



**MPLAB[®] ASM30,
MPLAB[®] LINK30
AND UTILITIES
USER'S GUIDE**

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KEELOQ, MPLAB, PIC, PICmicro, PICSTART, PRO MATE and PowerSmart are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.


Amplab, FilterLab, microID, MXDEV, MXLAB, PICMASTER, SEEVAL, SmartShunt and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Application Maestro, dsPICDEM, dsPICDEM.net, dsPICworks, ECAN, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, PICKit, PICDEM, PICDEM.net, PICtail, PowerCal, PowerInfo, PowerMate, PowerTool, rLAB, rPIC, Select Mode, SmartSensor, SmartTel and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

Serialized Quick Turn Programming (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.

© 2003, 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 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona and Mountain View, California in October 2003. The Company's quality system processes and procedures are for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, non-volatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Table of Contents

Preface	1
Part 1 – MPLAB ASM30 Assembler	
<hr/>	
Chapter 1. Assembler Overview	
1.1 Introduction	9
1.2 Highlights	9
1.3 MPLAB ASM30 and Other Development Tools	9
1.4 Feature Set	10
1.5 Input/Output Files	10
Chapter 2. MPLAB ASM30 Command Line Interface	
2.1 Introduction	15
2.2 Highlights	15
2.3 Syntax	15
2.4 Options that Modify the Listing Output	16
2.5 Options that Control Informational Output	26
2.6 Options that Control Output File Creation	27
2.7 Other Options	28
Chapter 3. Assembler Syntax	
3.1 Introduction	29
3.2 Highlights	29
3.3 Internal Preprocessor	29
3.4 Source Code Format	30
3.5 Constants	32
3.6 Summary	34
Chapter 4. Assembler Expression Syntax and Operation	
4.1 Introduction	35
4.2 Highlights	35
4.3 Expressions	35
4.4 Operators	36
4.5 Special Operators	37
Chapter 5. Assembler Symbols	
5.1 Introduction	41
5.2 Highlights	41
5.3 What are Symbols	41
5.4 Reserved Names	41
5.5 Local Symbols	42

5.6 Giving Symbols Other Values	43
5.7 The Special DOT Symbol	43
5.8 Using Executable Symbols in a Data Context	43

Chapter 6. Assembler Directives

6.1 Introduction	45
6.2 Highlights	45
6.3 Directives that Define Sections	46
6.4 Assembler Directives that Modify the way Program Memory is Filled	48
6.5 Assembler Directives that Initialize Constants	49
6.6 Assembler Directives that Declare Symbols	52
6.7 Assembler Directives that Define Symbols	53
6.8 Assembler Directives that Modify the Section Location Counter	54
6.9 Assembler Directives that Format the Output Listing	57
6.10 Conditional Assembler Directives	58
6.11 Substitution/Expansion Assembler Directives	59
6.12 Miscellaneous Assembler Directives	61
6.13 Assembler Directives for Debug Information	62

Part 2 – MPLAB LINK30 Linker

Chapter 7. Linker Overview

7.1 Introduction	67
7.2 Highlights	67
7.3 MPLAB LINK30 and Other Development Tools	67
7.4 Feature Set	68
7.5 Input/Output Files	68

Chapter 8. MPLAB LINK30 Command Line Interface

8.1 Introduction	73
8.2 Highlights	73
8.3 Syntax	73
8.4 Options that Control Output File Creation	74
8.5 Options that Control Runtime Initialization	78
8.6 Options that Control Informational Output	79
8.7 Options that Modify the Link Map Output	82

Chapter 9. Linker Scripts

9.1 Introduction	83
9.2 Highlights	83
9.3 Overview of Linker Scripts	83
9.4 Command Line Information	84
9.5 Contents of a Linker Script	84
9.6 Creating a Custom Linker Script	98
9.7 Linker Script Command Language	98
9.8 Expressions in Linker Scripts	112

Chapter 10. Linker Processing

10.1 Introduction	119
10.2 Highlights	119
10.3 Overview of Linker Processing	119
10.4 Memory Addressing	121
10.5 Linker Allocation	122
10.6 Global and Weak Symbols	124
10.7 Handles	125
10.8 Initialized Data	126
10.9 Read-only Data	129
10.10 Stack Allocation	131
10.11 Heap Allocation	132
10.12 Interrupt Vector Tables	132

Part 3 – MPLAB LIB30 Archiver/Librarian

Chapter 11. MPLAB LIB30 Archiver/Librarian

11.1 Introduction	139
11.2 Highlights	139
11.3 MPLAB LIB30 and Other Development Tools	140
11.4 Feature Set	140
11.5 Input/Output Files	140
11.6 Syntax	141
11.7 Options	141
11.8 Scripts	143

Part 4 – Utilities

Chapter 12. Utilities Overview

12.1 Introduction	147
12.2 Highlights	147
12.3 What are Utilities	147

Chapter 13. pic30-bin2hex Utility

13.1 Introduction	149
13.2 Highlights	149
13.3 Input/Output Files	149
13.4 Syntax	149
13.5 Options	149

Chapter 14. pic30-nm Utility

14.1 Introduction	151
14.2 Highlights	151
14.3 Input/Output Files	151
14.4 Syntax	151
14.5 Options	152
14.6 Output Formats	153

Chapter 15. pic30-objdump Utility

15.1 Introduction	155
15.2 Highlights	155
15.3 Input/Output Files	155
15.4 Syntax	155
15.5 Options	156

Chapter 16. pic30-ranlib Utility

16.1 Introduction	159
16.2 Highlights	159
16.3 Input/Output Files	159
16.4 Syntax	159
16.5 Options	159

Chapter 17. pic30-strings Utility

17.1 Introduction	161
17.2 Highlights	161
17.3 Input/Output Files	161
17.4 Syntax	161
17.5 Options	162

Chapter 18. pic30-strip Utility

18.1 Introduction	163
18.2 Highlights	163
18.3 Input/Output Files	163
18.4 Syntax	163
18.5 Options	164

Chapter 19. pic30-lm Utility

19.1 Introduction	165
19.2 Highlights	165
19.3 Syntax	165
19.4 Options	165

Part 5 – dsPIC30F Simulator

Chapter 20. Command Line Simulator

20.1 Introduction	169
20.2 Highlights	169
20.3 Syntax	169
20.4 Options	170

Part 6 – Appendices

Appendix A. Assembler Errors/Warnings/Messages

A.1 Introduction	175
A.2 Highlights	175
A.3 Fatal Errors	175
A.4 Errors	176

Table of Contents

A.5 Warnings	183
A.6 Messages	187
Appendix B. Linker Errors/Warnings	
B.1 Introduction	189
B.2 Highlights	189
B.3 Errors	189
B.4 Warnings	194
Appendix C. MPASM™ Assembler Compatibility	
C.1 Introduction	197
C.2 Highlights	197
C.3 Compatibility	197
C.4 Examples	200
C.5 Converting PIC18FXXX Assembly Code to dsPIC30FXXXX Assembly Code	201
Appendix D. MPLINK™ Linker Compatibility	
D.1 Introduction	207
D.2 Highlights	207
D.3 Compatibility	207
D.4 Migration to MPLAB LINK30	207
Appendix E. MPLIB™ Librarian Compatibility	
E.1 Introduction	209
E.2 Highlights	209
E.3 Compatibility	209
E.4 Examples	209
Appendix F. Useful Tables	
F.1 ASCII Character Set	211
F.2 Hexadecimal to Decimal Conversion	212
Appendix G. GNU Free Documentation License	213
Glossary	219
Index	231
Worldwide Sales and Service	241

NOTES:

Preface

INTRODUCTION

The purpose of this document is to help you use Microchip Technology's language tools for dsPIC® devices based on GNU technology. The language tools discussed are:

- MPLAB ASM30 Assembler
- MPLAB LINK30 Linker
- MPLAB® LIB30 Archiver/Librarian
- Other Utilities

HIGHLIGHTS

Topics covered in this chapter are:

- About this Guide
- Recommended Reading
- Troubleshooting
- The Microchip Web Site
- Development Systems Customer Notification Service
- Customer Support

ABOUT THIS GUIDE

Document Layout

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

Part 1 - MPLAB ASM30 Assembler

- **Chapter 1: Assembler Overview** – gives an overview of assembler operation.
- **Chapter 2: MPLAB ASM30 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 LINK30 Linker

- **Chapter 7: Linker Overview** – gives an overview of linker operation.
- **Chapter 8: MPLAB LINK30 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.

