffset(__PROGRAM_END),w1

.pword __PROGRAM_END
.long __PROGRAM_END
```


© 2008 Microchip Technology Inc.

Chapter 11. Linker Examples

11.1 INTRODUCTION

The 16-bit devices include many architectural features that require special handling by the linker. The 16-bit compiler and assembler each provide a syntax than can be used to designate certain elements of an application for special handling. In C, a rich set of attributes are available to modify variable and function definitions (see the "*MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs User's Guide"* DS51284). In assembly language, variables and functions are abstracted into memory sections, which become inputs to the linker. The assembler provides another set of attributes that are available to modify section definitions (see **Section 6.8 "Directives that Modify Section Alignment"**).

This chapter includes a number of 16-bit specific linker examples and shows the equivalent syntax in C and assembly language.

11.2 HIGHLIGHTS

Topics covered in this chapter are:

- Memory Addresses and Relocatable Code
- Locating a Variable at a Specific Address
- Locating a Function at a Specific Address
- Saving and Restoring the PSVPAG Register
- Locating a Constant at a Specific Address in Program Memory
- Locating and Accessing Data in EEPROM Memory
- Creating an Incrementing Modulo Buffer in X Memory
- Creating a Decrementing Modulo Buffer in Y Memory
- Locating the Stack at a Specific Address
- Locating and Reserving Program Memory

11.3 MEMORY ADDRESSES AND RELOCATABLE CODE

For most applications it is preferable to write fully relocatable source code, thus allowing the linker to determine the exact addresses in memory where functions and variables are placed. The final address of external symbols in data memory and program memory can be determined from the link map output, as shown in this excerpt:

```
...
External Symbols in Data Memory (by address):

                0x0802                  __curbrk
                0x0804                  _Stdin
                0x082c                  _Stdout
                0x0854                  _Stderr
                0x087c                  _Files
                0x088c                  _Aldata
                0x0890                  _Size_block
...
External Symbols in Data Memory (by name):

                0x0802                  __curbrk
                0x088c                  _Aldata
                0x087c                  _Files
                0x0890                  _Size_block
                0x0854                  _Stderr
                0x0804                  _Stdin
                0x082c                  _Stdout
...
```

In some cases it is necessary for the programmer to specify the address where a certain variable or function should be located. Traditionally this is done by creating a user-defined section and writing a custom linker script. The 16-bit assembler and compiler provide a set of attributes that can be used to specify absolute addresses and memory spaces directly in source code. When these attributes are used, custom linker scripts are not required.

Note: By specifying an absolute address, the programmer assumes the responsibilty to ensure the specified address is reasonable and available. If the specified address is out of range, or conflicts with a statically allocated resource such as section `.text` in program memory, a link error will occur. Often it is useful to first build an application without specifying an absolute address, so that the resulting memory map can be examined. A summary of memory usage by the linker appears in the link map, and may also be written to the console with the `--report-mem` option.

© 2008 Microchip Technology Inc.

11.4 LOCATING A VARIABLE AT A SPECIFIC ADDRESS

In this example, array `buf1` is located at a specific address in data memory. The address of `buf1` can be confirmed by executing the program in the simulator, or by examining the link map.

```
#include "stdio.h"
int __attribute__((address(0x900))) buf1[128];
void main()
{
  printf("0x900 = 0x%x\n", &buf1);
}
```

The equivalent array definition in assembly language appears below. The `.align` directive is optional and represents the default alignment in data memory. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
        .section *,address(0x900),bss,near
        .global  _buf1
        .align   2
_buf1:  .space   256
```

11.5 LOCATING A FUNCTION AT A SPECIFIC ADDRESS

In this example, function `func` is located at a specific address. Two built-in compiler functions are used to calculate the program memory address, which is not otherwise available in C.

```
#include "stdio.h"
void __attribute__((address(0x2000))) func()
{}
void main()
{
  long addr;

  addr = ((long) __builtin_tblpage(func) << 16)
         + __builtin_tbloffset(func);
  printf("0x2000 = 0x%lx\n", addr);
}
```

The equivalent function definition in assembly language appears below. The `.align` directive is optional and represents the default alignment in program memory. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
        .section *,address(0x2000),code
        .global  _func
        .align   2
_func:  return
```

11.6 SAVING AND RESTORING THE PSVPAG REGISTER

In this example, the constant `status_string` is located in the compiler-managed PSV section, while the constant `gamma_factor` is located in a separate PSV section. You are required to place a constant in a separate PSV section if the compiler-managed PSV section is full or if you want to locate the constant at a specific address. A PSV section has a capacity of 32 Kbytes.

The compiler will initialize the PSV bit in the CORCON register and the PSVPAG register only for the compiler-managed PSV section (`auto_psv`) on start-up (in `crt0.s`). To properly access `gamma_factor`, you must manually manage the PSVPAG register. Namely, save the PSVPAG register before accessing `gamma_factor`, set the PSVPAG to access `gamma_factor` and restore the original PSVPAG after accessing `gamma_factor`. To determine the PSVPAG of a constant stored in program memory, one can use the `__builtin_psvpage()` helper function.

When the PSVPAG is modified to access `gamma_factor`, be careful not to access constants stored in the compiler-managed PSV section, such as string constants used with `printf()`. Any attempts to access constants stored in the compiler-managed PSV section with an incorrect PSVPAG will fail.

Note: On devices with less than 16K instruction words, there is only one PSVPAG and manual mangement of the PSVPAG register is not required.

```
#include "stdio.h"
#include "p30fxxxx.h"

const char __attribute__ ((space(auto_psv))) status_string[2][10] =
{"System OK", "Key Made"};
const int __attribute__ ((space(psv))) gamma_factor[3] = {13, 23, 7};

int main(void)
{
 unsigned psv_shadow;
 unsigned key, seed = 17231;

 /* print the first status string */
 printf ("%s\n", status_string[0]);

 /* save the PSVPAG */
 psv_shadow = PSVPAG;

 /* set the PSVPAG for accessing gamma_factor[] */
 PSVPAG = __builtin_psvpage (gamma_factor);

 /* build the key from gamma_factor */
 key = (seed + gamma_factor[0] + gamma_factor[1]) / gamma_factor[2];

 /* restore the PSVPAG for the compiler-managed PSVPAG */
 PSVPAG = psv_shadow;

 /* print the second status message */
 printf ("%s \n", status_string[1]);
}
```


© 2008 Microchip Technology Inc.

11.7 LOCATING A CONSTANT AT A SPECIFIC ADDRESS IN PROGRAM MEMORY

In this example, the constant `table` is located at a specific address in program memory. When a constant is specifically placed at an address in program memory, it must be placed in its own PSV section using the `space(psv)` attribute. If a device has only one PSV page (16K instruction words or less), the (`psv`) section and (`auto_psv`) section will share the same PSV page by default.

> **Note:** It is not possible to place a constant at a specific address in Program Memory using the `space(auto_psv)` attribute. Only the `space(psv)` attribute may be used to perform this task.

The `__builtin_tblpage()` and `__builtin_tbloffset()` helper functions can be used to find the address of a constant stored in program memory. As with the example in the previous section, if you want to access `table`, you must manually manage PSVPAG and restore its original value. The `__builtin_psvpag()` helper function may be used to assist with this task.

```
#include "stdio.h"
#include "p30fxxxx.h"

const unsigned __attribute__ ((space(psv), address (0x2000)))
table[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int main(void)
{
 unsigned psv_shadow;
 unsigned sum=0, u;
 long addr;

/* compute the address of table and print it */
addr = ((long) __builtin_tblpage(table) << 16) +
__builtin_tbloffset(table);

 /* print the address of table */
 printf ("table[] is stored at address 0x%lx\n", addr);

 /* save the PSVPAG */
 psv_shadow = PSVPAG;

 /* set the PSVPAG for accessing table[] */
 PSVPAG = __builtin_psvpage (table);

 /* sum the values in table[] */
 for (u=0; u<10; u++) {
      sum += table[u];
     }

 /* restore the PSVPAG for the compiler-managed PSVPAG */
 PSVPAG = psv_shadow;

 /* print the sum */
 printf ("sum is %d\n", sum);
}
```

The equivalent constant definition for the array `table` in assembly language appears below. The `.align` directive is optional and represents the default alignment in program memory. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
        .section *,address(0x2000),psv
        .global  _table
        .align   2
_table:
        .word   0,1,2,3,4,5,6,7,8,9
```

In order to allocate `table` in data memory, the `space(psv)` attribute could be changed to `space(data)`. In this case, the specified address would be a data memory address. In the absence of a `space` attribute, the keyword `const` directs the C compiler to allocate the variable in the same space as other compiler constants. Constants are allocated in program memory by default, or in data memory if the constants-in-data memory model is selected.

11.8 LOCATING AND ACCESSING DATA IN EEPROM MEMORY

In this example, two arrays are defined in data EEPROM. `Table1` is aligned to a 32-bit address, so it will be eligible for erasing or programming using the row programming algorithm. `Table2` is defined with standard alignment, so it must be erased or programmed one word at a time. The macro `_EEDATA` is used to place a variable in the Data EEPROM section of memory and align the variable to the specified byte boundary. This macro is defined in the processor header files for devices which contain data flash. This example is targeted for the dsPIC30F6014 processor, and includes the processor header file `p30f6014.h`.

The compiler and linker treat Data EEPROM like any other custom-defined (`psv`) section. The compiler will not set the PSV bit in the CORCON register or set the PSVPAG to access Data EEPROM; you must perform these tasks manually. Once the PSV bit and PSVPAG are set for Data EEPROM, the compiler can freely access any variables stored there. As this example shows, you can use the `__builtin_psvpage()` and `__builtin_psvoffset()` helper functions for variables stored in Data EEPROM.

If an application stores constants in the compiler-managed PSV section of program memory (`auto_psv`) and Data EEPROM, the compiler will set the PSV bit in the CORCON register and set PSVPAG for the compiler-managed PSV section on start-up (in `crt0.s`). You must manually modify PSVPAG to access Data EEPROM using the technique shown in **Section 11.6 "Saving and Restoring the PSVPAG Register"**. Namely, the original PSVPAG must be saved before the access, and then restored after the access.

Note 1:	When placing variables in Data EEPROM, the `const` qualifier is optional.
2:	The `address()` attribute can be used to locate variables in Data EEPROM to a specific address.

© 2008 Microchip Technology Inc.

```
/* load SFR definitions and macros */
#include "p30f6014.h"

/* load standard I/O definitions */
#include "stdio.h"

unsigned int _EEDATA(32) Table1[16]={0};

unsigned int _EEDATA(2) Table2[4]=
{0x1234, 0x5678, 0x9ABC, 0xDEF0};

unsigned int i,*ee_rd_ptr;
unsigned int temp_data[4];

int main( void )
{
  /* enable the PSV window since no program constants stored
     in program memory */
  CORCONbits.PSV = 1;

  /* set the PSV page using a built-in function */
  PSVPAG = __builtin_psvpage(&Table2);

  /* initialize EEPROM read pointer using a built-in function */
  ee_rd_ptr = (unsigned int *) __builtin_psvoffset(&Table2);

  /* read integer data from EEPROM */
  temp_data[0] = *ee_rd_ptr++;
  temp_data[1] = *ee_rd_ptr++;
  temp_data[2] = *ee_rd_ptr++;
  temp_data[3] = *ee_rd_ptr;
}
```

The equivalent array definitions for `Table1` and `Table2` in assembly language appear below. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
        .global _Table1
        .section  *,eedata
        .align  32
_Table1:
        .space 32

        .global _Table2
        .section  *,eedata
        .align  2
_Table2:
        .word   0x1234
        .word   0x5678
        .word   0x9ABC
        .word   0xDEF0
```

11.9 CREATING AN INCREMENTING MODULO BUFFER IN X MEMORY

An incrementing modulo buffer for use in assembly language can be easily defined in C. In this example, the macro `_XBSS` is used to define an array whose memory alignment is the smallest power of two that is greater than or equal to its size. `_XBSS` is defined in the processor header file, which in this example is `p30f6014.h`.

```
#include "p30f6014.h"
#include "stdio.h"

int _XBSS(128) xbuf[50];

void main()
{
  printf("Should be zero: %x\n", (int) &xbuf % 128);
}
```

The equivalent definition in assembly language appears below. The section alignment could have specified with a separate `.align` directive. By using `*` as a section name, the linker is afforded maximum flexibility to allocate memory.

```
        .global _xbuf
        .section  *,xmemory,bss,align(128)
_xbuf:  .space 100
```

11.10 CREATING A DECREMENTING MODULO BUFFER IN Y MEMORY

A decrementing modulo buffer for use in assembly language can be easily defined in C. In this case, the ending address `+1` of the array must be aligned. There is not a suitable predefined macro in the processor header files for this purpose, so variable attributes are specified directly. The `far` attribute is recommended because Y memory does not fall within the near space on all devices, and the compiler uses a small-data memory model by default.

```
#include "stdio.h"

int __attribute__((space(ymemory), far, reverse(128))) ybuf[50];

void main()
{
  printf("Should be zero: %x\n",
         ((int) &ybuf + sizeof(ybuf)) % 128);
}
```

Note 1: The `reverse()` attribute can be used with constants stored in program memory only if they are located in a psv section, not the compiler-managed `auto_psv` section.

2: The `reverse()` attribute can be used with constants stored in Data EEPROM memory.

The equivalent definition in assembly language appears below. Reverse section alignment can only be specified as an argument to the `.section` directive.

```
        .global _ybuf
        .section  *,ymemory,reverse(128)
_ybuf:  .space 100
```


© 2008 Microchip Technology Inc.

11.11 LOCATING THE STACK AT A SPECIFIC ADDRESS

By default, the linker allocates a maximum-size stack using the largest unused block of data memory. In cases where it is necessary for the programmer to specify the location and size of the stack explicitly, a custom linker script can be used to allocate the stack from a user-defined section.

The following text could be added to a custom linker script to allocate the stack. In this example, 0x100 bytes of memory are reserved for the stack, to be allocated at adress 0x1800. Two symbols are declared, `__SP_init` and `__SPLIM_init`, for use by the C run-time start-up module. The programmer is responsible to specify an address that does not conflict with other statically allocated resources, such as the MPLAB ICD 2 RAM buffer at 0x800.

```
.stack 0x1800 :
{
  __SP_init = .;
  . += 0x100;
  __SPLIM_init = .;
  . += 8;
} >data
```

`__SP_init` defines the initial value for the stack pointer (W15) and `__SPLIM_init` defines the initial value for the stack pointer limit register (SPLIM). Assembly programmers can use these symbols to initialize W15 and SPLIM at the beginning of the `__reset` function.

Notice that 8 additional bytes are reserved after the defintion of `__SPLIM_init`. This allocates additional space for stack error processing. This value should be increased further to account for stack usage by the interrupt handler itself, if a stack error interrupt handler is installed. The default interrupt handler does not require additional stack usage.

11.12 LOCATING AND RESERVING PROGRAM MEMORY

In this example, a block of program memory is reserved for a special purpose, such as a bootloader. An arbitrary sized function is allocated in the block, with the remaining space reserved for expansion or other purposes.

The following output section definition is added to a custom linker script:

```
BOOT_START = 0xA200;
BOOT_LEN = 0x400;

my_boot BOOT_START :
{
 *(my_boot);
 . = BOOT_LEN;  /* advance dot to the maximum length */
} > program
```

Note the "dot assignment" (`.=`) that appears inside the section definition after the input sections. Dot is a special variable that represents the location counter, or next fill point, in the current section. It is an offset relative to the start of the section. The statement in effect says "no matter how big the input sections are, make sure the output section is full size."

The following C function will be allocated in the reserved block:

```
void __attribute__((section("my_boot"))) func1()
{
  /* etc. */
}
```

The equivalent assembly language would be:

```
   .section my_boot,code
   .global _func1
_func1:
   ; and so on..
   return
```

If the bootloader is allocated at the start of program memory, a custom linker script is not be required. Instead, the function could be defined with attribute boot. For example:

```
void __attribute__((boot)) func1()
{
  /* and so on.. */
}
```

The equivalent definition in assembly language:

```
   .section *,code,boot
   .global _func1
_func1:
   ; and so on..
   return
```

In this case, program memory will be automatically reserved by specifying a Code-Guard Security™ boot segment in FBS configuration word settings, or by specifying a user-defined boot segment with linker command option. See **Section 10.14 "Boot and Secure Segments"** for more information.

© 2008 Microchip Technology Inc.

MPLAB® ASSEMBLER, LINKER AND UTILITIES FOR PIC24 MCUs AND dsPIC® DSCs USER'S GUIDE

Part 3 – MPLAB Object Archiver/Librarian for PIC24 MCUs and dsPIC® DSCs

Chapter 12. MPLAB Object Archiver/Librarian for 16-bit Devices.....179

NOTES:

© 2008 Microchip Technology Inc.

Chapter 12. MPLAB Object Archiver/Librarian for 16-bit Devices

12.1 INTRODUCTION

The MPLAB Object Archiver/Librarian for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB LIB30) creates, modifies and extracts files from archives. This tool is one of several utilities (`pic30-ar`). An "archive" is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called "members" of the archive).

The original files' contents, mode (permissions), timestamp, owner and group are preserved in the archive, and can be restored on extraction.

The 16-bit archiver/librarian can maintain archives whose members have names of any length; however, if an `f` modifier is used, the file names will be truncated to 15 characters.

The archiver is considered a binary utility because archives of this sort are most often used as "libraries" holding commonly needed subroutines.

The archiver creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier `s`. Once created, this index is updated in the archive whenever the archiver makes a change to its contents (save for the `q` update operation). An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

You may use `nm -s` or `nm --print-armap` to list this index table. If an archive lacks the table, another form of the 16-bit archiver/librarian called `ranlib` can be used to add only the table.

The 16-bit archiver/librarian is designed to be compatible with two different facilities. You can control its activity using command line options or, if you specify the single command line option `-M`, you can control it with a script supplied via standard input.

12.2 HIGHLIGHTS

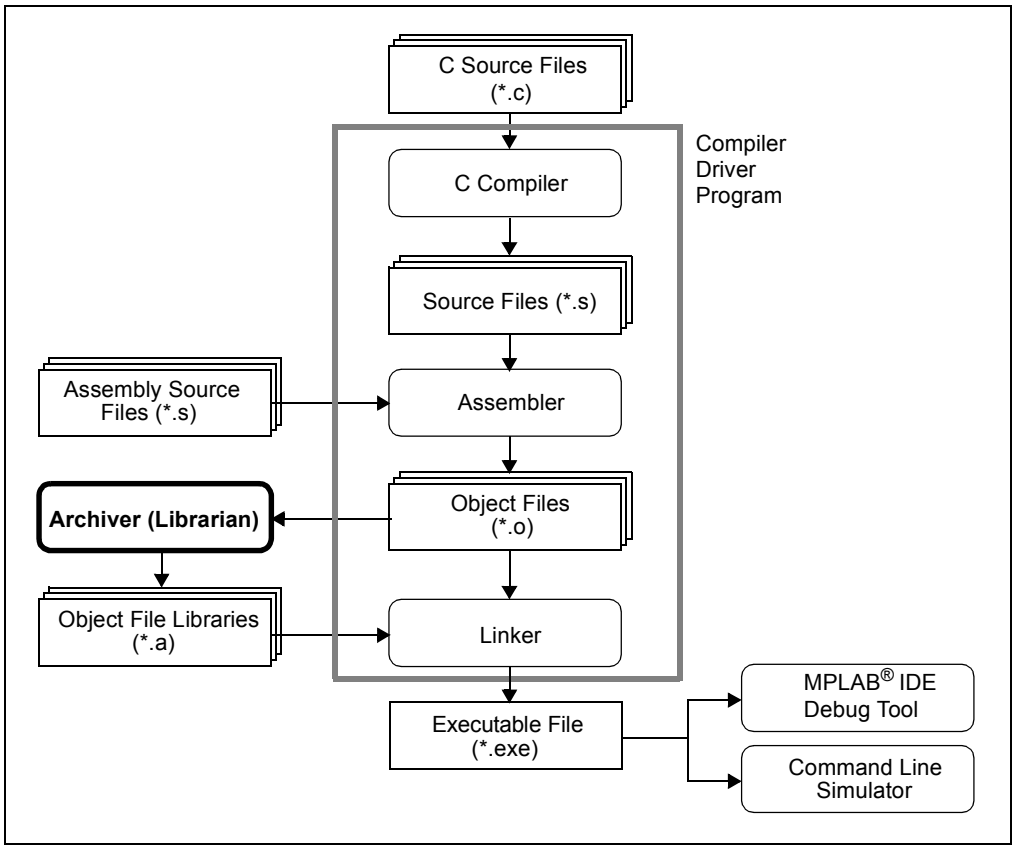
Topics covered in this chapter are:

- Arichiver/Libraian and Other Development Tools
- Feature Set
- Input/Output Files
- Syntax
- Options
- Scripts

12.3 ARICHIVER/LIBRAIAN AND OTHER DEVELOPMENT TOOLS

The 16-bit librarian creates an archive file from object files created by the 16-bit assembler. Archive files may then be linked by the 16-bit linker with other relocatable object files to create an executable file. See Figure 12-1 for an overview of the tools process flow.

FIGURE 12-1: TOOLS PROCESS FLOW



12.4 FEATURE SET

Notable features of the assembler include:

- Available for Windows
- Command Line Interface

12.5 INPUT/OUTPUT FILES

The 16-bit archiver/librarian generates archive files (`.a`). An archive file is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files.

By default, object files are processed in the COFF format. To specify COFF or ELF format explicitly, use the `-omf` option on the command line, as shown:

```
pic30-ar -omf=coff [options...]
pic30-ar -omf=elf [options...]
```

Alternatively, the environment variable `PIC30_OMF` may be used to specify object file format for the dsPIC30F language tools.

© 2008 Microchip Technology Inc.

12.6 SYNTAX

```
pic30-ar [-]P[MOD [RELPOS] [COUNT]] ARCHIVE [MEMBER...]
pic30-ar -M [ <mri-script ]
```

12.7 OPTIONS

When you use the 16-bit archiver/librarian with command line options, the archiver insists on at least two arguments to execute: one key letter specifying the operation (optionally accompanied by other key letters specifying modifiers), and the archive name.

```
pic30-ar [-]P[MOD [RELPOS][COUNT]] ARCHIVE [MEMBER...]
```

Note: Command line options are case sensitive.

Most operations can also accept further *MEMBER* arguments, specifying archive members. Without specifying members, the entire archive is used.

The 16-bit archiver/librarian allows you to mix the operation code P and modifier flags *MOD* in any order, within the first command line argument. If you wish, you may begin the first command line argument with a dash.

The P keyletter specifies what operation to execute; it may be any of the following, but you must specify only one of them.

TABLE 12-1: OPERATION TO EXECUTE

Option	Function
d	Delete modules from the archive. Specify the names of modules to be deleted as MEMBER...; the archive is untouched if you specify no files to delete. If you specify the v modifier, the 16-bit archiver/librarian lists each module as it is deleted.
m	Use this operation to move members in an archive. The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member. If no modifiers are used with m, any members you name in the MEMBER arguments are moved to the end of the archive; you can use the a, b or i modifiers to move them to a specified place instead.
p	Print the specified members of the archive, to the standard output file. If the v modifier is specified, show the member name before copying its contents to standard output. If you specify no MEMBER arguments, all the files in the archive are printed.
q	Append the files MEMBER... into ARCHIVE .
r	Insert the files MEMBER... into ARCHIVE (with replacement). If one of the files named in MEMBER... does not exist, the archiver displays an error message, and leaves undisturbed any existing members of the archive matching that name. By default, new members are added at the end of the file; but you may use one of the modifiers a, b or i to request placement relative to some existing member. The modifier v used with this operation elicits a line of output for each file inserted, along with one of the letters a or r to indicate whether the file was appended (no old member deleted) or replaced.
t	Display a table listing the contents of ARCHIVE, or those of the files listed in MEMBER..., that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group and size, you can request that by also specifying the v modifier. If you do not specify a MEMBER, all files in the archive are listed. For example, if there is more than one file with the same name (fie) in an archive (b.a), then pic30-ar t b.a fie lists only the first instance; to see them all, you must ask for a complete listing in pic30-ar t b.a.

TABLE 12-1: OPERATION TO EXECUTE (CONTINUED)

Option	Function
x	Extract members (named MEMBER) from the archive. You can use the v modifier with this operation, to request that the archiver list each name as it extracts it. If you do not specify a MEMBER, all files in the archive are extracted.

A number of modifiers (MOD) may immediately follow the P keyletter to specify variations on an operation's behavior.

TABLE 12-2: MODIFIERS

Option	Function
a	Add new files after an existing member of the archive. If you use the modifier a, the name of an existing archive member must be present as the RELPOS argument, before the ARCHIVE specification.
b	Add new files before an existing member of the archive. If you use the modifier b, the name of an existing archive member must be present as the RELPOS argument, before the ARCHIVE specification. (Same as i.)
c	Create the archive. The specified ARCHIVE is always created if it did not exist, when you requested an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.
f	Truncate names in the archive. The 16-bit archiver/librarian will normally permit file names of any length. This will cause it to create archives that are not compatible with the native archiver program on some systems. If this is a concern, the f modifier may be used to truncate file names when putting them in the archive.
i	Insert new files before an existing member of the archive. If you use the modifier i, the name of an existing archive member must be present as the RELPOS argument, before the ARCHIVE specification. (Same as b.)
l	This modifier is accepted but not used.
N	Uses the COUNT parameter. This is used if there are multiple entries in the archive with the same name. Extract or delete instance COUNT of the given name from the archive.
o	Preserve the original dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.
P	Use the full path name when matching names in the archive. The 16-bit archiver/librarian cannot create an archive with a full path name (such archives are not POSIX compliant), but other archive creators can. This option will cause the archiver to match file names using a complete path name, which can be convenient when extracting a single file from an archive created by another tool.
s	Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running pic30-ar s on an archive is equivalent to running ranlib on it.
S	Do not generate an archive symbol table. This can speed up building a large library in several steps. The resulting archive cannot be used with the linker. In order to build a symbol table, you must omit the S modifier on the last execution of the archiver, or you must run ranlib on the archive.
u	Normally, pic30-ar r... inserts all files listed into the archive. If you would like to insert only those of the files you list that are newer than existing members of the same names, use this modifier. The u modifier is allowed only for the operation r (replace). In particular, the combination qu is not allowed, since checking the timestamps would lose any speed advantage from the operation q.
v	This modifier requests the verbose version of an operation. Many operations display additional information, such as, file names processed when the modifier v is appended.
V	This modifier shows the version number of the 16-bit archiver/librarian.

© 2008 Microchip Technology Inc.

12.8 SCRIPTS

If you use the single command line option `-M` with the archiver, you can control its operation with a rudimentary command language.

```
pic30-ar -M [ <SCRIPT ]
```

> **Note:** Command line options are case sensitive.

This form of the 16-bit archiver/librarian operates interactively if standard input is coming directly from a terminal. During interactive use, the archiver prompts for input (the prompt is AR >), and continues executing even after errors. If you redirect standard input to a script file, no prompts are issued, and the 16-bit archiver/librarian abandons execution (with a nonzero exit code) on any error.

The archiver command language is **not** designed to be equivalent to the command line options; in fact, it provides somewhat less control over archives. The only purpose of the command language is to ease the transition to the 16-bit archiver/librarian for developers who already have scripts written for the MRI “librarian” program.

The syntax for the 16-bit archiver/librarian command language is straightforward:

- commands are recognized in upper or lower case; for example, `LIST` is the same as `list`. In the following descriptions, commands are shown in upper case for clarity.
- a single command may appear on each line; it is the first word on the line.
- empty lines are allowed, and have no effect.
- comments are allowed; text after either of the characters “*” or “;” is ignored.
- Whenever you use a list of names as part of the argument to an `pic30-ar` command, you can separate the individual names with either commas or blanks. Commas are shown in the explanations below, for clarity.
- “+” is used as a line continuation character; if “+” appears at the end of a line, the text on the following line is considered part of the current command.

Table 12-3 shows the commands you can use in archiver scripts, or when using the archiver interactively. Three of them have special significance.

TABLE 12-3: ARCHIVER SCRIPTS COMMANDS

Option	Function
`OPEN` or `CREATE`	Specify a “current archive”, which is a temporary file required for most of the other commands.
`SAVE`	Commits the changes so far specified by the script. Prior to `SAVE`, commands affect only the temporary copy of the current archive.
`ADDLIB ARCHIVE` `ADDLIB ARCHIVE (MODULE, MODULE,...MODULE)`	Add all the contents of `ARCHIVE` (or, if specified, each named `MODULE` from `ARCHIVE`) to the current archive. Requires prior use of `OPEN` or `CREATE`.
`ADDMOD MEMBER, MEMBER, ... MEMBER`	Add each named `MEMBER` as a module in the current archive. Requires prior use of `OPEN` or `CREATE`.
`CLEAR`	Discard the contents of the current archive, canceling the effect of any operations since the last `SAVE`. May be executed (with no effect) even if no current archive is specified.

TABLE 12-3: ARCHIVER SCRIPTS COMMANDS (CONTINUED)

Option	Function
CREATE ARCHIVE	Creates an archive, and makes it the current archive (required for many other commands). The new archive is created with a temporary name; it is not actually saved as ARCHIVE until you use SAVE. You can overwrite existing archives; similarly, the contents of any existing file named ARCHIVE will not be destroyed until SAVE.
DELETE MODULE, MODULE, ... MODULE	Delete each listed MODULE from the current archive; equivalent to pic30-ar -d ARCHIVE MODULE ... MODULE. Requires prior use of OPEN or CREATE.
DIRECTORY ARCHIVE (MODULE, ... MODULE) [OUTPUTFILE]	List each named MODULE present in ARCHIVE. The separate command VERBOSE specifies the form of the output: when verbose output is off, output is like that of pic30-ar -t ARCHIVE MODULE.... When verbose output is on, the listing is like pic30-ar -tv ARCHIVE MODULE.... Output normally goes to the standard output stream; however, if you specify OUTPUTFILE as a final argument, the 16-bit archiver/librarian directs the output to that file.
END	Exit from the archiver with a 0 exit code to indicate successful completion. This command does not save the output file; if you have changed the current archive since the last SAVE command, those changes are lost.
EXTRACT MODULE, MODULE, ... MODULE	Extract each named MODULE from the current archive, writing them into the current directory as separate files. Equivalent to pic30-ar -x ARCHIVE MODULE.... Requires prior use of OPEN or CREATE.
LIST	Display full contents of the current archive, in "verbose" style regardless of the state of VERBOSE. The effect is like pic30-ar tv ARCHIVE. (This single command is a 16-bit archiver/librarian enhancement, rather than present for MRI compatibility.) Requires prior use of OPEN or CREATE.
OPEN ARCHIVE	Opens an existing archive for use as the current archive (required for many other commands). Any changes as the result of subsequent commands will not actually affect ARCHIVE until you next use SAVE.
REPLACE MODULE, MODULE, ... MODULE	In the current archive, replace each existing MODULE (named in the REPLACE arguments) from files in the current working directory. To execute this command without errors, both the file, and the module in the current archive, must exist. Requires prior use of OPEN or CREATE.
VERBOSE	Toggle an internal flag governing the output from DIRECTORY. When the flag is on, DIRECTORY output matches output from pic30-ar -tv ....
SAVE	Commits your changes to the current archive and actually saves it as a file with the name specified in the last CREATE or OPEN command. Requires prior use of OPEN or CREATE.

© 2008 Microchip Technology Inc.

Part 4 – Utilities

Chapter 13. Utilities Overview 187
Chapter 14. pic30-bin2hex Utility 189
Chapter 15. pic30-nm Utility 191
Chapter 16. pic30-objdump Utility 195
Chapter 17. pic30-ranlib Utility 199
Chapter 18. pic30-strings Utility 201
Chapter 19. pic30-strip Utility 203
Chapter 20. pic30-lm Utility 205

NOTES:

© 2008 Microchip Technology Inc.

Chapter 13. Utilities Overview

13.1 INTRODUCTION

This chapter discusses general information about the utilities.

13.2 HIGHLIGHTS

Topics covered in this chapter are:

- What are Utilities

13.3 WHAT ARE UTILITIES

Utilities are tools available for use with the assembler and/or linker. The archiver/librarian utility was discussed in a previous chapter.

TABLE 13-1: AVAILABLE UTILITIES

Utility	Description
`pic30-ar`	Creates, modifies and extracts files from archives/libraries.
`pic30-bin2hex`	Converts a linked object file into an Intel® hex file.
`pic30-nm`	Lists symbols from an object file.
`pic30-objdump`	Displays information about object files.
`pic30-ranlib`	Generates an index from the contents of an archive and stores it in the archive.
`pic30-strings`	Prints the printable character sequences.
`pic30-strip`	Discards all symbols from an object file.
`pic30-lm`	Displays information about the compiler license.

NOTES:

© 2008 Microchip Technology Inc.

Chapter 14. pic30-bin2hex Utility

14.1 INTRODUCTION

The binary-to-hexadecimal (`pic30-bin2hex`) utility converts binary files (from the 16-bt linker) to Intel hex format files, suitable for loading into device programmers.

14.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

14.3 INPUT/OUTPUT FILES

- Input: COFF or ELF formatted binary object files
- Output: Intel hex files

By default, object files are processed in the COFF format. To specify COFF or ELF format explicitly, use the `-omf` option on the command line, as shown:

```
pic30-bin2hex -omf=coff file1.out
pic30-bin2hex -omf=elf file2.out
```

Alternatively, the environment variable `PIC30_OMF` may be used to specify object file format for the dsPIC30F language tools.

Because the Intel hex file format is byte-oriented, and the 16-bit program counter is not, program memory sections require special treatment. Each 24-bit program word is extended to 32 bits by inserting a so-called "phantom byte". Each program memory address is multiplied by 2 to yield a byte address.

For example, a section that is located at 0x100 in program memory will be represented in the hex file as 0x200. Consider the following assembly language source:

```
; file test.s
.section foo,code,address(0x100)
.pword 0x112233
```

The following commands will assemble the source file and create an Intel hex file:

```
pic30-as -o test.o test.s
pic30-bin2hex test.o
```

The file "test.hex" will be produced, with the following contents:

```
:020000040000fa
:040200003322110096
:00000001FF
```

Notice that the data record (line 2) has a load address of 0200, while the source code specified address 0x100. Note also that the data is represented in "little-endian" format, meaning the least significant byte appears first. The phantom byte appears last, just before the checksum.

14.4 SYNTAX

Command line syntax is:

```
pic30-bin2hex [-omf=format] [-v] file
```

Example 14.1: hello.cof

Convert the absolute COFF executable file `hello.cof` to `hello.hex`

```
pic30-bin2hex hello.cof
```

14.5 OPTIONS

The following options are supported.

TABLE 14-1: pic30-bin2hex OPTIONS

Option	Function
object_file -a	Sort the contents of the object file in ascending address order. For a summary of the object file contents, add the `-v` option (`-va`).
-omf=*format*	Specify object file format. The following formats are supported: COFF, ELF. Format names are case-insensitive. COFF in the default.
-v	Print a table of diagnostic information to standard output in the format shown in Example 14-2.

EXAMPLE 14-2: -v OPTION OUTPUT

```
writing hello.hex

section  PC address   byte address   length (w/pad)   actual length  (dec)
-------   ----------   ------------   --------------   --------------------
.reset             0              0              0x8             0x6  (6)
.text          0x100          0x200           0x6a28          0x4f9e  (20382)
.dinit        0x3614         0x6c28            0xda4           0xa3b  (2619)
.const        0x3ce6         0x79cc             0x40            0x30  (48)
.ivt             0x4            0x8             0xf8            0xba  (186)
.aivt           0x84          0x108             0xf8            0xba  (186)

Total program memory used (bytes):                            0x5b83  (23427)
```


© 2008 Microchip Technology Inc.

Chapter 15. pic30-nm Utility

15.1 INTRODUCTION

The `pic30-nm` utility produces a list of symbols from object files. Each item in the list consists of the symbol value, symbol type and symbol name.

15.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options
- Output Formats

15.3 INPUT/OUTPUT FILES

- Input: Object archive files
- Output: Object archive files. If no object files are listed as arguments, `pic30-nm` assumes the file `a.out`.

15.4 SYNTAX

Command line syntax is:

```
pic30-nm [ -A | -o | --print-file-name ]
      [ -a | --debug-syms ] [ -B ]
      [ --defined-only ] [ -u | --undefined-only ]
      [ -f format | --format=format ] [ -g | --extern-only ]
      [ --help ] [-l | --line-numbers ]
      [ -n | -v | --numeric-sort ]  [-omf=format]
      [ -p | --no-sort ]
      [ -P | --portability ] [ -r | --reverse-sort ]
      [ -s  --print-armap ] [ --size-sort ]
      [ -t radix | --radix=radix ] [ -V | --version ]
      [ OBJFILE... ]
```

15.5 OPTIONS

The long and short forms of options, shown in Table 15-1 as alternatives, are equivalent.

TABLE 15-1: pic30-nm OPTIONS

Option	Function
-A -o --print-file-name	Precede each symbol by the name of the input file (or archive member) in which it was found, rather than identifying the input file once only, before all of its symbols.
-a --debug-syms	Display all symbols, even debugger-only symbols; normally these are not listed.
-B	The same as --format=bsd.
--defined-only	Display only defined symbols for each object file.
-u --undefined-only	Display only undefined symbols (those external to each object file).
-f *format* --format=*format*	Use the output format *format*, which can be bsd, sysv or posix. The default is bsd. Only the first character of *format* is significant; it can be either upper or lower case.
-g --extern-only	Display only external symbols.
--help	Show a summary of the options to pic30-nm and exit.
-l --line-numbers	For each symbol, use debugging information to try to find a filename and line number. For a defined symbol, look for the line number of the address of the symbol. For an undefined symbol, look for the line number of a relocation entry that refers to the symbol. If line number information can be found, print it after the other symbol information.
-n -v --numeric-sort	Sort symbols numerically by their addresses, rather than alphabetically by their names.
-omf=*format*	Specify object file format. The following formats are supported: COFF, ELF. Format names are case-insensitive. COFF in the default.
-p --no-sort	Do not bother to sort the symbols in any order; print them in the order encountered.
-P --portability	Use the POSIX.2 standard output format instead of the default format. Equivalent to -f posix.
-r --reverse-sort	Reverse the order of the sort (whether numeric or alphabetic); let the last come first.
-s --print-armap	When listing symbols from archive members, include the index: a mapping (stored in the archive by pic30-ar or pic30-ranlib) of which modules contain definitions for which names.
--size-sort	Sort symbols by size. The size is computed as the difference between the value of the symbol and the value of the symbol with the next higher value. The size of the symbol is printed, rather than the value.
-t *radix* --radix=*radix*	Use *radix* as the radix for printing the symbol values. It must be d for decimal, o for octal or x for hexadecimal.
-V --version	Show the version number of pic30-nm and exit.

© 2008 Microchip Technology Inc.

15.6 OUTPUT FORMATS

The symbol value is in the radix selected by the options, or hexadecimal by default.

If the symbol type is lowercase, the symbol is local; if uppercase, the symbol is global (external). Table 15-2 shows the symbol types:

TABLE 15-2: SYMBOL TYPES

Symbol	Description
A	The symbol's value is absolute, and will not be changed by further linking.
B	The symbol is in the uninitialized data section (known as BSS).
C	The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.
D	The symbol is in the initialized data section.
N	The symbol is a debugging symbol.
R	The symbol is in a read only data section.
T	The symbol is in the text (code) section.
U	The symbol is undefined.
V	The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
W	The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
?	The symbol type is unknown, or object file format specific.

NOTES:

© 2008 Microchip Technology Inc.

Chapter 16. pic30-objdump Utility

16.1 INTRODUCTION

The `pic30-objdump` utility displays information about one or more object files. The options control what particular information to display.

16.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

16.3 INPUT/OUTPUT FILES

- Input: Object archive files
- Output: Object archive files. If no object files are listed as arguments, `pic30-nm` assumes the file `a.out`.

16.4 SYNTAX

Command line syntax is:

```
pic30-objdump [ -a | --archive-headers ]
      [ -d | --disassemble ]
      [ -D | --disassemble-all ]
      [ -EB | -EL | --endian={big | little } ]
      [ -f | --file-headers ]
      [ --file-start-context ]
      [ -g | --debugging ]
      [ -h | --section-headers | --headers ]
      [ -H | --help ]
      [ -j name | --section=name ]
      [ -l | --line-numbers ]
      [ -M options | --disassembler-options=options]
      [-omf=format]
      [ --prefix-addresses]
      [ -r | --reloc ]
      [ -s | --full-contents ]
      [ -S | --source ]
      [ --[no-]show-raw-insn ]
      [ --start-address=address ]
      [ --stop-address=address ]
      [ -t | --syms ]
      [ -V | --version ]
      [ -w | --wide ]
      [ -x | --all-headers ]
      [ -z | --disassemble-zeroes ]
      OBJFILE...
```

OBJFILE... are the object files to be examined. When you specify archives, pic30-objdump shows information on each of the member object files.

16.5 OPTIONS

The long and short forms of options, shown in Table 16-1, as alternatives, are equivalent. At least one of the following options -a, -d, -D, -f, -g, -G, -h, -H, -p, -r, -R, -S, -t, -T, -V or -x must be given.

TABLE 16-1: pic30-objdump OPTIONS

Option	Function
-a --archive-header	If any of the OBJFILE files are archives, display the archive header information (in a format similar to ls -l). Besides the information you could list with pic30-ar tv, pic30-objdump -a shows the object file format of each archive member.
-d --disassemble	Display the assembler mnemonics for the machine instructions from OBJFILE. This option only disassembles those sections that are expected to contain instructions.
-D --disassemble-all	Like -d, but disassemble the contents of all sections, not just those expected to contain instructions.
-EB -EL --endian={big|little}	Specify the endianness of the object files. This only affects disassembly. This can be useful when disassembling a file format that does not describe endianness information, such as S-records.
-f --file-header	Display summary information from the overall header of each of the OBJFILE files.
--file-start-context	Specify that when displaying inter-listed source code/disassembly (assumes '-S') from a file that has not yet been displayed, extend the context to the start of the file.
-g --debugging	Display debugging information. This attempts to parse debugging information stored in the file and print it out using a C like syntax. Only certain types of debugging information have been implemented.
-h --section-header --header	Display summary information from the section headers of the object file. File segments may be relocated to nonstandard addresses, for example by using the -Ttext, -Tdata or -Tbss options to ld. However, some object file formats, such as a.out, do not store the starting address of the file segments. In those situations, although ld relocates the sections correctly, using pic30-objdump -h to list the file section headers cannot show the correct addresses. Instead, it shows the usual addresses, which are implicit for the target.
-H --help	Print a summary of the options to pic30-objdump and exit.
-j *name* --section=*name*	Display information only for section *name*.
-l --line-numbers	Label the display (using debugging information) with the filename and source line numbers corresponding to the object code or relocs shown. Only useful with -d, -D or -r.
-M *options* --disassembler-options=*options*	Pass target specific information to the disassembler. The dsPIC30F device supports the following target specific options: symbolic - Will perform symbolic disassembly.

© 2008 Microchip Technology Inc.

TABLE 16-1: pic30-objdump OPTIONS (CONTINUED)

Option	Function
-omf=*format*	Specify object file format. The following formats are supported: COFF, ELF. Format names are case-insensitive. COFF in the default.
--prefix-addresses	When disassembling, print the complete address on each line. This is the older disassembly format.
-r --reloc	Print the relocation entries of the file. If used with -d or -D, the relocations are printed interspersed with the disassembly.
-s --full-contents	Display the full contents of any sections requested.
-S --source	Display source code intermixed with disassembly, if possible. Implies -d.
--show-raw-insn	When disassembling instructions, print the instruction in hex, as well as in symbolic form. This is the default except when --prefix-addresses is used.
--no-show-raw-insn	When disassembling instructions, do not print the instruction bytes. This is the default when --prefix-addresses is used.
--start-address=*address*	Start displaying data at the specified address. This affects the output of the -d, -r and -s options.
--stop-address=*address*	Stop displaying data at the specified address. This affects the output of the -d, -r and -s options.
-t --syms	Print the symbol table entries of the file. This is similar to the information provided by the pic30-nm program.
-V --version	Print the version number of pic30-objdump and exit.
-w --wide	Format some lines for output devices that have more than 80 columns.
-x --all-header	Display all available header information, including the symbol table and relocation entries. Using -x is equivalent to specifying all of -a -f -h -r -t.
-z --disassemble-zeroes	Normally the disassembly output will skip blocks of zeroes. This option directs the disassembler to disassemble those blocks, just like any other data.

NOTES:

© 2008 Microchip Technology Inc.

Chapter 17. pic30-ranlib Utility

17.1 INTRODUCTION

The `pic30-ranlib` utility generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. You may use `pic30-nm -s` or `pic30-nm --print-armap` to list this index. An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

Running `pic30-ranlib` is completely equivalent to executing `pic30-ar -s` (i.e., the 16-bit archiver/librarian with the `-s` option).

17.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

17.3 INPUT/OUTPUT FILES

- Input: Archive files
- Output: Archive files

17.4 SYNTAX

Command line syntax is:

`pic30-ranlib [-omf=`*`format`*`] [-v | -V | --version] ARCHIVE`

17.5 OPTIONS

The long and short forms of options, shown here as alternatives, are equivalent.

TABLE 17-1: pic30-ranlib OPTIONS

Option	Function
`-omf=`*`format`*	Specify object file format. The following formats are supported: COFF, ELF. Format names are case-insensitive. COFF in the default.

`-v`
`-V`
`--version`
	Show the version number of `pic30-ranlib`

NOTES:

© 2008 Microchip Technology Inc.

Chapter 18. pic30-strings Utility

18.1 INTRODUCTION

For each file given, the `pic30-strings` utility prints the printable character sequences that are at least 4 characters long (or the number given in the options) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

`pic30-strings` is mainly useful for determining the contents of non-text files.

18.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

18.3 INPUT/OUTPUT FILES

- Input: Any files
- Output: Standard output

18.4 SYNTAX

Command line syntax is:

```
pic30-strings [-a | --all | -] [-f | --print-file-name]
      [--help] [-min-len | -n min-len | --bytes=min-len]
      [-omf=format] [-t radix | --radix=radix]
      [-v | --version] FILE...
```

18.5 OPTIONS

The long and short forms of options, shown in Table 18-1 as alternatives, are equivalent.

TABLE 18-1: pic30-strings OPTIONS

Option	Function

`-a`
`--all`
`-`
	Do not scan only the initialized and loaded sections of object files; scan the whole files.

`-f` `--print-file-name`	Print the name of the file before each string.
`--help`	Print a summary of the program usage on the standard output and exit.
-min-len `-n` *min-len* `--bytes=`*min-len*	Print sequences of characters that are at least *`-min-len`* characters long, instead of the default 4.
`-omf=`*`format`*	Specify object file format. The following formats are supported: COFF, ELF. Format names are case-insensitive. COFF in the default.
`-t` *radix* `--radix=`*radix*	Print the offset within the file before each string. The single character argument specifies the radix of the offset `- o` for octal, `x` for hexadecimal or `d` for decimal.
`-v` `--version`	Print the program version number on the standard output and exit.

© 2008 Microchip Technology Inc.

Chapter 19. pic30-strip Utility

19.1 INTRODUCTION

The `pic30-strip` utility discards all symbols from the object and archive files specified. At least one file must be given. `pic30-strip` modifies the files named in its argument, rather than writing modified copies under different names.

19.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

19.3 INPUT/OUTPUT FILES

- Input: Object or archive files
- Output: Object or archive files. If no object or archive files are listed as arguments, `pic30-size` assumes the file `a.out`.

19.4 SYNTAX

Command line syntax is:

```
pic30-strip [ -g | -S | --strip-debug ] [ --help ]
      [ -K symbolname | --keep-symbol=symbolname ]
      [ -N symbolname | --strip-symbol=symbolname ]
      [ -o file ] [-omf=format]
      [ -p | --preserve-dates ]
      [ -R sectionname | --remove-section=sectionname ]
      [ -s | --strip-all ] [--strip-unneeded]
      [ -v | --verbose ]  [ -V | --version ]
      [ -x | --discard-all ] [ -X | --discard-locals ]
      OBJFILE...
```

19.5 OPTIONS

The long and short forms of options, shown in Table 19-1 as alternatives, are equivalent.

TABLE 19-1: pic30-strip OPTIONS

Option	Function

`-g`
`-S`
`--strip-debug`
	Remove debugging symbols only.

`--help`	Show a summary of the options to `pic30-strip` and exit.
`-K` *symbolname* `--keep-symbol=`*symbolname*	Keep only symbol *`symbolname`* from the source file. This option may be given more than once.
`-N` *symbolname* `--strip-symbol=`*symbolname*	Remove symbol *`symbolname`* from the source file. This option may be given more than once, and may be combined with strip options other than `-K`.
`-o` *file*	Put the stripped output in *`file`*, rather than replacing the existing file. When this argument is used, only one `OBJFILE` argument may be specified.
`-omf=`*`format`*	Specify object file format. The following formats are supported: COFF, ELF. Format names are case-insensitive. COFF in the default.
`-p` `--preserve-dates`	Preserve the access and modification dates of the file.
`-R` *sectionname* `--remove-section=`*sectionname*	Remove any section named *`sectionname`* from the output file. This option may be given more than once. Note that using this option inappropriately may make the output file unusable.
`-s` `--strip-all`	Remove all symbols.
`--strip-unneeded`	Remove all symbols that are not needed for relocation processing.
`-v` `--verbose`	Verbose output: list all object files modified. In the case of archives, `pic30-strip -v` lists all members of the archive.
`-V` `--version`	Show the version number for `pic30-strip`.
`-x` `--discard-all`	Remove non-global symbols.
`-X` `--discard-locals`	Remove compiler-generated local symbols. (These usually start with `L` or "`.`".)

© 2008 Microchip Technology Inc.

Chapter 20. pic30-lm Utility

20.1 INTRODUCTION

The `pic30-lm` utility displays information about the compiler license. For full-product versions, `pic30-lm` displays the license number. For demo-product versions, `pic30-lm` displays the number of days remaining on the license. The `pic30-lm` utility may also be used to upgrade a demo product to a full product.

20.2 HIGHLIGHTS

Topics covered in this chapter are:

- Syntax
- Options

20.3 SYNTAX

The pic30-lm command-line syntax is:

```
pic30-lm [-?] [-u license]
```

If `pic30-lm` is invoked without options, it does one of the following things:

1. If the installed 16-bit compiler product is a full product, then the license number of the product is displayed. You should have this license number available when you contact Microchip for technical support.
2. If the installed 16-bit compiler product is a demo product, then the number of days remaining on the license is displayed.

No more than one option may be specified at any one time. If more than one option is specified, or if the syntax of the option is incorrect, `pic30-lm` will not perform any action other than reporting the fact that it has been misused.

20.4 OPTIONS

The `pic30-lm` options are shown below.

TABLE 20-1: pic30-lm OPTIONS

Option	Function
-?	Displays usage information for `pic30-lm`. A brief description of the -? and -u options is displayed
-u license	Upgrade a demo version to a full version. Spaces between -u and license are optional. The license parameter should be the license key that is printed on the bottom of the compiler box. Type the license key exactly as it appears on the box, including the correct case for any letters that appear in the license key.

NOTES:

© 2008 Microchip Technology Inc.

Part 5 – Command-Line Simulator

Chapter 21. SIM30 Command-Line Simulator 209

NOTES:

© 2008 Microchip Technology Inc.

Chapter 21. SIM30 Command-Line Simulator

21.1 INTRODUCTION

A basic command-line simulator (`sim30.exe`) may be used to test and debug dsPIC30F/33F DSC and PIC24F/H MCU programs.

21.2 HIGHLIGHTS

Topics covered in this chapter are:

- Syntax
- Options

21.3 SYNTAX

The simulator is invoked from the Windows command prompt as follows:

```
sim30 [command-file-name]
```

where the optional parameter `command-file-name` names a text file containing simulator commands, one per line. If the command file is specified, the simulator reads commands from the file before reading commands from the keyboard.

EXAMPLE 21-1: HELLO.COF

To run the file `hello.cof` using the simulator, first load the COFF file. Next, reset the processor. Then, enable the C library I/O. Finally, run the program and quit the simulator. Check `UartOut.txt` for output. (If using the `hello.c` file included in the examples directory of the installation disk to create the `hello.cof` file, the output file `UartOut.txt` would contain "Hello, world!")