Part 3 - MPLAB LIB30 Archiver/Librarian

- **Chapter 11: MPLAB LIB30 Archiver/Librarian** – details command line options for the librarian.

Part 4 - Utilities

- **Chapter 12: Utilities Overview** – gives an overview of utilities and their operation.
- **Chapter 13: pic30-bin2hex Utility** – details command line options for binary-to-hexadecimal conversion.
- **Chapter 14: pic30-nm Utility** – details command line options for listing symbols in an object file.
- **Chapter 15: pic30-objdump Utility** – details command line options for displaying information about object files.
- **Chapter 16: pic30-ranlib Utility** – details command line options for creating an archive index.
- **Chapter 17: pic30-strings Utility** – details command line options for printing character sequences.
- **Chapter 18: pic30-strip Utility** – details command line options for discarding all symbols from an object file.
- **Chapter 19: pic30-lm Utility** – details command line options for displaying information about the MPLAB C30 license.

Part 5 - dsPIC30F Simulator

- **Chapter 20: Command Line Simulator** – describes the command line simulator that supports dsPIC® DSC tools.

Part 6 - Appendices

- **Appendix A: Assembler Errors/Warnings/Messages** – contains a descriptive list of the errors, warnings and messages generated by MPLAB ASM30.
- **Appendix B: Linker Errors/Warnings** – contains a descriptive list of the errors and warnings generated by MPLAB LINK30.
- **Appendix C: MPASM™ Assembler Compatibility** – contains information on compatibility with MPASM assembler, examples and recommendations for migration to MPLAB ASM30.
- **Appendix D: MPLINK™ Linker Compatibility** – contains information on compatibility with MPLINK linker, examples and recommendations for migration to MPLAB LINK30.

- **Appendix E: MPLIB™ Librarian Compatibility** – contains information on compatibility with MPLIB librarian, examples and recommendations for migration to MPLAB LIB30.
- **Appendix F: Useful Tables** – lists some useful tables: the ASCII character set and hexadecimal to decimal conversion.
- **Appendix G: GNU Free Documentation License** – details the license requirements for using the GNU language tools.

Conventions Used in this Guide

This manual uses the following documentation conventions:

TABLE 1: DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Main Document (Arial font):		
Italic characters	Referenced books	<i>MPLAB IDE User's Guide</i>
	Emphasized text	...is the <i>only</i> compiler...
Interface References (Arial font):		
Initial caps	A window, dialog or menu selection	Configuration Bits window, Settings dialog, Enable Programmer
Quotes	A field name in a window or dialog	"Save files before running the debugger"
Underlined, italic text with right arrow	A menu selection path	<u>File>Save</u>
Bold characters	A dialog button or tab	OK button, Power tab
Characters in angle brackets < >	A key on the keyboard	<Tab>, <Ctrl-C>
Code References (Courier font):		
Plain characters	File names and paths	c:\autoexec.bat
	Bit values	0, 1
	Sample code	#define START
Square brackets []	Optional arguments	mpasmwin [main.asm]
Curly brackets and pipe character: { }	Choice of mutually exclusive arguments An OR selection	errorlevel {0 1}
Italic characters	A variable argument; it can be either a type of data (in lower case characters) or a specific example (in uppercase characters).	pic30-gcc <i>filename</i>
Ellipses...	Replaces repeated instances of text	list ["list_option...", "list_option"]
0xnnnn	A hexadecimal number where <i>n</i> is a hexadecimal digit	0xFFFF, 0x007A, 0x1A
'bnnnn	A binary number where <i>n</i> is a digit	'b00100, 'b10

Documentation Updates

All documentation becomes dated, and this document is no exception. Since Microchip language and other tools are constantly evolving to meet customer needs, some actual tool descriptions may differ from those in this document. Please refer to our web site to obtain the latest documentation available.

Documentation Numbering Conventions

Documents are numbered with a "DS" number. The 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 = The document number.
A = The revision level of the document.

RECOMMENDED READING

This document describes how to use MPLAB C30 compiler for dsPIC devices. For more information on MPLAB C30 and the use of other tools, the following are recommended reading:

README Files

For the latest information on Microchip tools, read the associated README files (ASCII text files) included with the software.

dsPIC® Language Tools Getting Started (DS70094)

A guide to installing and working with the Microchip language tools (MPLAB ASM30, MPLAB LINK30 and MPLAB C30) for dsPIC digital signal controllers (DSC's). Examples using the dsPIC simulator, and MPLAB SIM30, are provided.

MPLAB® C30 C Compiler User's Guide (DS51284)

A guide to using the dsPIC DSC C compiler. MPLAB LINK30 is used with this tool.

dsPIC® Language Tools Libraries (Preliminary)

DSP, dsPIC peripheral and standard (including math) libraries for use with dsPIC language tools.

GNU HTML Documentation

This documentation is provided on the language tool CD-ROM. It describes the standard GNU development tools, upon which these tools are based.

dsPIC30F Enhanced Flash 16-Bit Digital Signal Controllers General Purpose and Sensor Families Data Sheet (DS70083)

Data sheet for dsPIC30F digital signal controller (DSC). Gives an overview of the device and its architecture. Details memory organization, DSP operation and peripheral functionality. Includes electrical characteristics.

dsPIC30F Family Reference Manual (DS70046)

This manual explains the operation of the dsPIC30F MCU family architecture and peripheral modules.

dsPIC30F Programmer's Reference Manual (DS70030)

Programmer's guide to dsPIC30F devices. Includes the programmer's model and instruction set.

Microchip Web Site

Our web site (<http://www.microchip.com>) contains a wealth of documentation. Individual data sheets, application notes, tutorials and user's guides are all available for easy download. All documentation is in Adobe® Acrobat® (pdf) format.

TROUBLESHOOTING

See the README files for information on common problems not addressed in this document.

THE MICROCHIP WEB SITE

Microchip provides online support on the Microchip World Wide Web (WWW) site. The web site is used by Microchip as a means to make files and information easily available to customers. To view the site, you must have access to the Internet and a web browser, such as, Netscape Navigator® or Microsoft® Internet Explorer.

The Microchip web site is available by using your favorite Internet browser to reach:

<http://www.microchip.com>

The web site provides a variety of services. Users may download files for the latest development tools, data sheets, application notes, user's guides, articles and sample programs. A variety of information specific to the business of Microchip is also available, including listings of Microchip sales offices, distributors and factory representatives.

Technical Support

- Frequently Asked Questions (FAQ)
- Online Discussion Groups – conferences for products, development systems, technical information and more
- Microchip Consultant Program Member Listing
- Links to other useful web sites related to Microchip products

Engineer's Toolbox

- Design Tips
- Device Errata

Other Available Information

- Latest Microchip Press Releases
- Listing of seminars and events
- Job Postings

DEVELOPMENT SYSTEMS CUSTOMER NOTIFICATION SERVICE

Microchip started the customer notification service to help our customers keep current on Microchip products with the least amount of effort. Once you subscribe, you will receive e-mail notification whenever we change, update, revise or have errata related to your specified product family or development tool of interest.

Go to the Microchip web site at (<http://www.microchip.com>) and click on Customer Change Notification. Follow the instructions to register.

The Development Systems product group categories are:

- Compilers
- Emulators
- In-Circuit Debuggers
- MPLAB IDE
- Programmers

Here is a description of these categories:

Compilers – The latest information on Microchip C compilers and other language tools. These include the MPLAB® C17, MPLAB C18 and MPLAB C30 C compilers; MPASM™ and MPLAB ASM30 assemblers; MPLINK™ and MPLAB LINK30 object linkers; MPLIB™ and MPLAB LIB30 object librarians.

Emulators – The latest information on Microchip in-circuit emulators. This includes the MPLAB ICE 2000 and MPLAB ICE 4000.

In-Circuit Debuggers – The latest information on the Microchip in-circuit debugger, MPLAB ICD 2.

MPLAB IDE – The latest information on Microchip MPLAB® IDE, the Windows Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB SIM and MPLAB SIM30 simulators, MPLAB IDE Project Manager and general editing and debugging features.

Programmers – The latest information on Microchip device programmers. These include the MPLAB PM3 and PRO MATE® II device programmers and PICSTART® Plus development programmer.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Corporate Applications Engineer (CAE)
- Hotline

Customers should call their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. See the back cover for a list of sales offices and locations.

Corporate Applications Engineers (CAEs) may be contacted at (480) 792-7627.

In addition, there is a Systems Information and Upgrade Line. This line provides system users a list of the latest versions of all of Microchip's development systems software products. Plus, this line provides information on how customers can receive any currently available upgrade kits.

The Hotline Numbers are:

1-800-755-2345 for U.S. and most of Canada.

1-480-792-7302 for the rest of the world.



MPLAB® ASM30, MPLAB® LINK30 AND UTILITIES USER'S GUIDE

Part
1

MPLAB ASM30 Assembler

Part 1 – MPLAB ASM30 Assembler

Chapter 1. Assembler Overview	9
Chapter 2. MPLAB ASM30 Command Line Interface	15
Chapter 3. Assembler Syntax	29
Chapter 4. Assembler Expression Syntax and Operation	35
Chapter 5. Assembler Symbols	41
Chapter 6. Assembler Directives	45

NOTES:

Chapter 1. Assembler Overview

1.1 INTRODUCTION

MPLAB ASM30 produces relocatable machine code from symbolic assembly language for dsPIC 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

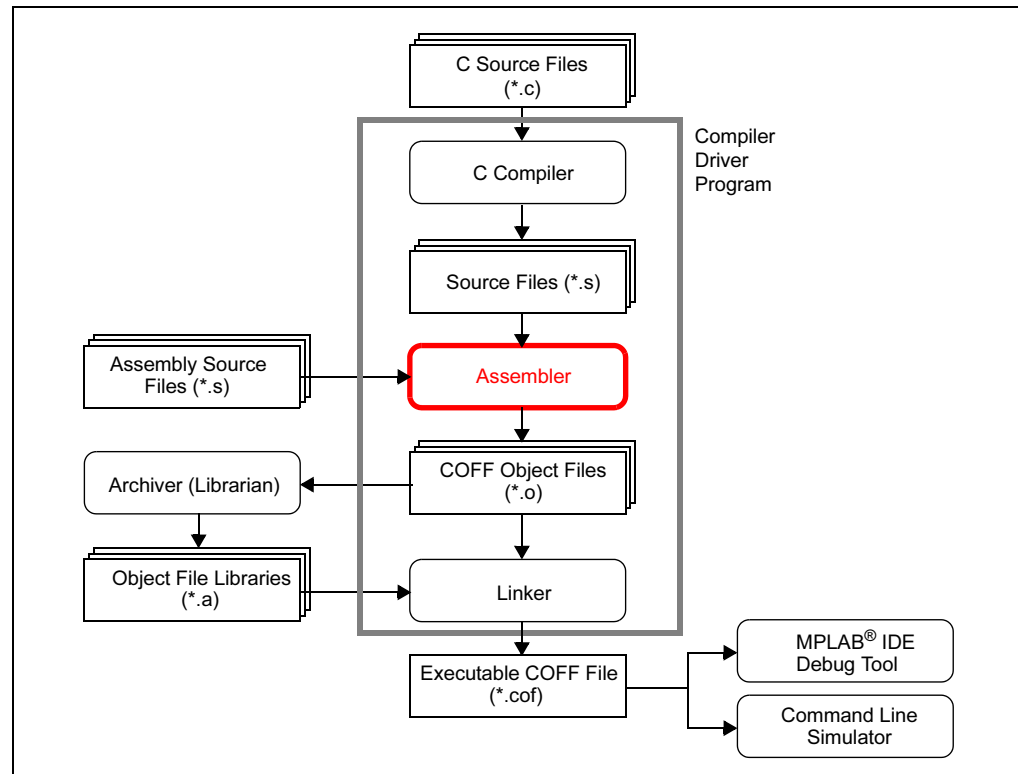
Topics covered in this chapter are:

- MPLAB ASM30 and Other Development Tools
- Feature Set
- Input/Output Files

1.3 MPLAB ASM30 AND OTHER DEVELOPMENT TOOLS

MPLAB ASM30 translates user assembly source files. In addition, the dsPIC C Compiler (MPLAB C30) 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 COFF file. See Figure 1-1 for an overview of the tools process flow.

FIGURE 1-1: TOOLS PROCESS FLOW



1.4 FEATURE SET

Notable features of the assembler include:

- Support for the entire dsPIC instruction set
- Support for fixed-point and floating-point data
- Available for Windows
- Command Line Interface
- Rich Directive Set
- Flexible Macro Language
- Integrated component of MPLAB IDE

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 PICmicro® MCU's), MPLAB ASM30 does not generate error files, HEX files, or symbol and debug files. MPLAB ASM30 is capable of creating a listing file and a relocatable COFF object file (that may or may not contain debugging information). MPLAB LINK30, the linker, is used with MPLAB ASM30 to produce the final object files, map files and final COFF file for debugging with MPLAB IDE (see Figure 1-1).

1.5.1 Source Files

The assembler accepts, as input, a source file that consists of dsPIC30FXXX 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® C30 C Compiler User's Guide* 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,"r"
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 Files

The assembler creates a relocatable COFF 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. “MPLAB ASM30 Command Line Interface”**) on the command line to override the default name.

1.5.3 Listing Files

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. “MPLAB ASM30 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,"r"
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

NOTES:

Chapter 2. MPLAB ASM30 Command Line Interface

2.1 INTRODUCTION

This chapter describes how to use the assembler from the command line and the command line options that are supported.

For information on using the assembler with MPLAB IDE, please refer to *dsPIC® 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 MPLAB ASM30 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.

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:

<code>c</code>	Omit false conditionals
<code>d</code>	Omit debugging directives
<code>h</code>	Include high-level source
<code>i</code>	Include section information
<code>l</code>	Include assembly
<code>m</code>	Include macro expansions
<code>n</code>	Omit forms processing
<code>s</code>	Include symbols
<code>=file</code>	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`.

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

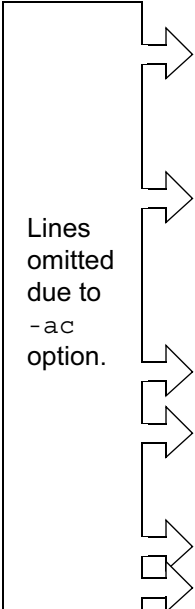
page 1

```
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
```

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

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

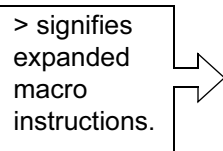
```

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

MPLAB ASM30 Listing: example2.6.s

page 1

> signifies
expanded
macro
instructions.



```
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
```

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          .sbtbl " 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.

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 bolded.

```
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 dsPIC specific command line options. The assembler then exits.

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 `--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.

2.6.2 `-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.

2.6.3 `-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.

2.6.4 `--relax`

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

2.6.5 `--no-relax`

Turn relaxation off. This is the default behavior.

2.6.6 `-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”.

2.6.7 `-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
```

Chapter 3. Assembler Syntax

3.1 INTRODUCTION

This chapter discusses syntax for MPLAB ASM30 source code.

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® C30 User's Guide* 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 neater 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]
```

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 *dsPIC® Programmer's Reference Manual* for more details.

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 *dsPIC[®] Programmer's Reference Manual*.) 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 `ax`. Octal numbers are distinguished by a leading `o`. Binary numbers are distinguished by a leading `B`. Decimal numbers require no special leading or trailing character.

Examples:

`#axe`, `#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 *dsPIC[®] Programmer's Reference Manual* 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 *dsPIC[®] Programmer's Reference Manual* 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 */
```

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-31415926535897932384626433832795028841971.693993751E-40
```

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.

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 or label
;	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

Chapter 4. Assembler Expression Syntax and Operation

4.1 INTRODUCTION

This chapter discusses expression syntax and operation for MPLAB ASM30.

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 units. On the dsPIC 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.

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, 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: OPERATORS

Operator	Description	Example
*	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)
	Bitwise Inclusive Or	2 4 (=6)
&	Bitwise And	4 & 6 (=4)
^	Bitwise Exclusive Or	4 ^ 6 (=2)
!	Bitwise Or Not	0x1010 ! 0x5050 (=0xBFBF)
+	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)

4.5 SPECIAL OPERATORS

The assembler provides a set of special operators for accessing data in program memory, obtaining the program address of a constant or symbol, and obtaining a handle to a program address. In addition, the assembler provides a set of special operators for obtaining the size of and starting address of a section that is resolved by the linker.