```
sim30
dsPIC30> lc hello.cof ; load the COFF file
dsPIC30> rp           ; reset the processor
dsPIC30> io nul       ; enable C library I/O (stdin is nul)
dsPIC30> e            ; execute (run) the program
dsPIC30> q            ; quit the simulation session
```

21.4 OPTIONS

Table 21-1 summarizes the commands supported by the simulator. Each command should be terminated by pressing the <enter> key.

Simple editing of the command line is available using the <backspace> key.

Note: The commands are NOT case sensitive.

TABLE 21-1: SUPPORTED SIMULATOR COMMANDS

Option	Description
AF	`AF [<frequency>]` Alter or display the oscillator frequency. If the `frequency` parameter is omitted, the current oscillator frequency is displayed.
BC	`BC <location> ...[locations]` Breakpoint Clear.
BS	`BS <location> ...[locations]` Breakpoint Set.
DA	`DA` Display the accumulators.
DB	`DB` Display the breakpoints.
DC	`DC` Display PC disassembled.
DF	`DF [start] [end]` Display File Registers between specified addresses.
DH	`DH` Display Help on all.
DM	`DM [start] [end]` Display Program Memory between specified addresses.
DP	`DP` Display Profile. If the simulator is running in verbose mode (see the VO command), instruction execution statistics are displayed.
DS	`DS` Display Status register fields.
DW	`DW` Display the W Registers.
E	`E` Execute.
FC	`FC <location> [locations]` File register Clear.
FS	`FS <location> <location/ value> [value]` File register Set.
H	`H` Halt.
HE	`HE [ON | OFF]` Halt on Error. Enables or disables halt on error. Specifying `ON` enables halt on error; specifying `OFF` disables halt on error. Omitting the parameter causes the current halt on error status to be displayed.
HW	`HW [ON | OFF]` Halt on Warning. Enables or disables halt on warning. Specifying `ON` enables halt on warning; specifying `OFF` disables halt on warning. Omitting the parameter causes the current halt on warning status to be displayed.

© 2008 Microchip Technology Inc.

TABLE 21-1: SUPPORTED SIMULATOR COMMANDS (CONTINUED)

Option	Description

IO
 	
IF
	`IO [stdin [stdout]]`
	Enable simulated file I/O.
	`IF`
	Disable simulated file I/O.
	The simulator supports the C compiler's standard library I/O functions. This allows standard C programs to be written and tested on the simulator.
	Support for the standard I/O functions of the C compiler is enabled using the IO simulator command. Once enabled, it can be disabled using the IF command. If enabled, `stdin`, `stdout` and `stderr` use the UART1 peripheral. By default, a stimulus file named `UartIn.txt` (for `stdin`) and a response file named `UartOut.txt` (for both `stdout` and `stderr`) are attached to the UART. Both files are opened in eight-bit binary format. The simulator looks for `UartIn.txt` in the current working directory. If no such file exists, no attachment is made to the UART1 receive register, and an error message is displayed. Similarly, the simulator creates (or over-writes) the file `UartOut.txt` in the current working directory. The default filenames `UartIn.txt` and `UartOut.txt` may be overridden by explicitly naming the files with the IO command's `stdin` and `stdout` parameters, respectively. The special name `nul` may be used to indicate that nothing is to be attached to the corresponding stream.
	The UART1 peripheral is used in polled mode; interrupts are not used. All other file I/O is directed to the host file system. When C standard I/O is enabled, any other stimulus or response files connected to the UART1 peripheral will be detached, and the above file names will be attached. When C standard I/O is disabled, the on-demand files are detached and the UART1 is left with no attached stimulus or response files.

LC	`LC <filename>` Load Program Memory from a COFF file.
LD	`LD <devicename>` Load parameters for a device, including memory configuration and peripheral set. See the on-line file "Readme for MPLAB SIM.txt" for a list of supported devices.
LF	`LF <filename> [displacement]` Load File Registers from an Intel hex file starting at offset `displacement`.
LP	`LP <filename> [displacement]` Load Program Memory from an Intel hex file starting at the offset `displacement`.
LS	`LS [<filename>]` Load a Stimulus Control Language (SCL) file. If the `filename` parameter is specified, the named file is analyzed by the SCL compiler, and a stimulus schedule is created and attached to the simulation session. If the `filename` parameter is omitted, any previously loaded SCL file is detached from the simulation session.
MC	`MC <location> [locations]` Program Memory Clear.
MS	`MS <location> <location/ value> [value]` Program Memory Set.
PS	`PS <value>` PC Set.
Q	`Q` Quit.
RC	`RC` Reset the simulation clock to cycle zero.
RP	`RP` Reset processor.

TABLE 21-1: SUPPORTED SIMULATOR COMMANDS (CONTINUED)

Option	Description
S	S Step.
VF	VF Verbose off.
VO	VO Verbose on.

© 2008 Microchip Technology Inc.

Part 6 – Appendices

Appendix A. Assembler Errors/Warnings/Messages 215
Appendix B. Linker Errors/Warnings 229
Appendix C. Deprecated Features 237
Appendix D. MPASM™ Assembler Compatibility 239
Appendix E. MPLINK™ Linker Compatibility 249
Appendix F. MPLIB™ Librarian Compatibility 251
Appendix G. Useful Tables 253
Appendix H. GNU Free Documentation License 255

NOTES:

© 2008 Microchip Technology Inc.

Appendix A. Assembler Errors/Warnings/Messages

A.1 INTRODUCTION

MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB ASM30) generates errors, warnings and messages. A descriptive list of these outputs is shown here.

A.2 HIGHLIGHTS

Topics covered in this appendix are:

- Fatal Errors
- Errors
- Warnings
- Messages

A.3 FATAL ERRORS

The following errors indicate that an internal error has occurred in the assembler. Please contact Microchip Technology for support if any of the following errors are generated:

- A dummy instruction cannot be used!
- bad floating-point constant: exponent overflow, probably assembling junk
- bad floating-point constant: unknown error code=error_code
- C_EFCN symbol out of scope
- Can't continue
- Can't extend frag num. chars
- Can't open a bfd on stdout name
- Case value val unexpected at line _line_ of file "_file_"
- emulations not handled in this configuration
- error constructing pop_table_name pseudo-op table: err_txt
- expr.c(operand): bad atof_generic return val val
- failed sanity check.
- filename:line_num: bad return from bfd_install_relocation: val
- filename:line_num: bad return from bfd_install_relocation
- Inserting "name" into symbol table failed: error_string
- Internal error: pic30_get_g_or_h_mode_value called with an invalid operand type
- Internal error: pic30_get_p_or_q_mode_value called with an invalid operand type
- Internal error: pic30_insert_dsp_writeback called with an invalid operand type
- Internal error: pic30_insert_dsp_x_prefetch_operation called with an invalid offset
- Internal error: pic30_insert_dsp_x_prefetch_operation called with an invalid operand type
- Internal error: pic30_insert_dsp_y_prefetch_operation called with an invalid offset
- Internal error: pic30_insert_dsp_y_prefetch_operation called with an invalid operand type

- invalid segment “name”; segment “name” assumed
- label “temp$” redefined
- macros nested too deeply
- missing emulation mode name
- multiple emulation names specified
- Relocation type not supported by object file format
- reloc type not supported by object file format
- rva not supported
- rva without symbol
- unrecognized emulation name ‘em’
- Unsupported BFD relocation size in bytes

A.4 ERRORS

Symbol A B C D E F I L M N O P R S T U W

Symbol

.abort detected. Abandoning ship.

User error invoked with the `.abort` directive.

.else without matching .if - ignored.

A `.else` directive was seen without a preceding .if directive.

“.elseif” after “.else” - ignored

A `.elseif` directive specified after a `.else` directive. Modify your code so that the `.elseif` directive comes before the `.else` directive.

“.elseif” without matching “.if” - ignored.

A `.elseif` directive was seen without a preceding `.if` directive.

“.endif” without “.if”

A `.endif` directive was seen without a preceding `.if` directive.

.err encountered.

User error invoked with the `.err` directive.

 # sign not valid in data allocation directive.

The # sign cannot be used within a data allocation directive (`.byte`, `.word`, `.pword`, `.long`, etc.)

 # warnings, treating warnings as errors.

The `--fatal-warnings` command line option was specified on the command line and warnings existed.

A

absolute address can not be specified for section '.const'

Section .const is a C compiler resource. Although it is permissible for an application to allocate constants in section .const explicitly, it is not permissible to assign an absolute address for this section.

Absolute address must be greater than or equal to 0.

A negative absolute address was specified as the target for the `DO` or `BRA` instruction. The assembler does not know anything about negative addresses.

© 2008 Microchip Technology Inc.

Alignment in CODE section must be at least 2 units.

The alignment value for the `.align` directive must be at least 2 units. Either no alignment was specified or an alignment less than 2 was specified. Modify the `.align` directive to have an alignment of at least 2.

Attributes for section 'name' conflict with implied attributes

Certain section names have implied attributes. In this case, the attributes specified in a `.section` directive conflict with its implied attributes. See **Section 6.3 "Directives that Define Sections"** for more information.

B

backw. ref to unknown label "#:", 0 assumed.

A backwards reference was made to a local label that was not seen. See **Section 5.4 "Reserved Names"** for more information on local labels.

bad defsym; format is --defsym name=value.

The format for the command line option `--defsym` is incorrect. Most likely, you are missing the = between the name and the value.

Bad expression.

The assembler did not recognize the expression. See **Chapter 3. "Assembler Syntax"**, **Chapter 4. "Assembler Expression Syntax and Operation"** and **Chapter 5. "Assembler Symbols"**, for more details on assembler syntax.

bignum invalid; zero assumed.

The big number specified in the expression is not valid.

Byte operations expect an offset between -512 and 511.

The offset specified in [Wn+offset] or [Wn-offset] exceeded the maximum or minimum value allowed for byte instructions.

C

Cannot call a symbol (name) that is not located in an executable section.

Attempted to `CALL` a symbol that is not located in a `CODE` section.

Cannot create floating-point number.

Could not create a floating-point number because of exponent overflow or because of a floating-point exception that prohibits the assembler from encoding the floating-point number.

Cannot redefine executable symbol 's'

A statement label or an executable section cannot be redefined with a .set or .equ directive.

Cannot reference executable symbol (name) in a data context.

An attempt was made to use a symbol in an executable section as a data address. To reference an executable symbol in a data context, the `psvoffset()` or `tbloffset()` operator is required.

Cannot use a constant as the argument of dmaoffset.

An attempt was made to use a constant as the argument to a dmaoffset.

Cannot use operator on a symbol (name) that is not located in a code, psv or eedata section.

You cannot use one of the special operators (`tbloffset`, `tblpage`, `psvoffset`, `psvpage`, `handle` or `paddr`) on a symbol that is not located in a `code`, `psv` or `eedata` section.

Cannot use operator with this directive.

An attempt was made to use a special operator (`tbloffset`, `tblpage`, `psvoffset`, `psvpage`, `handle` or `paddr`) with a data allocation directive that does not allocate enough bytes to store the requested data.

Cannot write to output file.

For some reason, the output file could not be written to. Check to ensure that you have write permission to the file and that there is enough disk space.

Can't open file_name for reading.

The specified input source file could not be opened. Ensure that the file exists and that you have permission to access the file.

D

directive directive not supported in pic30 target.

The pic30 target does not support this directive. This directive is available in other versions of the assembler, but the pic30 target does not support it for one reason or another. Please check **Chapter 6. "Assembler Directives"** for a complete list of supported directives.

duplicate "else" - ignored.

Two `.else` directives were specified for the same `.if` directive.

E

end of file inside conditional.

The file ends without terminating the current conditional. Add a `.endif` to your code.

end of macro inside conditional.

A conditional is unterminated inside a macro. The `.endif` directive to end the current conditional was not specified before seeing the `.endm` directive.

Expected comma after symbol-name: rest of line ignored.

Missing comma from the `.comm` directive after the symbol name.

Expected constant expression for fill argument.

The fill argument for the `.fill`, `.pfill`, `.skip`, `.pskip`, `.space` or `.pspace` directive must be a constant value. Attempted to use a symbol. Replace symbol with a constant value.

Expected constant expression for new-lc argument.

The new location counter argument for the `.org` directive must be a constant value. Attempted to use a symbol. Replace symbol with a constant value.

Expected constant expression for repeat argument.

The repeat argument for the `.fill`, `.pfill`, `.skip`, `.pskip`, `.space` or `.pspace` directive must be a constant value. Attempted to use a symbol. Replace symbol with a constant value.

Expected constant expression for size argument.

The size argument for the `.fill` or `.pfill` directive must be a constant value. Attempted to use a symbol. Replace symbol with a constant value.

Expression too complex.

An expression is too complex for the assembler to process.

© 2008 Microchip Technology Inc.

F

floating point number invalid; zero assumed.

The floating-point number specified in the expression is not valid.

I

Ignoring attempt to re-define symbol 'symbol'.

The symbol that you are attempting to define with `.comm` or `.lcomm` has already been defined and is not a common symbol.

Invalid expression (expr) contained inside of the brackets.

Assembler did not recognize the expression between the brackets.

invalid identifier for ".ifdef"

The identifier specified after the `.ifdef` must be a symbol. See **Section 5.3 "What are Symbols"** and **Section 6.10 "Directives that Control Conditional Assembly"** for more details.

Invalid mnemonic: 'token'

The token being parsed is not a valid mnemonic for the instruction set.

invalid listing option 'optarg'

The sub-option specified is not valid. Acceptable sub-options are `c`, `d`, `h`, `l`, `m`, `n`, `v` and `=`.

Invalid operands specified ('insn'). Check operand #n.

The operands specified were invalid. The assembler was able to match n-1 operands successfully. Although there is no assurance that operand #n is the culprit, it is a general idea of where you should begin looking.

Invalid operand syntax ('insn').

This message usually comes hand-in-hand with one of the previous operand syntax errors.

Invalid post increment value. Must be +/- 2, 4 or 6.

Assembler saw [Wn]+=value, where value is expected to be a +/- 2, 4 or 6. value was not correct. Specify a value of +/- 2, 4 or 6.

Invalid post decrement value. Must be +/- 2, 4 or 6.

Assembler saw [Wn]-=value, where value is expected to be a +/- 2, 4 or 6. value was not correct. Specify a value of +/- 2, 4 or 6.

Invalid register in operand expression.

Assembler was attempting to find either pre- or post-increment or decrement. The operand did not contain a register. Specify one of the registers w0-w16 or W0-W16.

Invalid register in expression reg.

Assembler saw [junk] or [junk]+=n or [junk]-=n. Was expecting a register between the brackets. Specify one of the registers w0-w16 or W0-W16 between the brackets.

Invalid use of ++ in operand expression.

Assembler was attempting to find either pre- or post-increment. The operand specified was neither pre-increment [++Wn] nor post-increment [Wn++]. Make sure that you are not using the old syntax of [Wn]++.

Invalid use of -- in operand expression.

Assembler was attempting to find either pre- or post-decrement. The operand specified was neither pre-decrement [--Wn] nor post-decrement [Wn--]. Make sure that you are not using the old syntax of [Wn]--.

Invalid value (#) for relocation name.

The final value of the relocation is not a valid value for the operand associated with the given relocation.

'name' is not a valid attribute name.

While processing a `.section` directive, the assembler found an identifier that is not a valid section attribute.

L

Length of .comm "sym" is already #. Not changed to #.

An attempt was made to redefine the length of a common symbol.

M

misplaced)

Missing parenthesis when expanding a macro. The syntax \(...) will literally substitute the text between the parenthesis into the macro. The trailing parenthesis was missing from this syntax.

Missing model parameter.

Missing symbol in the `.irp` or `.irpc` directive.

Missing right bracket.

The assembler did not see the terminating bracket ']'.

Missing size expression.

The `.lcomm` directive is missing the length expression.

Missing ')' after formals.

Missing trailing parenthesis when listing the macro formals inside of parenthesis.

Missing ')' assumed.

Expected a terminating parenthesis ')' while parsing the expression. Did not see one where expected so assumes where you wanted the trailing parenthesis.

Missing ']' assumed.

Expected a terminating brace ']' while parsing the expression. Did not see one where expected so assumes where you wanted the trailing brace.

Mnemonic not found.

The assembler was expecting to parse an instruction and could not find a mnemonic.

N

Negative of non-absolute symbol name.

Attempted to take the negative of a symbol name that is non-absolute. For example, `.word -`*`sym`*, where *`sym`* is external.

New line in title.

The `.title` heading is missing a terminating quote.

non-constant expression in ".elseif" statement.

The argument of the `.elseif` directive must be a constant value able to be resolved on the first pass of the directive. Ensure that any `.equ` of a symbol used in this argument is located before the directive. See **Section 6.10 "Directives that Control Conditional Assembly"** for more details.

© 2008 Microchip Technology Inc.

non-constant expression in ".if" statement.

The argument of the `.if` directive must be a constant value able to be resolved on the first pass of the directive. Ensure that any `.equ` of a symbol used in this argument is located before the directive. See **Section 6.10 "Directives that Control Conditional Assembly"** for more details.

Number of operands exceeds maximum number of 8.

Too many operands were specified in the instruction. The largest number of operands accepted by any of the dsPIC30F instructions is 8.

O

Only support plus register displacement (i.e., [Wb+Wn]).

Assembler found [Wb-Wn]. The syntax only supports a plus register displacement.

Operands share encoding bits. The operands must encode identically.

Two operands are register with displacement addressing mode [Wb+Wn]. The two operands share encoding bits so the Wn portion must match or be able to be switched to match the Wb of the other operand.

operation combines symbols in different segments.

The left-hand side of the expression and the right-hand side of the expression are located in two different sections. The assembler does not know how to handle this expression.

operator modifier must be preceded by a #.

The modifier (`tbloffset`, `tblpage`, `psvoffset`, `psvpage`, `handle`) was specified inside of an instruction, but was not preceded by a #. Include the # to represent that this is a literal.

P

paddr modifier not allowed in instruction.

The `paddr` operator was specified in an instruction. This operator can only be specified in a `.pword` or `.long` directive as those are the only two locations that are wide enough to store all 24 bits of the program address.

PC relative expression is not a valid GOTO target

The assembler does not support expressions which modify the program counter of a GOTO destination such as ". + 4" or "sym + 100".

R

Register expected as first operand of expression expr.

Assembler found [junk+anything] or [junk-anything]. The only valid expression contained in brackets with a + or a - requires that the first operand be a register.

Register or constant literal expected as second operand of expression expr.

Assembler found [Wn+junk] or [Wn-junk]. The only valid operand for this format is register with plus or minus literal offset or register with displacement.

Requested alignment 'n' is greater than alignment of absolute section 'name'

When the `address()` attribute is used to specify an absolute address for a section, it constrains the ability of the assembler to align objects within the section. The alignment specified in a `.align` or `.palign` directive must not be greater than the alignment implied by the section address.

S

section alignment must be a power of two

The argument to an `align()` or `reverse()` section attribute was invalid.

section address 0xnnnn exceeds near data range
section address must be even
section address must be in range [0..0x7ffffe]

The argument to an `address()` section attribute was invalid.

Symbol 'name' can not be both weak and common.

Both the `.weak` directive and `.comm` directive were used on the same symbol within the same source file.

syntax error in .startof. or .sizeof.

The assembler found either `.startof.` or `.sizeof.`, but did not find the beginning parenthesis '(' or ending parenthesis ')'. See **Section 4.5.5 "Obtaining the Size of a Specific Section"** and **Section 4.5.6 "Obtaining the Starting Address of a Specific Section"** for details on the `.startof.` and `.sizeof.` operators.

T

This expression is not a valid GOTO target

The assembler does not support expressions that include unresolved symbols as a GOTO destination.

Too few operands ('insn').

Too few operands were specified for this instruction.

Too many operands ('insn').

Too many operands were specified for this instruction.

U

unexpected end of file in irp or irpc

The end of the file was seen before the terminating `.endr` directive.

unexpected end of file in macro definition.

The end of the file was seen before the terminating `.endm` directive.

Unknown pseudo-op: 'directive'.

The assembler does not recognize the specified directive. Check to see that you have spelled the directive correctly. Note: the assembler expects that anything that is preceded by a dot (.) is a directive.

W

WAR hazard detected.

The assembler found a Write After Read hazard in the instruction. A WAR hazard occurs when a common W register is used for both the source and destination given that the source register uses pre/post-increment/decrement.

Word operations expect even offset.

An attempt was made to specify [Wn+offset] or [Wn-offset] where offset is even with a word instruction.

Word operations expect an even offset between -1024 and 1022.

The offset specified in [Wn+offset] or [Wn-offset] was even, but exceeded the maximum or minimum value allowed for word instructions.

© 2008 Microchip Technology Inc.

A.5 WARNINGS

The assembler generates warnings when an assumption is made so that the assembler could continue assembling a flawed program. Warnings should not be ignored. Each warning should be specifically looked at and corrected to ensure that the assembler understands what was intended. Warning messages can sometimes point out bugs in your program.

Symbol

.def pseudo-op used inside of .def/.endef: ignored.

The specified directive is not allowed within a *.def/.endef* pair. *.def/.endef* directives are used for specifying debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, note that:

1. you want to use the *.line* directive to specify the line number information for the symbol, and
2. you cannot nest *.def/.endef* directives.

.dim pseudo-op used outside of .def/.endef: ignored.

The specified directive is only allowed within a *.def/.endef* pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a *.def* directive before specifying this directive.

.endef pseudo-op used outside of .def/.endef: ignored.

The specified directive is only allowed within a *.def/.endef* pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a *.def* directive before specifying this directive.

.fill size clamped to 8.

The size argument (second argument) of the *.fill* directive specified was greater then eight. The maximum size allowed is eight.

.fillupper expects a constant positive byte value. 0xXX assumed.

The *.fillupper* directive was specified with an argument that is not a constant positive byte value. The last *.fillupper* value that was specified will be used.

.fillupper not specified in a code section. .fillupper ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.fillvalue expects a constant positive byte value. 0xXX assumed.

The *.fillvalue* directive was specified with an argument that is not a constant positive byte value. The last *.fillvalue* value that was specified will be used.

.fillvalue not specified in a code section. .fillvalue ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.ln pseudo-op inside .def/.endef: ignored.

The specified directive is not allowed within a *.def/.endef* pair. *.def/.endef* directives are used for specifying debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, note that:

1. you want to use the .line directive to specify the line number information for the symbol, and
2. you cannot nest .def/.endef directives.

.loc outside of .text.

The *.loc* directive must be specified in a *.text* section. The assembler has seen this directive in a non-*.text* section. The directive has no effect.

.loc pseudo-op inside .def/.endef: ignored.

The specified directive is not allowed within a *.def/.endef* pair. *.def/.endef* directives are used for specifying debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, note that:

1. you want to use the .line directive to specify the line number information for the symbol, and
2. you cannot nest .def/.endef directives.

.palign not specified in a code section. .palign ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.pbyte not specified in a code section. .pbyte ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.pfill not specified in a code section. .pfill ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.pfill size clamped to 8.

The size argument (second argument) of the *.fill* directive specified was greater then eight. The maximum size allowed is eight.

.pfillvalue expects a constant positive byte value. 0xXX assumed.

The *.pfillvalue* directive was specified with an argument that is not a constant positive byte value. The last *.pfillvalue* value that was specified will be used as if this directive did not exist.

.pfillvalue not specified in a code section. .pfillvalue ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.pword not specified in a code section. .pword ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

© 2008 Microchip Technology Inc.

.size pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a *.def/.endef* pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a *.def* directive before specifying this directive.

.scl pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a *.def/.endef* pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a *.def* directive before specifying this directive.

.tag pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a *.def/.endef* pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a *.def* directive before specifying this directive.

.type pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a *.def/.endef* pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a *.def* directive before specifying this directive.

.val pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a *.def/.endef* pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a *.def* directive before specifying this directive.

A

Alignment too large: 2^15 assumed.

An alignment greater than 2^15 was requested. 2^15 is the largest alignment request that can be made.

B

badly formed .dim directive ignored

The arguments for the .dim directive were unable to be parsed. This directive is used to specify debugging information and normally is only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, the arguments for the .dim directive are constant integers separated by a comma.

D

Directive not specified in a code section. Directive ignored.

The directive on the indicated line must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

E

error setting flags for "*section_name*": *error_message.*

If this warning is displayed, then the GNU code has changed as the if statement always evaluates false.

Expecting even address. Address will be rounded.

The absolute address specified for a CALL or GOTO instruction was odd. The address is rounded up. You will want to ensure that this is the intended result.

Expecting even offset. Offset will be rounded.

The PC-relative instruction at this line contained an odd offset. The offset is rounded up to ensure that the PC-relative instruction is working with even addresses.

I

Ignoring changed section attributes for *section_name.*

This section's attributes have already been set, and the new attributes do not match those previously set.

Ignoring fill value in absolute section.

A fill argument cannot be specified for either the *.org* or *.porg* directive when the current section is absolute.

Implied attributes for section 'name' are deprecated

Certain section names have implied attributes. In this case, a section was defined without listing its implied attributes. For clarity and future compatibility, section attributes should be listed explicitly. See **Section 6.3 "Directives that Define Sections"** for more information.

L

Line numbers must be positive integers

The line number argument of the *.ln* or *.loc* directive was less than or equal to zero after specifying debugging information for a function. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, note that function symbols can only exist on positive line numbers.

M

Macro 'name' has a previous definition

A macro has been redefined without removing the previous definition with the *.purgem* directive.

mismatched .eb

The assembler has seen a *.eb* directive without first seeing a matching *.bb* directive. The *.bb* and *.eb* directives are the begin block and end block directives and must always be specified in pairs.

O

Overflow/underflow for .long may lose significant bits

A constant value specified in a *.long* directive is too large and will lose significant bits when encoded.

© 2008 Microchip Technology Inc.

Q

Quoted section flags are deprecated, use attributes instead

Previous version of the assembler recommended the use of single character section flags. For clarity and future compatibility, attribute names should be used instead.

R

Repeat argument < 0. .fill ignored

The repeat argument (first argument) of the *.fill* directive specified was less than zero. The repeat argument must be an integer that is greater than or equal to zero.

Repeat argument < 0. .pfill ignored

The repeat argument (first argument) of the *.pfill* directive specified was less than zero. The repeat argument must be an integer that is greater than or equal to zero.

S

Size argument < 0. .fill ignored

The size argument (second argument) of the *.fill* directive specified was less than zero. The size argument must be an integer that is between zero and eight, inclusive. If the size argument is greater than eight, it is deemed to have a value of eight.

Size argument < 0. .pfill ignored

The size argument (second argument) of the *.pfill* directive specified was less than zero. The size argument must be an integer that is between zero and eight, inclusive. If the size argument is greater than eight, it is deemed to have a value of eight.

'symbol_name' symbol without preceding function

A *.bf* directive was seen without the preceding debugging information for the function symbol. This directive is used to specify debugging information and normally is only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first *.def* the function symbol and give it a *.type* of function (C_FCN = 101).

T

tag not found for .tag *symbol_name*

This warning should not be seen unless the assembler was unable to create the given symbol name. You may want to follow up on this warning with the GNU folks. It looks like the code used to generate this warning if the symbol name was not in its tag hash. Code was added that will ensure to create the symbol if it is not in the tag hash. This means that the only way this warning can be reached is if the symbol could not be created.

U

unexpected storage class sclass

The assembler is processing the *.endef* directive and has either seen a storage class that it does not recognize or has not seen a storage class. This directive is used to specify debugging information and normally is only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must specify a storage class using the *.scl* directive, and that storage class cannot be one of the following:

1. Undefined static (C_USTATIC = 14)
2. External definition (C_EXTDEF = 5)
3. Undefined label (C_ULABEL = 7)
4. Dummy entry (end of block) (C_LASTENT = 20)
5. Line # reformatted as symbol table entry (C_LINE = 104)
6. Duplicate tag (C_ALIAS = 105)
7. External symbol in dmert public library (C_HIDDEN = 106)
8. Weak symbol - GNU extension to COFF (C_WEAKEXT = 127)

unknown section attribute '*flag*'

The *.section* directive does not recognize the specified section flag. Please see **Section 6.3 "Directives that Define Sections"**, for the supported section flags.

unsupported section attribute 'i'

The .section directive does not support the "i" section flag for COFF. Please see **Section 6.3 "Directives that Define Sections"**, for the supported section flags.

unsupported section attribute 'l'

The .section directive does not support the "l" section flag for COFF. Please see **Section 6.3 "Directives that Define Sections"**, for the supported section flags.

unsupported section attribute 'o'

The .section directive does not support the "o" section flag for COFF. Please see **Section 6.3 "Directives that Define Sections"**, for the supported section flags.

V

Value *get* truncated to use.

The fill value specified for either the *.skip*, *.pskip*, *.space*, *.pspace*, *.org* or *.porg* directive was larger than a single byte. The value has been truncated to a byte.

A.6 MESSAGES

The assembler generates messages when a non-critical assumption is made so that the assembler could continue assembling a flawed program. Messages may be ignored. However, messages can sometimes point out bugs in your program.

© 2008 Microchip Technology Inc.

Appendix B. Linker Errors/Warnings

B.1 INTRODUCTION

MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB LINK30) generates errors and warnings. A descriptive list of these outputs is shown here.

B.2 HIGHLIGHTS

Topics covered in this appendix are:

- Errors
- Warnings

B.3 ERRORS

Symbols

% by zero

Modulo by zero is not computable.

/ by zero

Division by zero is not computable.

A

A heap is required, but has not been specified.

A heap must be specified when using Standard C input/output functions.

Address 0x8 of *filename* section .reset is not within region reset

This error indicates a problem with the linker script. Normally section .reset is created by the linker script and includes a single GOTO instruction. If a linker script is included in the link as an input file, it will augment the built-in script instead of replacing it. Then section .reset will be created twice, resulting in an overflow. To correct this error, specify `--script` or `-T` on the link command before the linker script file name.

Address *addr* of *filename* section *secname* is not within region *region.*

Section *secname* has overflowed the memory region to which it was assigned.

C

Cannot access symbol (*name*) with file register addressing. Value must be less than 8192.

name is not located in near address space. A read or write of *name* could not be resolved with the small data memory model.

Cannot access symbol (name) at an odd address.

Instructions that operate on word-sized data require operands to be allocated at even addresses.

cannot move location counter backwards (from *address1* to *address2*).

The location counter can be advanced but it cannot be moved backwards. An operation is attempting to move it from *address1* backwards to *address2*.

cannot open linker script file *name*

Unable to open the specified linker script file. Check the file name and/or the path.

cannot open *name*:

Cannot open the input file *name*. Check for correct spelling, extension or path.

cannot PROVIDE assignment to location counter

The PROVIDE keyword may not be used to make an assignment to the location counter.

Can not use dmaoffset on a symbol (name) that is not located in a dma section.

The dmaoffset() operator can only be used on symbols that are located in dma memory.

Cannot use *operator* on a symbol (name) that is not located in an executable or read-only section.

The following operators can be applied to symbols in executable or read-only sections only: `tbloffset()`, `psvoffset()`, `tblpage()`, `psvpage()`, `handle()`, `paddr()`.

Cannot use relocation type *reloc* on a symbol (name) that is located in an executable section.

An attempt was made to use a symbol in an executable section as a data address. To reference an executable symbol in a data context, the `psvoffset()` or `tbloffset()` operator is required.

Could not allocate data memory

The linker could not find a way to allocate all of the sections that have been assigned to region 'data'.

Could not allocate program memory

The linker could not find a way to allocate all of the sections that have been assigned to region 'program'.

Could not allocate eedata memory

The linker could not find a way to allocate all of the sections that have been assigned to region 'eedata'.

Could not allocate section '*name*', because 'ymemory,near' is not a valid combination on this device

The linker could not alllocate section name because the combination of section attributes [`ymemory`,`near`] is not valid on the current device.

Could not allocate section *secname* at address *addr*.

An address has been specified for *secname* that conflicts with another section or the limit of memory.

D

Data region overlaps PSV window (%d bytes).

The data region address range must be less than the start address for the PSV window. This error occurs when the C compiler's "constants in code" option is selected and more than 32K of data memory is required for program variables.

© 2008 Microchip Technology Inc.

--data-init and --no-data-init options can not be used together.

`--data-init` creates a special output section named `.dinit` as a template for the run-time initialization of data, `--no-data-init` does not. Only one option can be used.

__DMA_BASE is needed, but not defined (check linker script?)

__DMA_END is needed, but not defined (check linker script?)

The symbols __DMA_BASE and __DMA_END must be defined in order to allocate variables or sections in dma memory. By convention these symbols are defined in the linker script for a particular device, if that device supports dma memory.

E

EOF in comment.

An end-of-file marker (EOF) was found in a comment.

F

op* forward reference of section *secname.

The section name being used in the operation has not been defined yet.

G

--gc-sections and -r may not be used together.

Do not use `--gc-sections` option which enables garbage collection of unused input sections with the `-r` option which generates relocatable output.

H

--handles and --no-handles options cannot be used together

`--handles` supports far code pointers; `--no-handles` does not. Only one option can be used.

I

includes nested too deeply.

`include` statements should be nested no deeper than 10 levels.

Illegal value for DO instruction offset (-2, -1 or 0).

These values are not permitted.

invalid assignment to location counter.

The operation is not a valid assignment to the location counter.

invalid hex number '*num.*'

A hexadecimal number can only use the digits 0-9 and A-F (or a-f). The number is identified as a hex value by using 0x as the prefix.

invalid syntax in flags.

The region attribute flags must be `w`, `x`, `a`, `r`, `i` and/or `l`. ('`!`' is used to invert the sense of any following attributes.) Any other letters or symbols will produce the invalid syntax error.

M

macros nested too deeply.

Macros should be nested no deeper than 10 levels.

missing argument to -m.

The emulation option (`-m`) requires a name for the emulation linker.

N

Near data space has overflowed by *num* bytes.

Near data space must fit within the lowest 8K address range. It includes the sections `.nbss` for static or non-initialized variables, and `.ndata` for initialized variables.

no input files.

The 16-bit linker requires at least one object file.

non constant address expression for section *secname.*

The address for the specified section must be a constant expression.

nonconstant expression for *name.*

`name` must be a constant expression.

Not enough contiguous memory for section *secname*.

The linker attempted to reallocate program memory to prevent a read-only section from crossing a PSV page boundary, but a memory solution could not be found.

Not enough memory for heap (*num* bytes available).

There was not enough memory free to allocate the heap.

Not enough memory for stack (*num* bytes available).

There was not enough memory free to allocate the minimum-sized stack.

O

object *name* was created for the *processor* which is not instruction set compatible with the target *processor*.

An object file to be linked was created for a different processor family than the link target, and the instruction sets are not compatible.

Odd values are not permitted for a new location counter.

When a `.org` or `.porg` directive is used in a code section, the new location counter must be even. This error also occurs if an odd value is assigned to the special DOT variable.

P

--pack-data and --no-pack-data options cannot be used together.

`--pack-data` fills the upper byte of each instruction word in the data initialization template with data. `--no-pack-data` does not. Only one option can be used.

PSV section *secname* exceeds 32 Kbytes (actual size = *num*).

The constant data table may not exceed the program memory page size that is implied by the PSVPAG register which is 32 Kbytes.

R

region *region* is full (*filename* section *secname*).

The memory region *`region`* is full, but section *`secname`* has been assigned to it.

--relax and -r may not be used together.

The option `--relax` which turns relaxation on may not be used with the `-r` option which generates relocatable output.

© 2008 Microchip Technology Inc.

relocation truncated to fit: PC RELATIVE BRANCH *name.*

The relative displacement to function *name* is greater than 32K instruction words. A function call to *name* could not be resolved with the small code memory model.

relocation truncated to fit: *relocation_type name.*

The relocated value of *name* is too large for its intended use.

S

section .handle must be allocated low in program memory.

A custom linker script has organized memory such that section .handle is not located within the first 32K words of program memory.

section *secname1* [*startaddr1—startaddr2*] overlaps section *secname2* [*startaddr1—startaddr2*]\n"),

There is not enough region memory to place both of the specified sections or they have been assigned to addresses that result in an overlap.

-shared not supported.

The option `-shared` is not supported by the 16-bit linker.

Symbol (name) is not located in an executable section.

An attempt was made to call or branch to a symbol in a bss, data or readonly section.

syntax error.

An incorrectly formed expression or other syntax error was encountered in a linker script.

U

undefined symbol '__reset' referenced in expression.

The library `-lpic30` is required, or some other input file that contains a start-up function. This error may result from a version or architecture mismatch between the linker and library files.

undefined symbol '*symbol*' referenced in expression.

The specified symbol has not been defined.

undefined reference to '_Ctype'

undefined reference to '_Tolotab'

undefined reference to '_Touptab'

These errors indicate a version mismatch between include files and library files, or between library files and precompiled object files. Make sure that all object files to be linked have been compiled with the same version of the 16-bit compiler. If you are using a precompiled object or library file from another vendor, request an update that is compatible with the latest version of the compiler.

undefined reference to '*symbol.*'

The specified symbol has not been defined. Either an input file has been omitted, a library file is incomplete or a circular reference exists between libraries. Circular references can be resolved with the `--start-group`, `--end-group` options.

unrecognized emulation mode: *target*
Supported emulations:

The specified target is not an emulation mode supported by the linker. The list of supported emulations follows the error message.

unrecognized -a option '*argument.*'

The `-a` option is not supported by 16-bit devices; so it is ignored.

unrecognized -assert option '*option.*'

The `-assert` option is not supported by 16-bit devices; so it is ignored.

unrecognized option '*option*'.

The specified option is not a recognized linker option. Check the option and its usage information with the `--help` option.

op* uses undefined section *secname.

The section referred to in the operation is not defined.

X

X data space has overflowed by *num* bytes.

The address range for X data space must be less than the start of Y data space. The start of Y data space is determined by the processor used.

Y

__YDATA_BASE is needed, but not defined.

By convention, the starting address of Y data memory for a particular device is defined in linker scripts using this name. The linker needed this information to allocate a section with `xmemory` or `ymemory` attribute, but could not find it.

B.4 WARNINGS

A

Addresses specified for READONLY section *name* are not valid for PSV window.

The application has specified absolute addresses for a read-only section that are not consistent with the PSV window. If two addresses have been specified, the least-significant 15 bits should be identical. Also, the most significant bit of the virtual address should be set.

C

cannot find entry symbol *symbol* defaulting to *value.*

The linker can't find the entry symbol, so it will use the first address in the text section. This message may occur if the `-e` option incorrectly contains an equal sign ('=') in the option (i.e., `-e=0x200)`.

common of '*name*' overridden by definition
defined here.

The specified variable name has been declared in more than one file with one instance being declared as common. The definition will override the common symbol.

common of '*name*' overridden by larger common
larger common is here.

The specified variable name has been declared in more than one file with different values. The smaller value will be overridden with the larger value.

common of '*name*' overriding smaller common
smaller common is here.

The specified variable name has been declared in more than one file with different values. The first one encountered was smaller and will be overridden with the larger value.

© 2008 Microchip Technology Inc.

D

data initialization has been turned off, therefore section *secname* will not be initialized.

The specified section requires initialization but data initialization has been turned off so the initial data values are discarded. Storage for the data sections will be allocated as usual.

data memory region not specified. Using default upper limit of *addr*.

The linker has allocated a maximum-size stack. Since the data memory region was not specified, a default upper limit was used.

definition of '*name*' overriding common
common is here.

The specified variable name has been declared in more than one file with one instance being declared as common. The definition will override the common symbol.

H

--heap option overrides HEAPSIZE symbol.

The `--heap` option has been specified and the `HEAPSIZE` symbol has been defined but they have different values so the `--heap` value will be used.

I

initial values were specified for a non-loadable data section (*name*). These values will be ignored.

By definition, a persistent data section implies data that is not initialized; therefore the values are discarded. Storage for the section will be allocated as usual.

M

multiple common of '*name*'
previous common is here.

The specified variable name has been declared in more than one file.

N

no memory region specified for section '*secname*'

Section *`secname`* has been assigned to a default memory region, but other non-default regions are also defined.

O

object *name* was created for the *processor* and references register *name*

An object file to be linked was created for a different processor family than the link target, and references a Special Function Register (SFR) that may not be compatible.

P

program memory region not specified. Using default upper limit of *addr*.

The linker has reallocated program memory to prevent a read-only section from crossing a PSV page boundary. Since the program memory region was not specified, a default upper limit was used.

R

READONLY section *secname* at *addr* crosses a PSVPAG boundary.

Address *addr* has been specified for a read-only section, causing it to cross a PSV page boundary. To allow efficient access of constant tables in the PSV window, it is recommended that the section should not cross a PSVPAG boundary.

'-retain-symbols-file' overrides '-s' and '-S'

If the strip all symbols option (`-s`) or the strip debug symbols option (`-S`) is used with `--retain-symbols-file` *FILE* only the symbols specified in the file will be kept.

S

--stack option overrides STACKSIZE symbol.

The `--stack` option has been specified and the `STACKSIZE` symbol has been defined but they have different values so the `--stack` value will be used.

T

target processor '*name*' does not match linker script

The link target processor specified on the command line does not match the linker script `OUTPUT_ARCH` command. The processor name specified on the command line takes precedence.

© 2008 Microchip Technology Inc.

Appendix C. Deprecated Features

C.1 INTRODUCTION

The features described below are considered to be obsolete and have been replaced with more advanced functionality. Projects which depend on deprecated features will work properly with versions of the language tools cited. The use of a deprecated feature will result in a warning; programmers are encouraged to revise their projects in order to eliminate any dependancy on deprecated features. Support for these features may be removed entirely in future versions of the language tools.

C.2 HIGHLIGHTS

Topics covered in this appendix are:

- Assembler Directives that Define Sections
- Reserved Section Names with Implied Attributes

C.3 ASSEMBLER DIRECTIVES THAT DEFINE SECTIONS

The following `.section` directive format was deprecated in v1.30. The new directive format may be found in **Section 6.3 "Directives that Define Sections"**.

.section name [, "flags"]

Definition

Assembles the following code into a section named `name`. If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

- `b` bss section (uninitialized data)
- `n` Section is not loaded
- `d` Data section (initialized data)
- `r` Read-only data section (PSV window)
- `x` Executable section

If the `n` flag is used by itself, the section defaults to uninitialized data.

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loadable data.

The following section names are recognized:

TABLE C-1: SECTION NAMES

Section Name	Default Flag
.text	x
.data	d
.bss	b

Note: Ensure that double quotes are used around flags. If the optional argument to the .section directive is not quoted, it is taken as a sub-section number. Remember, a single character in single quotes (i.e., 'b') is converted by the preprocessor to a number.

Example