TABLE 4-3: SPECIAL OPERATORS

Operators	Description
<code>tblpage (name)</code>	Get page for table read/write operations
<code>tbloffset (name)</code>	Get pointer for table read/write operations
<code>psvpage (name)</code>	Get page for PSV data window operations
<code>psvoffset (name)</code>	Get pointer for PSV data window operations
<code>paddr (label)</code>	Get 24-bit address of <i>label</i> in program memory
<code>handle (label)</code>	Get 16-bit reference to <i>label</i> in program memory
<code>.sizeof. (name)</code>	Get size of section <i>name</i> in address units
<code>.startof. (name)</code>	Get starting address of section <i>name</i>

4.5.1 Accessing Data in Program Memory

The dsPIC 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, dsPIC 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 dsPIC 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 dsPIC 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 and the 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 *dsPIC[®] Programmer's Reference Manual*, 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 64K bytes. 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:

```
.text
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:

```
; 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 32K bytes. 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, "x"  
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 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 dsPIC device program counter increments by 2 for each instruction word.

4.5.5 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
```

NOTES:

Chapter 5. Assembler Symbols

5.1 INTRODUCTION

This chapter discusses what symbols are and how to use them with MPLAB ASM30.

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

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       . + 0xFFFFFFFF8

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

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 dsPIC 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 dsPIC 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.

Chapter 6. Assembler Directives

6.1 INTRODUCTION

This chapter discusses directives for MPLAB ASM30. While there are some similarities with MPASM assembler directives, most of these directives are new or different in some way. The differences between MPASM assembler and MPLAB ASM30 directives have been pointed out in **Appendix C. “MPASM™ Assembler Compatibility”**. All MPLAB ASM30 directives are preceded by a period “.”.

6.2 HIGHLIGHTS

Topics covered in this chapter are:

- Directives that Define Sections
- Directives that Modify How Program Memory is Filled
- Directives that Initialize Constants
- Directives that Declare Symbols
- Directives that Define Symbols
- Directives that Modify the Section Location Counter
- Directives that Format the Output Listing
- Conditional Assembler Directives
- Substitution/Expansion Assembler Directives
- Miscellaneous Assembler 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 dsPIC 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 dsPIC DSC memory. See **Section 10.8.1 “Standard Data Section Names”** for details.

Subsections provide a way of organizing section contents in the object file that is different from the organization of the source code. Subsections may be numbered from 0 to 8192. For example, a section will be organized in the object file such that all subsections numbered “0” appear first, followed by all subsections numbered “1”, and so on. Subsection numbering is optional.

For these directives, any previous code section that was active is aligned to the next word.

.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 [subsection]

Definition

Assemble the following statements onto the end of the `.data` (initialized data) subsection numbered subsection. `subsection` is optional and defaults to 0 if omitted.

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
```

.section name [, "flags"]
.section name [, subsection]

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 6-1: 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
```

.text [subsection]

Definition

Assemble the following statements onto the end of the `.text` (executable code) subsection numbered subsection. `subsection` is optional and defaults to 0 if omitted.

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 ASSEMBLER 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.

.fillupper [value]

Definition

In a code section, sets the value to put in the upper byte (bits 16-23) of program memory when this byte is skipped because of a `.fill` or a `.align`. If the value is not specified, the value is reset to the NOP opcode (0x00).

Example

See Section Example that follows.

.fillvalue [value]

Definition

In a code section, sets the value to put in the lower two bytes (bits 0-15) of program memory when these bytes are skipped because of a `.fill` or a `.align`. If the value is not specified, the value is reset to 0x0000.

Example

See Section Example that follows.

.pfillvalue [value]

Definition

Sets the value to put in program memory (bits 0-23) when bytes are skipped because of a `.pfill` or a `.palign`. If the value is not specified, the value is reset to 0x00.

Example

See Section Example below.

Section Example

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

6.5 ASSEMBLER DIRECTIVES THAT INITIALIZE CONSTANTS

.ascii "string₁" | <##>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.

.asciz “string₁” | <##>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.

.byte expr₁ [, ..., 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 expr₁ [, ..., 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 value₁ [, ..., 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 value₁[, ..., value_n]

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

.float value₁[, ..., 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 value₁[, ..., 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.

.hword expr₁[, ..., expr_n]

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

.int expr₁[, ..., expr_n]

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

Floating-point numbers are in IEEE format.

.long expr₁[, ..., expr_n]

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

Floating-point numbers are in IEEE format.

.short expr₁[, ..., expr_n]

Same as `.word`.

.string "str"

Same as `.asciz`.

.word expr₁[, ..., expr_n]

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

Floating-point numbers are in IEEE format.

.pword expr₁[, ..., 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 ASSEMBLER DIRECTIVES THAT DECLARE SYMBOLS

.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

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.

.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.

6.7 ASSEMBLER DIRECTIVES THAT DEFINE SYMBOLS

.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 ASSEMBLER DIRECTIVES THAT MODIFY THE SECTION LOCATION COUNTER

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

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

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 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.

.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. `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).

`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 dsPIC 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 dsPIC 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.

6.9 ASSEMBLER DIRECTIVES THAT FORMAT THE OUTPUT LISTING

.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 CONDITIONAL ASSEMBLER DIRECTIVES

.else

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

.elseif

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`.

.ifdef symbol

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

.ifndef symbol

.ifnotdef symbol

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

6.11 SUBSTITUTION/EXPANSION ASSEMBLER DIRECTIVES

.exitm

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

.irp symbol, value₁
[, ..., 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, value₁
[, ..., value_n]

...

.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 arg1[=default]
[, ..., argn [=default]]**

...

.endm

Define macros that generate assembly output. To refer to arguments within the macro block, use `\arg`.

For example, if this macro were defined

```
.macro display_int sym
mov \sym,w0
rcall display_w0
.endm
```

then assembling

```
display_int result
```

is equivalent to assembling

```
mov result,w0
rcall display_w0
```

.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 ASSEMBLER DIRECTIVES

.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. MPLAB LINK30 will ignore this section when allocating program and data memory, but will combine all `.comment` sections together, in link order.

.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.

.print “string”

Prints *string* on the standard output during assembly.

6.13 ASSEMBLER DIRECTIVES FOR DEBUG INFORMATION

.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

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

.sleb128 expr1 [, ..., exprn]

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.

.val addr

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

NOTES:



MPLAB® ASM30, MPLAB® LINK30 AND UTILITIES USER'S GUIDE

Part 2 – MPLAB LINK30 Linker

Chapter 7. Linker Overview	67
Chapter 8. MPLAB LINK30 Command Line Interface	73
Chapter 9. Linker Scripts	83
Chapter 10. Linker Processing	119

Part
2

MPLAB LINK30 Linker

NOTES:

Chapter 7. Linker Overview

7.1 INTRODUCTION

MPLAB LINK30 produces binary code from relocatable object code and archives for dsPIC devices. The linker is a Windows console application that provides a platform for developing executable code. The linker is a port of the GNU linker from the Free Software Foundation.

7.2 HIGHLIGHTS

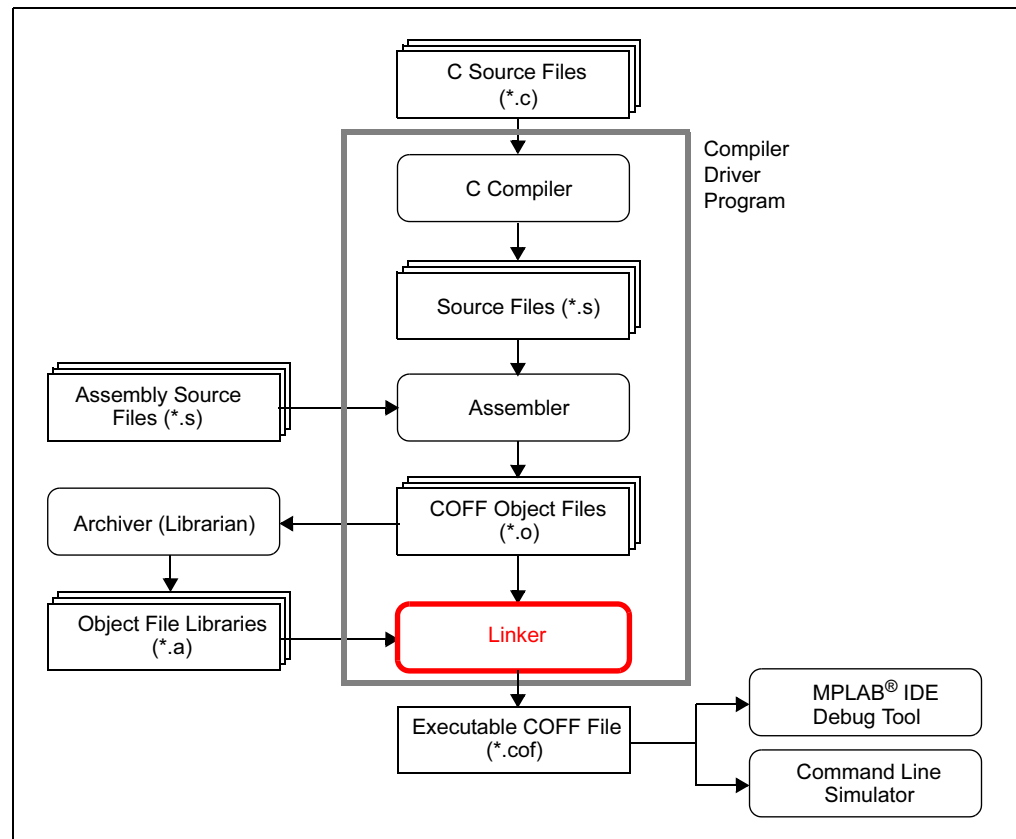
Topics covered in this chapter are:

- MPLAB LINK30 and Other Development Tools
- Feature Set
- Input/Output Files

7.3 MPLAB LINK30 AND OTHER DEVELOPMENT TOOLS

MPLAB LINK30 translates object files from the dsPIC assembler (MPLAB ASM30) and archives files from the dsPIC archiver/librarian (MPLAB LIB30) into an executable COFF file. See Figure 7-1 for an overview of the tools process flow.

FIGURE 7-1: TOOLS PROCESS FLOW



7.4 FEATURE SET

Notable features of the linker include:

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

7.5 INPUT/OUTPUT FILES

Linker input and output files are listed below.

TABLE 7-1: LINKER FILES

Input Files:	
.o	object file
.a	library file
.gld	linker script file
Output Files:	
.cof, .out	binary file
.map	map file

Unlike the MPLINK linker, MPLAB LINK30 does not generate absolute listing files. MPLAB LINK30 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.

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.

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.
- 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.

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)

Data Memory Usage

section	address	alignment gaps	total length (dec)
-----	-----	-----	-----
.bss	0x800	0	0x100 (256)
Total data memory used (bytes):			0x100 (256)

Dynamic Memory Usage

region	address	maximum length (dec)
-----	-----	-----
heap	0x900	0 (0)
stack	0x900	0x2f8 (760)
Maximum dynamic memory (bytes):		0x2f8 (760)

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
               0x00011a     _psv_init
               0x000106     _resetPRI
               0x00012a     _data_init

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

NOTES:

Chapter 8. MPLAB LINK30 Command Line Interface

8.1 INTRODUCTION

This chapter discusses MPLAB LINK30 command line interface.

For information on using the linker with MPLAB IDE, please refer to *dsPIC® Language Tools Getting Started* (DS70094).

8.2 HIGHLIGHTS

Topics covered in this chapter are:

- Syntax
- Options that Control Output File Creation
- Options that Control Runtime Initialization
- Options that Control Informational Output
- Options that Modify the Link Map Output

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`.

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

8.4.1 `--architecture arch (-A arch)`

Set architecture.

The architecture argument identifies the particular architecture in the dsPIC DSC family, 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 <i>sym</i> , the equals sign (" <code>=</code> ") and <i>expr</i> .

8.4.5 --discard-all (-x)

Discard all local symbols.

8.4.6 --discard-locals (-X)

Discard temporary local symbols.

8.4.7 --force-exe-suffix

Force generation of file with .exe suffix.

8.4.8 --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 `liblibname.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 `-r` 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.

8.4.9 --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.

8.4.10 --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.

8.4.11 --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.

8.4.12 --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.

8.4.13 --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.

8.4.14 --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`.

8.4.15 --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.

8.4.16 --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.

8.4.17 --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.

8.4.18 --sort-common

Sort common symbols by size.

This option tells `pic30-ld` to sort the common symbols by size when it places them in the appropriate output sections. First come all of the one byte symbols, then all of the two bytes, then all of the four bytes, and then everything else. This is to prevent gaps between symbols due to alignment constraints.

8.4.19 --strip-all (-s)

Strip all symbols.

Omit all symbol information from the output file.

8.4.20 --strip-debug (-S)

Strip debugging symbols.

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

8.4.21 -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.

8.4.22 -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.

8.4.23 -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.

8.4.24 --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.

8.4.25 --no-undefined

Allow no undefined symbols.

8.4.26 --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 RUNTIME INITIALIZATION

8.5.1 --data-init

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

Create a special output section named `.dinit` as a template for the runtime initialization of data. The C startup 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 runtime initialization of data. When this option is specified, the linker will select a shorter form of the C startup 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 runtime 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.

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 0xFF. This option consumes additional program memory and causes the template to appear as `NOPR` instructions if viewed in the disassembler (and will be executed as such by the dsPIC 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 runtime 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

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 <code>-l</code> .

8.6.5 --trace (-t)

Trace file.

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

8.6.6 --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.

8.6.7 -V

Print version and other information.

8.6.8 --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.

8.6.9 --version (-v)

Print version information.

8.6.10 --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
```

8.6.11 `--warn-once`

Warn only once per undefined symbol.

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

8.6.12 `--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

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 -M option, above.

Chapter 9. Linker Scripts

9.1 INTRODUCTION

This chapter discusses how to use and customize linker scripts to control MPLAB LINK30 functions.

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

The dsPIC Language Tools include a set of standard linker scripts: device-specific linker scripts (e.g., `p30f3014.gld`) and one generic linker script (`p30sim.gld`).

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.

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:

- Output File Format and Entry Point
- Memory Region Information
- Base Memory Address
- Input/Output Section Map
- Range Checking for Near and X Data Memory
- Interrupt Vector Tables
- SFR Addresses

9.5.1 Output File Format and Entry Point

The first several lines of a linker script define the output format, processor family and entry point:

```
/*
** Linker Script for p30f3014
*/
OUTPUT_FORMAT("coff-pic30")
OUTPUT_ARCH("pic30")
EXTERN(__resetPRI)
EXTERN(__resetALT)
ENTRY(__reset)
```

Future versions of MPLAB LINK30 may support additional output file formats and/or processor families. If so, the `OUTPUT_FORMAT` and `OUTPUT_ARCH` commands may be modified, respectively. The `EXTERN` commands force two C runtime startup modules to be loaded from archives. The linker will select one and discard the other, based on the `--data-init` option. The `ENTRY` command denotes the application entry point. By convention, the application entry point is named `__reset`.

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 (a!xr) : ORIGIN = 0x800,      LENGTH = 2048
    program (xr) : 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)
    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.

In the following sections, each `MEMORY` region will be discussed.

9.5.2.1 DATA REGION

```
data (a!xr) : 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). The region attributes `(a!xr)` specify how unmapped sections are to be handled. Unmapped sections that are marked allocatable (`a`) but not executable or read-only (`!xr`) should be placed in region `data`.

Any unmapped output section with attributes “allocatable, but not executable” will be assigned to region `data`, after all other sections have been assigned.

Note: Unmapped output sections occur when a new section is created in source code, but not defined in the linker script. The default region attributes are set so that a successful link with unmapped sections is likely. Best practice would ensure that any user-defined sections are explicitly mapped in the linker script.

9.5.2.2 PROGRAM REGION

```
program (xr) : 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 dsPIC DSC 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.

The region attributes (xr) specify how unmapped sections are to be handled. Unmapped sections that are marked executable (x) or read-only (r) should be placed in region program.

Any unmapped output section with attribute “executable” or “read-only” will be assigned to region program, after all other sections have been assigned.

Note: Section attributes may be specified in source code. See **Section 6.3 “Directives that Define Sections”** for details.

9.5.2.3 RESET, IVT AND AIVT REGIONS

```
reset          : ORIGIN = 0,          LENGTH = (4)
```

The reset region corresponds to the dsPIC 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)
__FWDTC     : 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.

9.5.2.5 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.

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 dsPIC `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.

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 __CODE_BASE :
{
    *(.handle);
    *(.libc) *(.libm) *(.libdsp); /* keep together in this order */
    *(.lib*);
    *(.text);
} >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. The input section `.handle` is used for function pointers and is loaded first at low addresses. This is followed by the library sections `.libc`, `.libm` and `.libdsp`. These sections 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`). Finally input sections names `.text` are included.

Note: Input section `.text` is reserved for application code. MPLAB ASM30 will automatically locate code in section `.text` unless instructed otherwise.