```
.section .const, "r"
   ; The following symbols (C1 and C2) will be placed
   ; in the named section ".const".
C1:   .word 0x1234
C2:   .word 0x5678
```

C.4 RESERVED SECTION NAMES WITH IMPLIED ATTRIBUTES

Implied attributes for the section names in the table below were deprecated in v1.30.

Reserved Name	Implied Attribute(s)
.xbss	bss, xmemory
.xdata	data, xmemory
.nbss	bss, near
.ndata	data, near
.ndconst	data, near
.pbss	bss, persist
.dconst	data
.ybss	bss, ymemory
.ydata	data, ymemory
.const	psv
.eedata	eedata

See **Section 6.3 "Directives that Define Sections"** for more information.

© 2008 Microchip Technology Inc.

Appendix D. MPASM™ Assembler Compatibility

D.1 INTRODUCTION

This information is provided for users of the MPASM assembler, Microchip Technology's PIC MCU device assembler. MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB ASM30) is not compatible with the MPASM assembler. Details on the compatibility issues, as well as examples and suggestions for migrating to the 16-bit assembler, are shown here.

For the lastest information on the MPASM assembler, see on-line help for this tool in MPLAB IDE.

D.2 HIGHLIGHTS

Topics covered in this appendix are:

- Compatibility
- Examples
- Converting PIC18F MCU Assembly Code to dsPIC30F DSC Assembly Code

D.3 COMPATIBILITY

Users migrating from MPASM assembler will face the following compatibility issues:

- Differences in Assembly Language
- Differences in Command Line Options
- Differences in Directives

D.3.1 Differences in Assembly Language

The instruction set for 16-bit devices has been expanded to support the new functionality of the architecture. Please refer to individual 16-bit device data sheets and *"dsPIC30F/33F Programmer's Reference Manual"* (DS70157) for more details.

In addition, the following syntactical differences exist:

- A colon ':' must precede label definitions suffix.
- Directives must be preceded by a dot '.'.

D.3.2 Differences in Command Line Options

The 16-bit assembler command line is incompatible with the MPASM assembler command line. Table D-1 summarizes the command line incompatibilities.

TABLE D-1: COMMAND LINE INCOMPATIBILITIES

MPASM Assembler	MPLAB Assembler for PIC24 MCUs and dsPIC® DSCs	Description
/?, /h	--help	Display help
/a	Not supported(1)	Set hex file format
/c	Not supported(2)	Enable/Disable case sensitivity
/dSYM	--defsym SYM=VAL	Define symbol
/e	Not supported(3)	Enable/Disable/Set Path for error file
/l	-a[sub-option...]	Enable/Disable/Set Path for listing file
/m	-am	Enable/Disable macro expansion
/o	-o OBJFILE	Enable/Disable/Set Path for object file
/p	-A ARCH	Set the processor type
/q	--verbose	Enable/Disable quiet mode (suppress screen output)
/r	Not Supported(4)	Defines default radix
/t	Not Supported(5)	List file tab size
/w0 /w1 /w2	-W, --no-warn	All messages Errors and warnings Errors only
/x	Not Supported(6)	Enable/Disable/Set Path for cross reference file

Note 1: The 16-bit assembler does not generate hex files. It is only capable of producing relocatable object files.

2: Assembler mnemonics and directives are not case sensitive; however, labels and symbols are. See **Chapter 5. "Assembler Symbols"** and **Chapter 6. "Assembler Directives"**, for more details.

3: Diagnostic messages are sent to standard error. It is possible to redirect standard error to a file using operating system commands.

4: The default radix in the 16-bit assembler is decimal. See **Section 3.5.1.1 "Integers"**, for a complete description.

5: The 16-bit assembler listing files utilize the tab settings of the operating system.

6: The 16-bit assembler does not generate cross-reference files. See the 16-bit linker section of this manual for information on creating cross-referenced files.

© 2008 Microchip Technology Inc.

D.3.3 Differences in Directives

Directives are assembler commands that appear in the source code but are not translated directly into opcodes. They are used to control the assembler: its input, output and data allocation. The dsPIC30 assembler does not support several MPASM directives or supports the directives differently. Table D-2 summarizes the assembler directive incompatibilities:

TABLE D-2: ASSEMBLER DIRECTIVE INCOMPATIBILITIES

MPASM™ Assembler	MPLAB® Assembler for PIC24 MCUs and dsPIC® DSCs	Description
__BADRAM	Not supported	Specify invalid RAM locations
BANKISEL	Not supported	Generate RAM bank selecting code for indirect addressing
BANKSEL	Not supported	Generate RAM bank selecting code
CBLOCK	Not supported	Define a block of constants
CODE	.text	Begins executable code section
__CONFIG	Not supported	Specify configuration bits
CONSTANT	.equ (syntax)	Declare symbol constant
DA	.ascii (syntax)	Store strings in program memory
DATA	Not supported	Create numeric and text data
DB	.byte	Declare data of one byte
DE	Not supported	Define EEPROM data
#DEFINE	.macro (syntax)	Define a text substitution label
DT	Not supported	Define table
DW	.word	Declare data of one word
ELSE	.else	Begin alternative assembly block to IF
END	.end	End program block
ENDC	Not supported	End an automatic constant block
ENDIF	.endif	End conditional assembly block
ENDM	.endm (not equivalent)	End a macro definition
ENDW	Not supported	End a while loop
EQU	.equ (syntax)	Define an assembly constant
ERROR	.error	Issue an error message
ERRORLEVEL	Not supported	Set error level
EXITM	Not supported	Exit from a macro
EXPAND	Not supported	Expand a macro listing
EXTERN	.extern	Declares an external label
FILL	.fill (syntax)	Fill memory
GLOBAL	.global	Exports a defined label
IDATA	.data	Begins initialized data section
__IDLOCS	Not supported	Specify ID locations
IF	.if	Begin conditionally assembled code block
IFDEF	.ifdef	Execute if symbol has been defined
IFNDEF	.ifndef	Execute if symbol has not been defined
#INCLUDE	.include (syntax)	Include additional source file

TABLE D-2: ASSEMBLER DIRECTIVE INCOMPATIBILITIES (CONTINUED)

MPASM™ Assembler	MPLAB® Assembler for PIC24 MCUs and dsPIC® DSCs	Description
LIST	.psize (not equivalent)	Listing options
LOCAL	Not supported	Declare local macro variable
MACRO	.macro (not equivalent)	Declare macro definition
__MAXRAM	Not supported	Specify maximum RAM address
MESSG	Not supported	Create user defined message
NOEXPAND	Not supported	Turn off macro expansion
NOLIST	.nolist	Turn off listing output
ORG	.org (not equivalent)	Set program origin
PAGE	.eject	Insert listing page eject
PAGESEL	Not supported	Generate ROM page selecting code
PROCESSOR	Not supported	Set processor type
RADIX	Not supported	Specify default radix
RES	.skip	Reserve memory
SET	.set (syntax)	Define an assembler variable
SPACE	Not supported	Insert blank listing lines
SUBTITLE	.sbttl	Specify program subtitle
TITLE	.title	Specify program title
UDATA	.bss	Begins uninitialized data section
UDATA_ACS	Not supported	Begins access uninitialized data section
UDATA_OVR	Not supported	Begins overlayed uninitialized data section
UDATA_SHR	Not supported	Begins shared uninitialized data section
#UNDEFINE	Not supported	Delete a substitution label
VARIABLE	.set (not equivalent)	Declare symbol variable
WHILE	Not supported	Perform loop while condition is true

D.4 EXAMPLES

EXAMPLE D-1: EQU VS .EQU

In MPASM assembler, the EQU directive is used to define an assembler constant.