9.5.4.3 DATA INITIALIZATION TEMPLATE

Section `.dinit` is created by the linker and contains information about uninitialized (`.bss`) and initialized (`.data`) sections in data memory. This information is used by the C startup module (`crt0.o`) in the runtime library `libpic30.a` to initialize data memory before the application's main entry point is called.

```
/*
** Initialized Data Template
**/
.dinit:
{
    *(.dinit);
} >program
```

For information about data initialization, see **Section 10.8.2 “Data Initialization Template”**.

9.5.4.4 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.”

```
/*
** 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.

9.5.4.5 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
```

The Configuration Fuse sections are supported by macros defined in the dsPIC 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.

9.5.4.6 DATA FLASH MEMORY

Section `.eedata` corresponds to the data flash memory on certain dsPIC devices. Although located in program memory space, data flash memory is organized like physical data memory.

```
/*
** Data Flash Memory
*/
.eedata :
{ *(.eedata) } >eedata
```

For example, to declare an array in data flash memory in assembly language:

```
.section .eedata,"r"
.global _mydata
_mydata:
.byte 1,2,3,4,5,6,7,8,9
.byte 0xa,0xb,0xc,0xd,0xe,0xf
```

The equivalent operation in C would utilize the `_EEDATA()` macro defined in the device-specific header files. The `_EEDATA()` macro takes one parameter, which specifies address alignment. The parameter must be an integer, a power of 2, and not less than 2. Unless a greater alignment is required for modulo buffer addressing, the value of 2 should be used, as shown:

```
#include <p30f3014.h>

char _EEDATA(2) mydata [] =
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf };
```

9.5.4.7 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.

9.5.4.8 OUTPUT SECTIONS IN X DATA MEMORY

RAM memory in dsPIC devices is logically separated into X and Y data spaces. Certain DSP instructions rely on the allocation of variables into one space or the other.

Two sections are provided for allocating variables in X data memory: `.xbss` for static or non-initialized variables, and `.xdata` for initialized variables:

```
/*
** X Static Data
*/
.xbss (NOLOAD):
{
    __bxdata = .;
    *(.xbss);
} >data

/*
** X Initialized Data
*/
.xdata :
{
    *(.xdata);
    __exdata = .;
} >data
```

Notice that section `.xbss` includes the `(NOLOAD)` attribute, while `.xdata` does not. The `(NOLOAD)` attribute implies no initial values. The presence or absence of this attribute tells the linker which sections need to be cleared (bss type) and which sections need to be initialized (data type).

Note: All sections in data memory are considered to be either bss type or data type.

When the linked output file is created, the initial values of data type sections are stored in section `.dinit`, the data initialization template in program memory.

9.5.4.9 OUTPUT SECTIONS IN NEAR DATA MEMORY

Certain dsPIC instructions rely on the allocation of variables in the lowest 8K of data space. This address range is referred to as near data memory.

Three sections are provided for allocating variables in near data memory: `.pbss` for persistent data, `.nbss` for static or non-initialized variables, and `.ndata` for initialized variables:

```
/*
** Persistent Data
*/
.pbss (NOLOAD):
{
    *(.pbss);
} >data

/*
** NEAR Static Data
*/
.nbss (NOLOAD):
{
    __bndata = .;
    *(.nbss);
} >data

/*
** NEAR Initialized Data and Constants
*/
.ndata :
{
    *(.ndata);
    *(.ndconst);
    __endata = .;
} >data
```

Section `.nbss` includes the `(NOLOAD)` attribute, while `.ndata` does not. The `(NOLOAD)` attribute implies no initial values. The presence or absence of this attribute tells the linker which sections need to be cleared (bss type) and which sections need to be initialized (data type).

9.5.4.10 OUTPUT SECTIONS IN GENERAL DATA MEMORY

Two sections are provided for allocating variables in general data memory: `.bss` for static or non-initialized variables, and `.data` for initialized variables:

```
/*
** Static Data
*/
.bss (NOLOAD):
{
    *(.bss);
} >data

/*
** Initialized Data and Constants
*/
.data :
{
    *(.data);
    *(.dconst);
} >data
```

Note that sections `.pbss` and `.bss` include the (NOLOAD) attribute, while `.data` does not. The (NOLOAD) attribute implies no initial values. The presence or absence of this attribute tells the linker which sections need to be cleared (bss type) and which sections need to be initialized (data type).

9.5.4.11 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.”

```
/*
** 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.

9.5.4.12 OUTPUT SECTIONS IN Y DATA MEMORY

RAM memory in dsPIC devices is logically separated into X and Y data spaces. Certain DSP instructions rely on the allocation of variables into one space or the other.

Two sections are provided for allocating variables in Y data memory: `.ybss` for static or non-initialized variables, and `.ydata` for initialized variables:

```
/*
** Y Static Data
*/
.ybss MAX( __YDATA_BASE , ALIGN(2)) (NOLOAD):
{
    *(.ybss);
} >data

/*
** Y Initialized Data
*/
.ydata MAX( (__YDATA_BASE + SIZEOF( .ybss)), ALIGN(2)) :
{
    *(.ydata);
} >data
```

The starting address of section `.ybss` is defined with a `MAX()` expression, which may be interpreted as:

Use the constant value `__YDATA_BASE`, or the current location counter with 2 byte alignment, whichever is greater.

Through the use of a `MAX()` expression for the starting address, section `.ybss` is allowed to float above other general purpose memory sections, but ensured not to fall below the start of Y data space. If necessary, the location counter will skip intervening space in order to reach the Y data space base address.

The `(NOLOAD)` attribute of section `.ybss` implies that this section is uninitialized.

The starting address of section `.ydata` is also defined with a `MAX()` expression. The `MAX()` expression specifies that `.ydata` will immediately follow `.ybss` even if `.ybss` has no contents. This permits efficient memory allocation, since both `.ybss` and `.ydata` can float above general purpose memory sections.

<p>Note: If the standard linker script is customized, make certain that no section is inserted between <code>.ybss</code> and <code>.ydata</code>, because that would interfere with the <code>MAX()</code> expression.</p>
--

9.5.4.13 OUTPUT SECTION .CONST

Output section `.const` is designed for use with the Program Space Visibility (PSV) window:

```
/*
** Constants in Program Memory
**
** This section is loaded into program memory and then
** mapped into data memory using the PSV data window.
** Section .const must be declared with the "r" section
** attribute, which identifies it as READONLY data.
*/
.const :
{
    *(.const);
} >program
```

The linker ensures that the entire contents of output section `.const` can be accessed with a single setting of the PSVPAG register. This enables efficient access of constant arrays. If section `.const` is used, the C runtime startup module will initialize the PSV window automatically.

9.5.5 Range Checking for Near and X Data Memory

Two range check expressions are included for the X data memory space and the Near data memory space:

```
/*
** Calculate overflow of X and Near data space
**
__X_OVERFLOW      = (((__exdata != __bxdata) && (__exdata >
__YDATA_BASE)) ?
                    (__exdata - __YDATA_BASE) : 0);
__NEAR_OVERFLOW = (((__endata != __bndata) && (__endata > 0x2000)) ?
                    (__endata - 0x2000) : 0);
```

These expressions calculate overflow (if any) for sections that are assigned to the X and Near data space. Note that the X data space limit varies by device, while the Near data space limit is fixed at 8K bytes, or address 0x2000. If either type of section extends past its respective addressing boundary, a link error will be reported.

Range checking for all other sections is provided as the memory regions are filled. A link error will be reported if any section falls outside of its assigned memory region.

9.5.6 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.

<p>Note: The programmer must insure that functions installed in interrupt vector tables conform to the architectural requirements of interrupt service routines.</p>

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.

9.5.7 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.

9.6 CREATING A CUSTOM LINKER SCRIPT

The standard dsPIC linker scripts are general purpose and will satisfy the demands of most applications. However, occasions may arise where a custom linker script is required. Such is the case when an exact absolute address must be specified for a user-defined section.

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.

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 means 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 8.4.9 “--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”**.

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 8.4.12 “--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 8.4.9 “--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.

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.

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 preceding 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.

9.7.5.1 OUTPUT SECTION DESCRIPTION

The full description of an output section looks like this:

```
section [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 section is required, so that the section name is unambiguous. 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 9.7.5.3 “Input Section Description”**)
- data values to include directly (see **Section 9.7.5.7 “Output Section Data”**)

9.7.5.2 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.

9.7.5.3 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.

9.7.5.4 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.

<code>*</code>	matches any number of characters
<code>?</code>	matches any single character
<code>[chars]</code>	matches a single instance of any of the <i>chars</i> ; the <code>-</code> character may be used to specify a range of characters, as in <code>[a-z]</code> to match any lower case letter
<code>\</code>	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) }
}
```

9.7.5.5 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`.

9.7.5.6 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)
  }
}
```

9.7.5.7 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) } }
```

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:

```
section [address] [(type)] : [AT(lma)]
{
    output-section-command
    output-section-command
    ...
} [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

Section, 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 runtime 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. MPLAB LINK30 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.

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 runtime 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, MPLAB LINK30 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));
```

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 dsPIC 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 MPLAB LINK30 assign space to common symbols even if a relocatable output file is specified (`-r`).

NOCROSSREFS(*section section ...*)

This command may be used to tell MPLAB LINK30 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(*arch*)

Specify a particular output machine architecture. dsPIC DSCs currently have only one machine architecture, "pic30".

OUTPUT_FORMAT(*format_name*)

The **OUTPUT_FORMAT** command names the object file format to use for the output file. The dsPIC language tools currently support one object file format, "coff-pic30".

TARGET(*bfdname*)

The **TARGET** command names the object file format to use when reading input files. It affects subsequent **INPUT** and **GROUP** commands. The dsPIC language tools currently support one object file format, "coff-pic30".

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.

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.

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	Notes
highest	1	left	! - ~	Prefix operators
	2	left	* / %	
	3	left	+ -	
	4	left	>> <<	
	5	left	== != > < <= >=	
	6	left	&	
	7	left		
	8	left	&&	
	9	left		
	10	right	? :	
lowest	11	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.

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.

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

$$(. + \text{exp} - 1) \& \sim(\text{exp} - 1)$$

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”**).

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:

Chapter 10. Linker Processing

10.1 INTRODUCTION

This chapter discusses how MPLAB LINK30 builds an application from input files.

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

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. Only output sections have an absolute address. Input sections are always relocatable.

If an input section is not explicitly assigned to an output section, the linker will allocate the unassigned section according to default rules specified in the linker script. 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.

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”**.

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 runtime 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 dsPIC architecture provides two ways to accomplish this task: table access instructions and the 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 **Section 4.5.1.1 “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, "x"
L1: .pbyte 1
L2: .pbyte 2
L3: .pbyte 3
L4: .pbyte 4
    .pbyte 5
    .pbyte 6
    .pbyte 7,8,9
```

In this example, the “x” 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, "x"
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()` facilitate PSV access in assembly language. See **Section 4.5.1.2 “Program Space Visibility (PSV) Data Window”** for more information.

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”**.

10.5 LINKER ALLOCATION

During the allocation phase, each input section must be assigned to a specific memory region in the target device. Addresses within a memory region are allocated sequentially, beginning with the lowest address and growing upwards.

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

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 __CODE_BASE :
{
    *(.handle);
    *(.libc) *(.libm) *(.libdsp); /* keep together in this order */
    *(.lib*);
    *(.text);
} >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:

- First, all input sections named `.handle` are collected and mapped into the output section. This means that `.handle` sections will occupy the lowest address range, a requirement for code handles.
- Second, input sections named `.libc`, `.libm` and `.libdsp` are collected and mapped into the output section. Grouping these sections ensures locality of reference for the runtime library functions, so that PC-relative instructions can be used for maximum efficiency.
- Third, 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`).
- Finally, all input sections named `.text` are collected and mapped into the output section. These sections contain executable application code, and will occupy the highest address range.

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 linker will select one based on region attributes (see **Section 10.5.3 “Allocating Unmapped Sections”**).

Memory regions are filled sequentially, from lower to higher addresses, in the same order that sections appear in the section map. 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 cannot be recovered. 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 `r` 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.

10.5.3 Allocating Unmapped Sections

After all sections that appear in the section map have been allocated, any remaining input sections are considered to be unmapped. Unmapped sections occur when a new section is created in source code, but not defined in the linker script. For each unmapped input section, the linker creates an output section with the same name. Unmapped output sections are then assigned to memory regions, based on region attributes specified in the linker script.

The linker supports the following region attributes, which are related to the section flags described in **Section 6.3 “Directives that Define Sections”**.

TABLE 10-1: REGION ATTRIBUTES

Region Attribute	Description	Related Section Flag
w	Read/write section	b
x	Executable section	x
a	Allocatable section	b, x, d or n
r	Read-only section	r
i or l	Initialized section	d
!	Invert the sense of any following attributes	n/a

Memory region attributes are optional. If region attributes are specified, they can help the linker determine where an unmapped section should be allocated. If region attributes are not specified, the linker will allocate unmapped sections into the first memory region.

For example, consider this memory region definition:

```
data    (a!xr) : ORIGIN = 0x800,      LENGTH = 2048
```

Here region data is defined with attributes (a!x). This means that unmapped sections that are allocatable (a) but not executable or read-only (!xr) will be assigned to region data.

Region attributes in the standard dsPIC linker scripts are set so that a successful link with unmapped sections is likely. Best practice would ensure that any user-defined sections are explicitly mapped in the linker script.

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, MPLAB ASM30 and MPLAB LINK30 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 handle relocation. 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 dsPIC 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. Several standard data sections have been defined which correspond to X data memory, near data memory, general data memory and Y data memory. These sections are located at the appropriate addresses by the linker script.

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 startup by the runtime library. When the application main program takes control, all variables in data memory have been initialized.

10.8.1 Standard Data Section Names

Traditionally, linkers based on the GNU technology support three sections in the linked binary file:

TABLE 10-2: DATA SECTION NAMES

Traditional Section Name	Description
.text	executable code
.data	data memory that receives initial values
.bss	data memory that is not initialized

The name “bss” dates back several decades, and means memory “Block Started by Symbol”. By convention, bss memory is filled with zeros during program startup.

The dsPIC Language Tools define several standard sections beyond the traditional three. The additional standard sections allow an application to locate initialized variables and non-initialized variables in X data memory, Y data memory, near data memory and general data memory. Together they provide great flexibility for application development. The standard dsPIC linker scripts automatically support the following sections in data memory:

TABLE 10-3: dsPIC DATA SECTIONS

dsPIC Section Name	Description	Section Flag
.xbss	X memory that is not initialized	“b”
.xdata	X memory that receives initial values	“d”
.nbss	Near memory that is not initialized	“b”
.ndata	Near memory that receives initial values	“d”
.ndconst	Constants in Near memory	“d”
.bss	General memory that is not initialized	“b”
.pbss	Persistent data memory	“b”
.data	General memory that receives initial values	“d”
.dconst	Constants in general memory	“d”
.ybss	Y memory that is not initialized	“b”
.ydata	Y memory that receives initial values	“d”

Note that bss-type sections (memory that is not initialized) have a “b” section flag, while data-type sections (memory that receives initial values) have a “d” section flag. For details on how to declare variables in specific data sections from C, refer to the *MPLAB® C30 User's Guide*. Applications written in assembly code can also take advantage of the standard data section names to allocate variables in data memory.

To declare a non-initialized buffer of 32 bytes in X data memory, use the following assembly code:

```
.section .xbss, "b"
.space 32
```

This example instructs the linker to allocate a buffer of 32 bytes in X data space. The buffer does not receive initial values, but will be filled with zeros at startup. Note that using double quotes around the section flag is important. A character in single quotes is converted to a number by the assembler pre-processor. Section flags must be declared with double quotes.

To declare an initialized table of values in Y data memory, using hexadecimal notation:

```
.section .ydata, "d"
.word 0x00, 0x11, 0x22, 0x33
.word 0x44, 0x55, 0x66, 0x77
```

This example instructs the linker to allocate a table of 8 constants in Y data space. Each constant is a word (2 bytes) and the initial values are copied in at startup.

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 flags, 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 dsPIC Language Tools support several standard 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 startup, and the bss-type sections are filled with zeros.

It would be inefficient for the runtime library to include logic dedicated to each of these standard sections. Also, dedicated logic could not accommodate user-defined sections, or any other sections whose names were not known at the time the library was created.

To remedy this situation, 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. Startup code in the runtime 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 10-4: 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 `0xFF` and causes the template to appear as `NOPR` instructions if viewed in the disassembler (and will be executed as such by the dsPIC device).

10.8.3 Runtime Library Support

In order to initialize variables in data memory, the data initialization template must be processed at startup, before the proper application takes control. For C programs, this function is performed by the startup 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 startup modules is provided in file `crt0.s` and `crt1.s`.

To utilize a startup module, the application must allow the runtime library to take control at device reset. This happens automatically for C programs. The application's `main()` function is invoked after the startup module has completed its work. Assembly language programs should use the following naming conventions to specify which routine takes control at device reset.