```
CORCONH EQU 0x45
```

In the 16-bit assembler, the .equ directive is used to define an assembler constant.

```
.equ CORCONH, 0x45
```

EXAMPLE D-2: UDATA VS .BSS

In MPASM assembler, the UDATA directive is used to begin an uninitialized data section.

```
UDATA
```

In the 16-bit assembler, the .bss directive is used to begin an uninitialized data section.

```
.bss
```


© 2008 Microchip Technology Inc.

D.5 CONVERTING PIC18F MCU ASSEMBLY CODE TO dsPIC30F DSC ASSEMBLY CODE

In order to convert your PIC18FXXX code to code that can be used with a dsPIC30FXXXX device, you must understand the following:

- Direct Translations
- Emulation Model

D.5.1 Direct Translations

Table D-3 lists all PIC18FXXX instructions and their corresponding replacements in the dsPIC30FXXXX instruction set. The assumption is made that all of the dsPIC30FXXXX instructions that use file registers as an operand can address at least 0x2000 bytes. Accessing file registers beyond this limit requires the use of indirection, and is not taken into consideration in this table. Also, the access RAM concept is not implemented on the dsPIC30FXXXX parts as all directly addressable memory, including special function registers, falls into the 0x0000-0x1FFF range.

TABLE D-3: PIC18FXXX INSTRUCTIONS

PIC18CXXX Legend	dsPIC30FXXXX Legend
k = literal value	Slit10 = 10-bit signed literal
	lit10 = 10-bit unsigned literal
f = file register address	Slit16 = 16-bit signed literal
a = access memory bit	lit23 = 23-bit unsigned literal
n = relative branch displacement	WREG = W0
b = bit position	f = file register
	bit3 = bit position (0...7)
	PROD = W2

TABLE D-4: INSTRUCTION SET COMPARISON

PIC18FXXX Instruction	dsPIC30FXXXX Instruction	Description	Result Location
ADDLW k	ADD.b #lit10,W0	Add literal to WREG	WREG
ADDWF f,0,a	ADD.b f,WREG	Add file register contents to WREG	WREG
ADDWF f,1,a	ADD.b f	Add WREG to file register contents	file register (f)
ADDWFC f,0,a	ADDC.b f,WREG	Add with carry file register contents to WREG	WREG
ADDWFC f,1,a	ADDC.b f	Add with carry WREG to file register contents	file register (f)
ANDLW k	AND.b #lit10,W0	Bit-wise AND literal with WREG	WREG
ANDWF f,0,a	AND.b f,WREG	Bit-wise AND file register contents with WREG	WREG
ANDWF f,1,a	AND.b f	Bit-wise AND WREG with file register contents	file register (f)
BC n	BRA C,Slit16	Branch to relative location if Carry bit is set	N/A
BCF f,b,a	BCLR.b f,#bit3	Clear single bit in file register	file register (f)
BN n	BRA N,Slit16	Branch to relative location if Negative bit is set	N/A
BNC n	BRA NC,Slit16	Branch to relative location if Carry bit is clear	N/A
BNN n	BRA NN,Slit16	Branch to relative location if Negative bit is clear	N/A
BNOV n	BRA NOV,Slit16	Branch to relative location if Overflow bit is clear	N/A
BNZ n	BRA NZ,Slit16	Branch to relative location if Zero bit is clear	N/A
BRA n	BRA Slit16	Branch to relative location	N/A
BSF f,b,a	BSET.b f,#bit3	Set single bit in file register	file register (f)
BTFSC f,b,a	BTSC.b f,#bit3	Test single bit, skip next instruction if clear	N/A

Note 1: No direct translation.

2: No direct translation. See **Section D.5.2 "Emulation Model"**.

TABLE D-4: INSTRUCTION SET COMPARISON (CONTINUED)

PIC18FXXX Instruction	dsPIC30FXXXX Instruction	Description	Result Location
BTFSS f,b,a	BTSS.b f,#bit3	Test single bit, skip next instruction if set	N/A
BTG f,b,a	BTG.b f,#bit3	Toggle single bit	file register (f)
BOV n	BRA OV,Slit16	Branch to relative location if Overflow bit is set	N/A
BZ n	BRA Z,Slit16	Branch to relative location if Zero bit is set	N/A
CALL k,0	CALL lit23	Call subroutine	N/A
CALL k,1	(Note 1)	Call subroutine using shadow registers	N/A
CLRF f,a	CLR.b f	Clear file register	file register (f)
CLRWDT	CLRWDT	Clear watchdog timer	WDT
COMF f,0,a	COM.b f,WREG	Complement file register	WREG
COMF f,1,a	COM.b f	Complement file register	file register (f)
CPFSEQ f,a	(Note 1)	Compare f with WREG, skip next instruction if equal	N/A
CPFSGT f,a	(Note 1)	Compare f with WREG, skip next instruction if f > WREG	N/A
CPFSLT f,a	(Note 1)	Compare f with WREG, skip next instruction if f < WREG	N/A
DAW	DAW.b W0	Decimal adjust WREG	WREG
DECF f,0,a	DEC.b f,WREG	Decrement f into WREG	WREG
DECF f,1,a	DEC.b f	Decrement f	file register (f)
DECFSZ f,0,a	(Note 1)	Decrement f into WREG, skip next instruction if zero	WREG
DECFSZ f,1,a	(Note 1)	Decrement f, skip next instruction if zero	file register (f)
DECFSNZ f,0,a	(Note 1)	Decrement f into WREG, skip next instruction if not zero	WREG
DECFSNZ f,1,a	(Note 1)	Decrement f, skip next instruction if not zero	file register (f)
GOTO k	GOTO lit23	Branch to absolute address	N/A
INCF f,0,a	INC.b f,WREG	Increment f into WREG	WREG
INCF f,1,a	INC.b f	Increment f	file register (f)
INCFSZ f,0,a	(Note 1)	Increment f into WREG, skip next instruction if zero	WREG
INCFSZ f,1,a	(Note 1)	Increment f, skip next instruction if zero	file register (f)
INCFSNZ f,0,a	(Note 1)	Increment f into WREG, skip next instruction if not zero	WREG
INCFSNZ f,1,a	(Note 1)	Increment f, skip next instruction if not zero	file register (f)
IORLW k	IOR.b #lit10,W0	Bit-wise inclusive-or literal with WREG	WREG
IORWF f,0,a	IOR.b f,WREG	Bit-wise inclusive-or file register contents with WREG	WREG
IORWF f,1,a	IOR.b f	Bit-wise inclusive-or WREG with file register contents	file register (f)
LFSR f,k	(Note 2)	Load literal value into file select register	FSRx
MOVF f,0,a	MOV.b f,WREG	Move file register contents into WREG	WREG
MOVF f,1,a	MOV.b f	Set status flags based on file register contents	N/A
MOVFF fs,fd	(Note 2)	Move file register contents to file register	file register (fd)
MOVLB k	N/A - no banking	Set current bank	BSR
MOVLW k	MOV.b #lit10,W0	Load literal value into WREG	WREG
MOVWF f,a	MOV.b WREG,f	Move WREG contents to file select register	file register (f)
MULLW k	(Note 2)	Multiply WREG by literal	PROD
MULWF f,a	MUL.b f	Multiply WREG by file register contents	PROD
NEGF f,a	NEG.b f	Negate file register contents	file register (f)
NOP	NOP	No operation	N/A

Note 1: No direct translation.

2: No direct translation. See **Section D.5.2 "Emulation Model"**.

© 2008 Microchip Technology Inc.

TABLE D-4: INSTRUCTION SET COMPARISON (CONTINUED)

PIC18FXXX Instruction	dsPIC30FXXXX Instruction	Description	Result Location
POP	SUB W15,#4,W15	Discard the top-of-stack	N/A
PUSH	RCALL .+2	Push current program counter onto stack	N/A
RCALL n	RCALL Slit16	Call subroutine at relative offset	N/A
RESET	RESET	Reset processor	N/A
RETFIE 0	RETFIE	Return from interrupt	N/A
RETFIE 1	POP.s RETFIE	Return from interrupt, restoring context from shadow regs	N/A
RETLW k	RETLW.b #lit10,W0	Return from subroutine with a literal value in WREG	WREG
RETURN 0	RETURN	Return from subroutine	N/A
RETURN 1	POP.s RETURN	Return from subroutine, restoring context from shadow regs	N/A
RLCF f,0,a	RLC.b f,WREG	Rotate contents of file register left through carry	WREG
RLCF f,1,a	RLC.b f	Rotate contents of file register left through carry	file register (f)
RLNCF f,0,a	RLNC.b f,WREG	Rotate contents of file register left (without carry)	WREG
RLNCF f,1,a	RLNC.b f	Rotate contents of file register left (without carry)	file register (f)
RRCF f,0,a	RRC.b f,WREG	Rotate contents of file register right through carry	WREG
RRCF f,1,a	RRC.b f	Rotate contents of file register right through carry	file register (f)
RRNCF f,0,a	RRNC.b f,WREG	Rotate contents of file register right (without carry)	WREG
RRNCF f,1,a	RRNC.b f	Rotate contents of file register right (without carry)	file register (f)
SETF f,a	SETM.b f	Set all bits in file register	file register (f)
SLEEP	(Note 2)	Put processor into sleep mode	N/A
SUBFWB f,0,a	SUBBR.b f,WREG	Subtract file register contents from WREG with borrow	WREG
SUBFWB f,1,a	SUBBR.b f	Subtract file register contents from WREG with borrow	file register (f)
SUBLW k	(Note 2)	Subtract WREG from literal	WREG
SUBWF f,0,a	SUB.b f,WREG	Subtract WREG from file register contents	WREG
SUBWF f,1,a	SUB.b f	Subtract WREG from file register contents	file register (f)
SUBWFB f,0,a	SUBB.b f,WREG	Subtract WREG from file register contents with borrow	WREG
SUBWFB f,1,a	SUBB.b f	Subtract WREG from file register contents with borrow	file register (f)
SWAPF f,0,a	(Note 2)	Swap nibbles of file register contents	WREG
SWAPF f,1,a	(Note 2)	Swap nibbles of file register contents	file register (f)
TBLRD	(Note 2)	Read value from program memory	TABLAT
TBLWT	(Note 2)	Write value to program memory	N/A
TSTFSZ f,a	(Note 2)	Skip next instruction if file register contents are zero	N/A
XORLW k	XOR.b #lit10,W0	Bit-wise exclusive-or WREG with literal	WREG
XORWF f,0,a	XOR.b f,WREG	Bit-wise exclusive-or WREG with contents of file register	WREG
XORWF f,1,a	XOR.b f	Bit-wise exclusive-or WREG with contents of file register	file register (f)

Note 1: No direct translation.

2: No direct translation. See **Section D.5.2 "Emulation Model"**.

D.5.2 Emulation Model

The PIC18FXXX parts can be modeled on a dsPIC30FXXXX by dedicating working registers to emulate PIC18FXXX special function registers.

TABLE D-5: REGISTERS TO EMULATE PIC18FXXX

Working Register	PIC18FXXX Equivalent
W0	WREG
W1	Scratch register
W2	PROD
W3	N/A – reserved for high-order 16-bits of multiplication
W4	TABLAT
W5	TBLPTR
W6	FSR0
W7	FSR1
W8	FSR2

Using these assignments, it is possible to emulate the remainder of the PIC18FXXX instructions that could not be represented by a single dsPIC30FXXXX instruction.

D.5.2.1 LFSR F,K

For FSR0:

```
MOV #k,W6
```

For FSR1:

```
MOV #k,W7
```

For FSR2:

```
MOV #k,W8
```

D.5.2.2 MOVFF FS,FD

This is equivalent to the following sequence of instructions:

```
MOV fs,W1
MOV W1,fd
```

D.5.2.3 MULLW K

If k <= 0x1f:

```
MUL.UU W0,#k,W2
```

If k > 0x1f:

```
MOV #k,W1
MUL.UU W0,W1,W2
```

D.5.2.4 SWAPF F,D,A

If d = 0:

```
MOV f,W0
SWAP.b W0
```

If d=1:

```
MOV f,W1
SWAP.b W1
MOV W1,f
```


© 2008 Microchip Technology Inc.

D.5.2.5 TBLRD

This instruction assumes that on the dsPIC30FXXXX part, only the lower two bytes of each instruction word are used.

TBLRD *:

```
TBLRDL [W5],W4
```

TBLRD *+:

```
TBLRDL [W5++],W4
```

TBLRD *-:

```
TBLRDL [W5--],W4
```

TBLRD +*:

```
TBLRDL [++W5],W4
```

D.5.2.6 TBLWT

This instruction assumes that on the dsPIC30FXXXX part, only the lower two bytes of each instruction word is used.

TBLWT *:

```
TBLWT W4,[W5]
```

TBLWT *+:

```
TBLWT W4,[W5++]
```

TBLWT *-:

```
TBLWT W4,[W5--]
```

TBLWT +*:

```
TBLWT W4,[++W5]
```

D.5.2.7 TSTFSZ F,A

This instruction can be emulated using a two-instruction sequence:

```
MOV f
BRA Z,1f
instruction to skip
1: next instruction
```

D.5.2.8 FSR ACCESSES

Use of the PIC18FXXX FSR complex addressing modes can be emulated by using the complex addressing modes of the dsPIC30FXXXX working registers. For example:

PIC18FXXX instruction: `ADDWF POSTINC1,1,0`

Effect:

1. Add the contents of the file register pointed to by FSR1 to WREG
2. Store the results in WREG
3. Post-increment FSR1

dsPIC30FXXXX sequence: `ADD.b W0,[W7],[W7++]`

© 2008 Microchip Technology Inc.

NOTES:

© 2008 Microchip Technology Inc.

Appendix E. MPLINK™ Linker Compatibility

E.1 INTRODUCTION

This information is provided for users of the MPLINK object linker, Microchip Technology's PIC MCU device linker. MPLAB Object Linker for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB LINK30) is not compatible with the MPLINK linker. Details on the compatibility issues, as well as examples and suggestions for migrating to the 16-bit linker, are shown here.

For the lastest information on the MPLINK linker, see on-line help for this tool in MPLAB IDE.

E.2 HIGHLIGHTS

Topics covered in this appendix are:

- Compatibility
- Migration to the 16-Bit Linker

E.3 COMPATIBILITY

The 16-bit linker command line is incompatible with the MPLINK command line. The following table summarizes the command line incompatibilities.

TABLE E-1: COMMAND LINE INCOMPATIBILITIES

MPLINK™ Linker	MPLAB® Linker for PIC24 MCUs and dsPIC® DSCs	Description
/?, /h	--help	Display help
/o	-o, --output	Specify output file. Default is a.out in both.
/m	-Map	Create map file
/l	-L, --library-path	Add directory to library search path
/k	-L1	Add directories to linker script search path
/n	Not supported(1)	Specify number of lines per listing page
/a	Not supported	Specify format of hex output file
/q	Not supported	Quiet mode
/d	Not supported(1)	Do not create an absolute listing file.

Note 1: The GNU linker does not create listing files. You can generate listing files for each object file using the GNU assembler.

E.4 MIGRATION TO THE 16-BIT LINKER

The 16-bit linker uses a sequential allocation algorithm and does not automatically fill in gaps that may appear due to alignment restrictions. In contrast, MPLINK linker uses a best-fit algorithm to fill available memory.

NOTES:

© 2008 Microchip Technology Inc.

Appendix F. MPLIB™ Librarian Compatibility

F.1 INTRODUCTION

This information is provided for users of the MPLIB object librarian, Microchip Technology's PIC MCU device librarian.The MPLAB Object Archiver/Librarian for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB LIB30) is not compatible with the MPLIB librarian. Details on the compatibility issues, as well as examples and suggestions for migrating to the 16-bit librarian, are shown here.

For the lastest information on the MPLIB librarian, see on-line help for MPLINK linker in MPLAB IDE.

F.2 HIGHLIGHTS

Topics covered in this appendix are:

- Compatibility
- Examples

F.3 COMPATIBILITY

The 16-bit archiver/librarian command line is incompatible with the MPLIB librarian command line. The following table summarizes the command line incompatibilities.

TABLE F-1: COMMAND LINE INCOMPATIBILITIES

MPLIB™ Librarian	MPLAB® Librarian for PIC24 MCUs and dsPIC® DSCs	Description
`/q`	Default mode	Quiet mode
`/c`	Default mode	Create library
`/t`	`-t`	List library
`/d`	`-d`	Delete member
`/r`	`-r`	Add or replace
`/x`	`-x`	Extract
`/?, /h`	`--help`	Display help

F.4 EXAMPLES

To create a library named `dsp` from three object modules named `fft.o`, `fir.o` and `iir.o`, use the following command line:

For MPLIB librarian to create `dsp.lib`:

```
MPLIB /c    dsp.lib  fft.o  fir.o  iir.o
```

For the 16-bit archiver/librarian to create `dsp.a`:

```
pic30-ar -r dsp.a    fft.o  fir.o  iir.o
```

To display the names of the object modules contained in a library file named dsp, use the following command line:

For MPLIB librarian:

```
MPLIB /t    dsp.lib
```

For the 16-bit archiver/librarian:

```
pic30-ar -t dsp.a
```


© 2008 Microchip Technology Inc.

Appendix G. Useful Tables

G.1 INTRODUCTION

Some useful tables are included for reference here.

G.2 HIGHLIGHTS

The tables are:

- ASCII Character Set
- Hexadecimal to Decimal Conversion

G.3 ASCII CHARACTER SET

		Most Significant Character							
Least Significant Character	**Hex**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	0	NUL	DLE	Space	0	@	P	`	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	Bell	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	–	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

G.4 HEXADECIMAL TO DECIMAL CONVERSION

This appendix describes how to convert hexadecimal to decimal. For each hex digit, find the associated decimal value. Add the numbers together.

High Byte				Low Byte			
Hex 1000	**Dec**	**Hex 100**	**Dec**	**Hex 10**	**Dec**	**Hex 1**	**Dec**
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

For example, hex A38F converts to 41871 as follows:

Hex 1000's Digit	Hex 100's Digit	Hex 10's Digit	Hex 1's Digit	Result
40960	768	128	15	41871 Decimal

© 2008 Microchip Technology Inc.

Appendix H. GNU Free Documentation License

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000, 2001, 2002 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

H.1 PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

H.2 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify, or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of

mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

© 2008 Microchip Technology Inc.

H.3 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

H.4 COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

H.5 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

c) State on the Title page the name of the publisher of the Modified Version, as the publisher.

d) Preserve all the copyright notices of the Document.

e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

h) Include an unaltered copy of this License.

i) Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

j) Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

k) For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

l) Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

m) Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

n) Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

o) Preserve any Warranty Disclaimers.

© 2008 Microchip Technology Inc.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

H.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

H.7 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

H.8 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

H.9 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

H.10 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

H.11 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

© 2008 Microchip Technology Inc.

Glossary

Absolute Section

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

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).

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

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

Archiver

A tool that creates and manipulates libraries.

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.

Assembler

A language tool that translates assembly language source code into machine code.

Assembly Language

A programming language that describes binary machine code in a symbolic form.

Assigned Section

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

Attribute, Section

Characteristics of sections, such as "executable", "readonly", or "data" that can be specified as flags in teh assembler `.section` directive.

Binary

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

Breakpoint, Hardware

An event whose execution will cause a halt.

Breakpoint, Software

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

A general-purpose programming language which features economy of expression, modern control flow and data structures and a rich set of operators.

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.

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.

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.

Configuration Bits

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

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.

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.

Device Programmer

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

© 2008 Microchip Technology Inc.

Digital Signal Controller

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

Digital Signal Processing

The computer manipulation of digital signals, commonly analog signals (sound or image) which have been converted to digital form (sampled).

Digital Signal Processor

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.

DSC

See Digital Signal Controller.

DSP

See Digital Signal Processor.

Endianess

Describes order of bytes in a multi-byte object.

Error File

A file containing error messages and diagnostics generated by a language tool.

Errors

Errors report problems that make it impossible to continue processing your program. When possible, errors identify the source file name and line number where the problem is apparent.

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.

Expressions

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

External Label

A label that has external linkage.

External Symbol

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

File Registers

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

Flash

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

GPR

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

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

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

Hex File

An ASCII file containing hexadecimal addresses and values (hex code) suitable for programming a device.

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 right-most digit counts ones, the next counts multiples of 16, then 16^2 = 256, etc.

ICD

In-Circuit Debugger. MPLAB ICD and MPLAB ICD 2 are Microchip's in-circuit debuggers for PIC16F87X and PIC18FXXX devices, respectively. These ICDs work with MPLAB IDE.

Identifier

A function or variable name.

IEEE

Institute of Electrical and Electronics Engineers.

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 (CPU) 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.

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.

Interrupt Handler

A routine that processes special code when an interrupt occurs.

Interrupt Service Routine

A function that is invoked when an interrupt occurs.

© 2008 Microchip Technology Inc.

Interrupt Vector

Address of an interrupt service routine or interrupt handler.

IRQ

See Interrupt Request.

ISO

See International Organization for Standardization.

ISR

See Interrupt Service Routine.

Librarian

See Archiver.

Library

See Archive.

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 Endianess

A data ordering scheme for multibyte data whereby the least significant byte 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.

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

Macroinstruction. 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.

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also uC.

Message

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

Microcontroller (MCU)

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

Mnemonics

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

MPASM Assembler

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

MPLAB ASM30/LINK30/LIB30

Previous names for Microchip's relocatable macro assembler, object linker and object archiver/librarian supporting 16-bit devices.

MPLAB C17/C18/C30

Previous names for various C compilers from Microchip. MPLAB C17 supports PIC17CXXX devices, MPLAB C18 supports PIC18CXXX and PIC18FXXXX devices, and MPLAB C30 supports dsPIC30F/33F DSCs and PIC24H/F MCUs.

MPLAB *Language Tool* for *Device*

Microchip's C compilers, assemblers and linkers for specifed 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 2

Microchip's in-circuit debugger for PIC16F87X, PIC18FXXX and dsPIC30FXXXX devices. The ICD works with MPLAB IDE. The main component of each ICD is the module. A complete system consists of a module, header, demo board, cables and MPLAB IDE Software.

MPLAB SIM30

Microchip's simulator that works with MPLAB IDE in support of 16-bit devices.

MPLIB Object Librarian

MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler (mpasm or mpasmwin v2.0) or MPLAB C Compiler for PIC18 MCUs .

MPLINK Object Linker

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip MPLAB C Compiler for PIC18 MCUs. 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.

Object Code

The machine code generated by an assembler or compiler.

© 2008 Microchip Technology Inc.

Object File

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).

Octal

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

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.

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.

Precedence

Rules that define the order of evaluation in expressions.

Program Counter

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

Program Counter Unit

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

The memory area in a device where instructions are stored.

PWM Signals

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

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.

Relaxation

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. The 16-bit assembler 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 file whose sections have not been assigned to a fixed location in memory.

Relocatable Section

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.

Section

A named sequence of code or data.

Section Attribute

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

SFR

See Special Function Registers.

Simulator

A software program that models the operation of devices.

Source Code

The form in which a computer program is written by the programmer. Source code is written in some 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

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

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 typically managed by the compiler when developing code in a high-level language.

Static RAM or SRAM

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

© 2008 Microchip Technology Inc.

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.

Storage Class

Determines the lifetime of an object.

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.

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.

Warning

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

NOTES:

© 2008 Microchip Technology Inc.

Index

Symbols

__C30_UART 165
__reset 146
_DMA_BASE 166
_main 146
_PROGRAM_END 166
. 47
.abort 74
.align 65, 140
.appline 74
.ascii 58
.asciz 59
.bss 50, 62
.bss section 92, 121, 145
.byte 59
.comm 62
.comm symbol, length 62
.const section 140, 146, 147
.data 50
.data section 93, 145
.def 76
.dim 76
.dinit section 145, 147
.double 59
.eject 69
.else 70
.elseif 70
.end 74
.endef 76
.endif 70
.endm 72
.endr 71, 72, 73
.equ 45, 63
.equiv 45, 63
.err 70
.error 70
.exitm 71
.extern 62
.fail 74
.file 76
.fill 66
.fillupper 56
.fillvalue 57
.fixed 60
.float 60
.global 62
.globl 62
.handle 94
.handle section 136, 144
.hword 61
.icd section 110
.ident 74
.if 70
.ifdecl 70
.ifdef 71
.ifndecl 70
.ifndef 71
.ifnotdecl 70
.ifnotdef 71
.incbin 74
.include 30, 31, 75
.int 61
.irp 71
.irpc 72
.lcomm 62
.lib* section 107
.libc section 107
.libdsp section 107
.libm section 107
.libperi section 107
.line 76
.list 69
.ln 74
.loc 75
.long 61
.macro 72
.memory 51
.nolist 69
.org 67
.palign 66
.pascii 58, 59
.pasciz 59
.pbss section 94
.pbyte 59, 138, 147
.pfill 66
.pfillvalue 57
.pincbin 75
.popsection 51
.porg 67
.print 75
.psize 69
.pskip 68
.pspace 68
.pstring 61
.purgem 73
.pushsection 51
.pword 61, 147
.rept 73
.reset section 106
.sbttl 69
.scl 76
.section name 51, 237
.set 45, 47, 63

.short 61
.single 60
.size 76, 77
.sizeof. 44
.skip 67
.sleb128 77
.space 67
.startof. 44
.string 61
.struct 68
.tag 77
.text 56
.text section 93, 107, 145
.title 69
.type 77
.uleb128 77
.val 77
.version 75
.weak 62, 143
.word 61
-(-) 89
$ 47

A

-A 89
-a 18
a.out 14, 29, 91
-a=file 26
-ac 19
Accessing Data 41
Accumulator Select 34
-ad 21
ADDR 131
-ah 22
-ai 23
Aivt Region 104
-al 23
ALIGN 132
alignment gaps 161
Allocatable Section 117
Allocating Memory 136
Allocating Unmapped Sections 141
-am 23
-an 25
ar utility 179
--architecture 89
Archiver 179
  Command-Line Interface 181
  Scripts 183
Arguments 34
-as 26
ASCII Character Set 253
Assembler
  Command-Line Interface 17
  Directives 33, 49
  Expression Syntax and Operation 39
  Overview 11
  Source 15
  Symbols 45
  Syntax 31
ASSERT 127
Assigning Output Sections to Regions 140
Assigning Values 115
Attributes
  Modify Section Types 52
  Represent Section Types 52
  Reserved Section Names 54
auto_psv 170, 171, 172, 174

B

Base Memory Addresses 105
bin2hex utility 189
Binary File 84
BLOCK 132
--boot 99
Build Options
  Linker 81
Building the Output File 137
Built-in Functions 131
  ADDR 131
  ALIGN 132
  BLOCK 132
  DEFINED 132
  LOADADDR 132
  MAX 133
  MIN 133
  NEXT 133
  SIZEOF 133

C

C30_UART 165
Character Constants 36
Characters 36
--check-sections 95
Command-Line Information
  Linker Scripts 102
Command-Line Interface
  Archiver/Librarian 181
  Assembler 17
  Linker 87
  Simulator 209
Comments 34, 113
Computing Absolute Addresses 137
Condition Codes 33
Conditional Assembly Directives
  .else 70
  .elseif 70
  .endif 70
  .err 70
  .error 70
  .if 70
  .ifdecl 70
  .ifdef 71
  .ifndecl 70
  .ifndef 71
  .ifnotdecl 70
  .ifnotdef 71
Configuration Region 104
Constant Data 42
Constants 128
  Fixed-Point Numbers 36
  Floating-Point Numbers 35
  Integer 35
  Locating in Program Memory 171

© 2008 Microchip Technology Inc.

Numeric 35
COPY 124
Creating Special Sections 136
--cref 98
crt0 ..... 85, 146, 149
crt1 146
Current Address 47
Custom Linker Script 113
Customer Notification Service 6
Customer Support 7

D

-d 89
Data Initialization Template 145
Data Memory 41, 137
Data Region 103
--data-init 94
-dc 89
Debug Information Directives
.def 76
.dim 76
.endef 76
.file 76
.line 76
.scl 76
.size 76, 77
.sleb128 77
.tag 77
.type 77
.uleb128 77
.val 77
Declare Symbols Directives
.bss 62
.comm 62
.extern 62
.global 62
.globl 62
.lcomm 62
.weak 62
Define Symbols Directives
.equ 63
.equiv 63
.set 63
DEFINED 132
--defsym 30, 89
Destination Select 34
Directive 33
Directives
Alignment 64
Assembler 49
Conditional 70
Debug Information 76
Declare Symbols 62
Define Symbols 63
Fill 56
Initialization 58
Miscellaneous 74
Output Listing 69
Section 50
Substitution/Expansion 71
--discard-all 89
--discard-locals 89
DMA_BASE 166
Documentation
Conventions 4
Layout 1
DOT Symbol 47
Dot Variable 129, 149
-dp 89
DSECT 124

E

EEDATA Memory Region 105
EEPROM, Locating and Accessing 172
Empty Expressions 39
--end-group 89
ENTRY 127
Entry Point 102
Escape Characters 36
Evaluation 130
Examples, Linker 167
EXCLUDE_FILE 119
Executable Section 117
Expression Syntax and Operation
Assembler 39
Expressions 39
Expressions, Empty 39
Expressions, Integer 39
EXTERN 127

F

--fatal-warnings 28
File Commands, Linker Scripts
GROUP 114
INCLUDE 114
INPUT 114
OUTPUT 115
SEARCH_DIR 115
STARTUP 115
File Extensions
Assembler 13, 83
Linker 83
File Registers 33
Files
Library 83
Linker Output 84
Linker Script 83
Listing 15
Map 84
Object 14, 83
Source 13
Fill Directives
.fillupper 56
.fillvalue 57
.pfillvalue 57
--fill-upper 89
Fixed-Point Numbers 36
Floating-Point Numbers 35
FORCE_COMMON_ALLOCATION 127
--force-exe-suffix 90
--force-link 90
Functions, Locating 169

G

gaps, alignment 161
--general 99
Getting a Grip 143
Global Symbols 143
GROUP 114

H

handle() 43, 136, 137, 144
Handles 143
--handles 94
Header 15
--heap 94
Heap Allocation 150
--help 28, 95
Hexadecimal to Decimal Conversion 254
High-level Source 15, 22

I

-I 30
-i 91
ICD Memory 110
INCLUDE 114
Infix Operators 40
INFO 124
Informational Output Options, Assembler
- --fatal-warnings 28
- --help 28
- -J 28
- --no-warn 28
- --target-help 28
- -v 29
- --verbose 29
- --version 29
- -W 28
- --warn 28

Informational Output Options, Linker
- --check-sections 95
- --help 95
- --no-check-sections 95
- --no-warn-mismatch 96
- --report-mem 96
- -t 96
- --trace 96
- --trace-symbol 96
- -V 96
- -v 96
- --verbose 96
- --version 96
- --warn-common 97
- --warn-once 97
- --warn-section-align 98
- -y 96

Initializated Section 117
Initialization Directives
- .ascii 58
- .asciz 59
- .byte 59
- .double 59
- .fixed 60
- .float 60
- .hword 61
- .int 61
- .long 61
- .pascii 58, 59
- .pasciz 59
- .pbyte 59
- .pstring 61
- .pword 61
- .short 61
- .single 60
- .string 61
- .word 61

Initialized Data 144
Initialized Section 117
INPUT 114
Input Section
- Common Symbols 121
- Example 121
- Wildcard Patterns 120

Input/Output Section Map 105
Integer Expressions 39
Integers 35
Internal Preprocessor 31
Internet Address, Microchip 6
Interrupt
- Handlers 150
- Request 152
- Vector Tables 111, 150

Invert Sense 117
IRQ 152
--isr 90
Ivt Region 104

J

-J 28

K

K Suffix 128
--keep-locals 29

L

-L 29, 91
-l 90
Label 32, 46, 138, 144
LENGTH 117
Librarian 179
- Command-Line Interface 181
- Scripts 183

--library 90
Library Files 83
--library-path 91
license manager utility 205
Link Map Options, Linker
- --cref 98
- -M 98
- -Map 98
- --print-map 98

Linker
- Allocation 139
- Command-Line Interface 87
- Examples 167

© 2008 Microchip Technology Inc.

File Extensions 83
Output File 84
Overview 81
Processing 135
Linker Scripts 101
Command Language 113
Command-Line Information 102
Concepts 113
Contents 102
Custom 113
Expressions 128
File 83
File Commands 114
Other Commands 127
Listing Files 15
Listing Output Options, Assembler 18
-a=file 26
-ac 19
-ad 21
-ah 22
-ai 23
-al 23
-am 23
-an 25, 26
--listing-cont-lines 28
--listing-lhs-width 27
--listing-lhs-width2 27
--listing-rhs-width 27
--listing-cont-lines 28
--listing-lhs-width 27
--listing-lhs-width2 27
--listing-rhs-width 27
Literals 33
LMA 114, 124, 132
Load Memory Address 114, 124, 132
LOADADDR 132
Loading Input Files 135
Local Symbols 46
Location Counter 108, 129
Location Counter Directives
.align 65
.fill 66
.org 67
.palign 66
.pfill 66
.porg 67
.pskip 68
.pspace 68
.skip 67
.space 67
.struct 68

M

-M 98
M Suffix 128
-Map 98
Map File 84
Mapping Sections 139
MAX 133
-MD 30
Memory Addressing 137
MEMORY Command 116
! 117
A 117
I 117
L 117
R 117
W 117
X 117
Memory Region Information 103
MIN 133
Miscellaneous Directives
.abort 74
.appline 74
.end 74
.fail 74
.incbin 74
.include 75
.indent 74
.ln 74
.loc 75
.pincbin 75
.print 75
.version 75
Mnemonic 32
Modification Options, Archiver/Librarian
a 182
b 182
c 182
f 182
i 182
l 182
N 182
o 182
P 182
S 182
s 182
u 182
V 182
v 182
MPLAB ASM30 11
MPLAB ICD 2 Debugger Memory 110
MPLAB LIB30 179
MPLAB LINK30 81

N

NEXT 133
nm utility 191
--no-check-sections 95
NOCROSSREFS 127
--no-data-init 94
--no-force-link 90
--no-handles 94
--noinhibit-exec 91
--no-isr 90
--no-keep-memory 91
NOLOAD 124
--no-pack-data 95
--no-relax 29
--no-undefined 93
--no-warn 28
--no-warn-mismatch 96

Numeric Constants 35

O

-o 29, 91
objdump utility 195
Object Files 14, 83
-omf 14, 29, 91, 190, 192, 197, 199, 202, 204
Operands 33
Operators 40, 130
    Infix 40
    Prefix 40
Optimize 91
Options, Archiver/Librarian
    d 181
    m 181
    p 181
    q 181
    r 181
    t 181
    x 182
Options, Assembler
    Informational Output 28
    Listing Output 18
    Other 30
    Output File Creation 29
Options, Linker
    Informational Output 95
    Link Map Output 98
    Output File Creation 88
    Run-time Initialization 94
Options, pic30-bin2hex
    -omf 190
    -v 190
Options, pic30-lm
    -? 205
    -u 205
Options, pic30-nm
    -A 192
    -a 192
    -B 192
    --debug-syms 192
    --defined-only 192
    --extern-only 192
    -f 192
    --format= 192
    -g 192
    --help 192
    -l 192
    --line-numbers 192
    -n 192
    --no-sort 192
    --numeric-sort 192
    -o 192
    -omf 192
    -P 192
    -p 192
    --portability 192
    --print-armap 192
    --print-file-name 192
    -r 192
    --radix= 192
    --reverse-sort 192
    -s 192
    --size-sort 192
    -t 192
    -u 192
    --undefined-only 192
    -V 192
    -v 192
    --version 192
Options, pic30-objdump
    -a 196
    --all-header 197
    --archive-header 196
    -D 196
    -d 196
    --debugging 196
    --disassemble 196
    --disassemble-all 196
    --disassembler-options= 196
    --disassemble-zeroes 197
    -EB 196
    -EL 196
    --endian= 196
    -f 196
    --file-header 196
    --file-start-context 196
    --full-contents 197
    -g 196
    -H 196
    -h 196
    --header 196
    --help 196
    -j 196
    -l 196
    --line-numbers 196
    -M 196
    --no-show-raw-insn 197
    -omf 197
    --prefix-addresses 197
    -r 197
    --reloc 197
    -S 197
    -s 197
    --section= 196
    --section-header 196
    --show-raw-insn 197
    --source 197
    --start-address= 197
    --stop-address= 197
    --syms 197
    -t 197
    -V 197
    --version 197
    -w 197
    --wide 197
    -x 197
    -z 197
Options, pic30-ranlib
    -omf 199
    -V 199

© 2008 Microchip Technology Inc.

-v ... 199
--version ... 199
Options, pic30-strings
-... 202
-a ... 202
--all ... 202
--bytes= ... 202
-f ... 202
--help ... 202
-n ... 202
-omf ... 202
--print-file-name ... 202
--radix= ... 202
-t ... 202
-v ... 202
--version ... 202
Options, pic30-strip
--discard-all ... 204
--discard-locals ... 204
-g ... 204
--help ... 204
-K ... 204
--keep-symbol= ... 204
-N ... 204
-o ... 204
-omf ... 204
-p ... 204
--preserve-dates ... 204
-R ... 204
--remove-section= ... 204
-S ... 204
-s ... 204
--strip-all ... 204
--strip-debug ... 204
--strip-symbol= ... 204
--strip-unneeded ... 204
-V ... 204
-v ... 204
--verbose ... 204
--version ... 204
-X ... 204
-x ... 204
Options, Simulator
AF ... 210
BC ... 210
BS ... 210
DA ... 210
DB ... 210
DC ... 210
DF ... 210
DH ... 210
DM ... 210
DP ... 210
DS ... 210
DW ... 210
E ... 210
FC ... 210
FS ... 210
H ... 210
HE ... 210
HW ... 210
IF ... 211
IO ... 211
LC ... 211
LD ... 211
LF ... 211
LP ... 211
LS ... 211
MC ... 211
MS ... 211
PS ... 211
Q ... 211
RC ... 211
RP ... 211
S ... 212
VF ... 212
VO ... 212
ORG ... 117
ORIGIN ... 117
Other Linker Script Commands
ASSERT ... 127
ENTRY ... 127
EXTERN ... 127
FORCE_COMMON_ALLOCATION ... 127
NOCROSSREFS ... 127
OUTPUT_ARCH ... 127
OUTPUT_FORMAT ... 127
TARGET ... 127
Other Options, Assembler
--defsym ... 30
-I ... 30
-p ... 30
--processor ... 30
OUTPUT ... 115
--output ... 91
Output File Creation Options, Assembler
--keep-locals ... 29
-L ... 29
-MD ... 30
--no-relax ... 29
-o ... 29
-omf ... 29
-R ... 29
--relax ... 29
-Z ... 30
Output File Format ... 102
Output File Options, Linker
-(-) ... 89
-A ... 89
--architecture ... 89
-d ... 89
-dc ... 89
--defsym ... 89
--discard-all ... 89
--discard-locals ... 89
-dp ... 89
--end-group ... 89
--fill-upper ... 89
--force-exe-suffix ... 90
--force-link ... 90

-i ... 91
--isr ... 90
-L ... 91
-l ... 90
--library ... 90
--library-path ... 91
--no-force-link ... 90
--noinhibit-exec ... 91
--no-isr ... 90
--no-keep-memory ... 91
--no-undefined ... 93
-o ... 91
-omf ... 91
--output ... 91
--p ... 91
--processor ... 91
-r ... 91
--relocateable ... 91
--retain-symbols-file ... 92
-S ... 92
-s ... 92
--script ... 92
--start-group ... 89
--strip-all ... 92
--strip-debug ... 92
-T ... 92
-Tbss ... 92
-Tdata ... 93
-Ttext ... 93
-u ... 93
--undefined ... 93
-Ur ... 91
--wrap ... 93
-X ... 89
-x ... 89
Output Formats, pic30-nm
? ... 193
A ... 193
B ... 193
C ... 193
D ... 193
N ... 193
R ... 193
T ... 193
U ... 193
V ... 193
W ... 193
Output Listing Directives
.eject ... 69
.list ... 69
.nolist ... 69
.psize ... 69
.sbttl ... 69
.title ... 69
Output Section
.reset ... 106
.text ... 107
Address ... 122
Attributes ... 123
Data ... 122
Description ... 121
Discarding ... 123
Fill ... 125
LMA ... 124
Region ... 125
Type ... 124
COPY ... 124
DSECT ... 124
INFO ... 124
NOLOAD ... 124
OVERLAY ... 124
Output Sections in
Configuration Memory ... 108
OUTPUT_ARCH ... 127
OUTPUT_FORMAT ... 127
OVERLAY ... 124
Overlay Description ... 125
Overview
Assembler ... 11
Linker ... 81

P

--p ... 91
-p ... 30
--pack-data ... 95
paddr() ... 43
Page Size ... 42
pic30 ... 196, 204
pic30-ar utility ... 179
pic30-bin2hex utility ... 189
pic30-lm utility ... 205
pic30-nm utility ... 191
pic30-objdump utility ... 195
pic30-ranlib utility ... 199
pic30-strings utility ... 201
pic30-strip utility ... 203
Pointer ... 42
Precedence ... 40, 130
Prefix Operators ... 40
Preprocessor, Internal ... 31
--print-map ... 98
Process Flow
Assembler ... 12
Librarian ... 180
Linker ... 82
Processing, Linker ... 135
--processor ... 30, 91
Program Address ... 43
Program Memory ... 41, 137
Program Memory, Locating and Reserving ... 175
Program Region ... 104
Program Space
Visibility Window ... 42, 125, 138, 140, 146
PROGRAM_END ... 166
PROVIDE ... 116
PSV Window ... 42, 125, 138, 140, 146, 237
psvoffset() ... 42
PSVPAG Register, Saving and Restoring ... 170
psvpage() ... 42

© 2008 Microchip Technology Inc.

R

-R 29
-r 91
ranlib utility 199
Read/Write Section 117
Reading, Recommended 5
Read-Only Data 147
Read-Only Section 117
Registers 33
Relative Branches 29
Relative Calls 29
--relax 29
relocatable 14
Relocatable Code 168
--relocateable 91
--report-mem 96
Reserved Names 45
RESET 150
Reset Region 104
Resolving Symbols 136
--retain-symbols-file 92
Run-time Initialization Options, Linker
    --data-init 94
    --handles 94
    --heap 94
    --no-data-init 94
    --no-handles 94
    --no-pack-data 95
    --pack-data 95
    --stack 95
Run-time Library Support 146

S

-S 92
-s 92
--script 92
Scripts
    Librarian 183
Scripts, Archiver/Librarian
    ADDLIB 183
    ADDMOD 183
    CLEAR 183
    CREATE 183, 184
    DELETE 184
    DIRECTORY 184
    END 184
    EXTRACT 184
    LIST 184
    OPEN 183, 184
    REPLACE 184
    SAVE 183, 184
    VERBOSE 184
SEARCH_DIR 115
Section Directives
    .bss 50
    .data 50
    .memory 51
    .popsection 51
    .pushsection 51
    .section name 51, 237
    .text 56
Section of an Expression 130
SECTIONS Command 118
--secure 99
Security Feature Options, Linker
    --boot 99
    --general 99
    --secure 99
SFR Addresses 112
SFRs 103, 112, 135
sim30 ..... 102, 209
Simple Assignments 115
Simulator Command-Line Interface 209
SIZEOF 133
Source Code 32
Source Files 13
Special Function Registers 103, 112, 135
Special Operators 41
    .sizeof 41
    .startof 41
    dmaoffset 41
    handle 41
    paddr 41
    psvoffset 41
    psvpage 41
    tbloffset 41
    tblpage 41
SPLIM 146, 149
--stack 95
Stack Allocation 149
Stack Pointer 146, 149
Stack Pointer Limit Register 146, 149
Stack, Locating 175
Standard Data Section Names 145
--start-group 89
Starting Address 44
STARTUP 115
Start-up Code 145
Start-up Module 146
Statement Format 32
Strings 36
strings utility 201
strip utility 203
--strip-all 92
--strip-debug 92

Substitution/Expansion Directives
.endm ... 72
.endr ... 71, 72, 73
.exitm ... 71
.irpc ... 72
.macro ... 72
.purgem ... 73
.rept ... 73
irp ... 71
Subtitle ... 15, 69
supported ... 210
Symbol Names ... 128
Symbol Table ... 15, 26, 76, 77, 114, 132
Symbols ... 45
Assembler ... 45
Syntax
Archiver/Librarian ... 181
Assembler ... 17, 31
Linker ... 87
pic30-bin2hex ... 190
pic30-nm ... 191
pic30-objdump ... 195
pic30-ranlib ... 199
pic30-strings ... 201
pic30-strip ... 203
Simulator ... 209

T

-T ... 92
-t ... 96
Table Access Instructions ... 138
TARGET ... 127
--target-help ... 28
tbloffset() ... 42, 138
tblpage() ... 42, 138
-Tbss ... 92
-Tdata ... 93
Title ... 69
Title Line ... 15
--trace ... 96
--trace-symbol ... 96
-Ttext ... 93

U

-u ... 93
--undefined ... 93
-Ur ... 91
User-Defined Section in Data Memory ... 110
User-Defined Section in Program Memory ... 108
Utilities ... 185

V

-V ... 96
-v ... 29, 96
Variables, Locating ... 169
--verbose ... 29, 96
--version ... 29, 96
Virtual Memory Address ... 114, 124
VMA ... 114, 124

W

-W ... 28
W15 ... 146, 149
--warn ... 28
--warn-common ... 97
--warn-once ... 97
--warn-section-align ... 98
Watchdog Timer, Disabling ... 109
Weak Symbols ... 143
Web Site, Microchip ... 6
White Space ... 32
--wrap ... 93

X

-X ... 89
-x ... 89
X Memory, Creating a Modulo Buffer ... 174

Y

-y ... 96
Y Memory, Creating a Modulo Buffer ... 174

Z

-Z ... 30

© 2008 Microchip Technology Inc.

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://support.microchip.com
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

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

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

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

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: 86-10-8528-2100
Fax: 86-10-8528-2104

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

China - Hong Kong SAR
Tel: 852-2401-1200
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: 86-755-8203-2660
Fax: 86-755-8203-1760

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

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

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

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

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

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 - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

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: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
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

01/02/08

© 2008 Microchip Technology Inc.