TABLE 10-5: MAIN ENTRY POINTS

Main Entry Name	Description
<code>__reset</code>	Takes control immediately after device reset
<code>_main</code>	Takes control after the startup 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 startup 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 MPLAB C30 command line.

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.

Note: The persistent data section `.pbss` is never cleared or initialized.

4. The function `main` is called with no parameters.
5. If `main` returns, the processor will reset.

The alternate startup 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 startup 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 dsPIC assembly language) for both modules is provided in the `c:\pic30_tools\src` directory. The startup 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 “r” 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,"r"
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 32K bytes. Only the least significant 16 bits of each instruction word are available for data storage. 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 “r” section attribute, the linker will locate the section after other sections 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 "r" section attribute. It will be
** loaded into program memory and mapped into data
** memory using the PSV window.
*/
userconstants ADDR :
{
    *(userconstants);
} >program
```

In this example, ADDR is optional and can be used to specify an absolute address. If the specified address causes the section to cross a PSV page boundary, a warning message will be issued. If no address is specified, the linker may reorganize sections in program memory to obtain the best fit while respecting PSV page and section alignment requirements.

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.

10.10 STACK ALLOCATION

The dsPIC 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 dsPIC 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, MPLAB LINK30 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, MPLAB LINK30 increases the minimum required size by a small amount to accommodate 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 = .;
} > data
```

In the user-defined section, two symbols are declared `__SP_init` and `__SPLIM_init` for use by the startup 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). 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 startup module uses these symbols to initialize the stack pointer and stack pointer limit register. Normally the startup module is provided by `libpic30.a` (for C programs) or `crt0.o` (for assembly programs). In special cases, the application may provide its own startup 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 MPLAB C30 standard C library, `libsx1.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 dsPIC DSC 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

The dsPIC DSC has two interrupt vector tables, a primary and an alternate table, each containing 62 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.

MPLAB C30 provides a special syntax for writing interrupt handlers. See the *MPLAB® C30 C Compiler User's Guide* 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. Unused vector table entries default to the `RESET` instruction, which resets the device.

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 runtime environment.

Applications may also provide a default interrupt handler. In assembly language, the name of the default interrupt handler is `__DefaultInterrupt`. In C the name is `_DefaultInterrupt`. Note that C requires only one leading underscore for any of the interrupt handler names.

The follow 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,"b"           ; persistent data storage
    .global FaultCount           ; is not affected by reset
FaultCount:
    .space 2                     ; count of unexpected interrupts
```

The standard naming convention for interrupt handlers is described in Table 10-6.

TABLE 10-6: STANDARD NAMING CONVENTIONS

IRQ#	Vector Function	Primary Name	Alternate Name
n/a	Reserved	__ReservedTrap0	__AltReservedTrap0
n/a	Oscillator fail trap	__OscillatorFail	__AltOscillatorFail
n/a	Address error trap	__AddressError	__AltAddressError
n/a	Stack error trap	__StackError	__AltStackError
n/a	Math error trap	__MathError	__AltMathError
n/a	Reserved	__ReservedTrap5	__AltReservedTrap5
n/a	Reserved	__ReservedTrap6	__AltReservedTrap6
n/a	Reserved	__ReservedTrap7	__AltReservedTrap7
0	INT0-External interrupt 0	__INT0Interrupt	__AltINT0Interrupt
1	IC1-Input capture 1	__IC1Interrupt	__AltIC1Interrupt
2	OC1-Output compare 1	__OC1Interrupt	__AltOC1Interrupt
3	TMR1-Timer 1	__T1Interrupt	__AltT1Interrupt
4	IC2-Input capture 2	__IC2Interrupt	__AltIC2Interrupt
5	OC2-Output compare 2	__OC2Interrupt	__AltOC2Interrupt
6	TMR2-Timer 2	__T2Interrupt	__AltT2Interrupt
7	TMR3-Timer 3	__T3Interrupt	__AltT3Interrupt
8	SPI™1-Serial peripheral interface 1	__SPI1Interrupt	__AltSPI1Interrupt
9	UART1RX-UART1 Receiver	__U1RXInterrupt	__AltU1RXInterrupt
10	UART1TX-UART1 Transmitter	__U1TXInterrupt	__AltU1TXInterrupt
11	ADC-ADC convert done	__ADCInterrupt	__AltADCInterrupt
12	NVM-NVM write complete	__NVMInterrupt	__AltNVMInterrupt
13	Slave I ² C Interrupt	__SI2CInterrupt	__AltSI2CInterrupt
14	Master I ² C Interrupt	__MI2CInterrupt	__AltMI2CInterrupt
15	CN-Input change interrupt	__CNInterrupt	__AltCNInterrupt
16	INT1-External interrupt 1	__INT1Interrupt	__AltINT1Interrupt
17	IC7-Input capture 7	__IC7Interrupt	__AltIC7Interrupt
18	IC8-Input capture 8	__IC8Interrupt	__AltIC8Interrupt
19	OC3-Output compare 3	__OC3Interrupt	__AltOC3Interrupt
20	OC4-Output compare 4	__OC4Interrupt	__AltOC4Interrupt
21	TMR4-Timer 4	__T4Interrupt	__AltT4Interrupt
22	TMR5-Timer 5	__T5Interrupt	__AltT5Interrupt
23	INT2-External interrupt 2	__INT2Interrupt	__AltINT2Interrupt
24	UART2RX-UART2 receiver	__U2RXInterrupt	__AltU2RXInterrupt
25	UART2TX-UART2 transmitter	__U2TXInterrupt	__AltU2TXInterrupt
26	SPI2-Serial peripheral interface 2	__SPI2Interrupt	__AltSPI2Interrupt
27	CAN1-Combined IRQ	__C1Interrupt	__AltC1Interrupt
28	IC3-Input capture 3	__IC3Interrupt	__AltIC3Interrupt
29	IC4-Input capture 4	__IC4Interrupt	__AltIC4Interrupt
30	IC5-Input capture 5	__IC5Interrupt	__AltIC5Interrupt
31	IC6-Input capture 6	__IC6Interrupt	__AltIC6Interrupt
32	OC5-Output compare 5	__OC5Interrupt	__AltOC5Interrupt
33	OC6-Output compare 6	__OC6Interrupt	__AltOC6Interrupt
34	OC7-Output compare 7	__OC7Interrupt	__AltOC7Interrupt
35	OC8-Output compare 8	__OC8Interrupt	__AltOC8Interrupt

TABLE 10-6: STANDARD NAMING CONVENTIONS (CONTINUED)

IRQ#	Vector Function	Primary Name	Alternate Name
36	INT3-External interrupt 3	__INT3Interrupt	__AltINT3Interrupt
37	INT4-External interrupt 4	__INT4Interrupt	__AltINT4Interrupt
38	CAN2-Combined IRQ	__C2Interrupt	__AltC2Interrupt
39	PWM-PWM period match	__PWMInterrupt	__AltPWMInterrupt
40	QE1-Position counter compare	__QE1Interrupt	__AltQE1Interrupt
41	DCI-CODEC transfer done	__DCIInterrupt	__AltDCIInterrupt
42	PLVD-Low voltage detect	__LVDInterrupt	__AltLVDInterrupt
43	FLTA-MPWM fault A	__FLTAInterrupt	__AltFLTAInterrupt
44	FLTB-MPWM fault B	__FLTBInterrupt	__AltFLTBInterrupt
45	Reserved	__Interrupt45	__AltInterrupt45
46	Reserved	__Interrupt46	__AltInterrupt46
47	Reserved	__Interrupt47	__AltInterrupt47
48	Reserved	__Interrupt48	__AltInterrupt48
49	Reserved	__Interrupt49	__AltInterrupt49
50	Reserved	__Interrupt50	__AltInterrupt50
51	Reserved	__Interrupt51	__AltInterrupt51
52	Reserved	__Interrupt52	__AltInterrupt52
53	Reserved	__Interrupt53	__AltInterrupt53

NOTES:



MPLAB® ASM30, MPLAB® LINK30 AND UTILITIES USER'S GUIDE

Part 3 – MPLAB LIB30 Archiver/Librarian

Chapter 11. MPLAB LIB30 Archiver/Librarian	139
--	-----

NOTES:

Chapter 11. MPLAB LIB30 Archiver/Librarian

11.1 INTRODUCTION

MPLAB LIB30 (`pic30-ar`) creates, modifies and extracts files from archives. 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.

MPLAB LIB30 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 MPLAB LIB30 called `ranlib` can be used to add only the table.

MPLAB LIB30 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.

11.2 HIGHLIGHTS

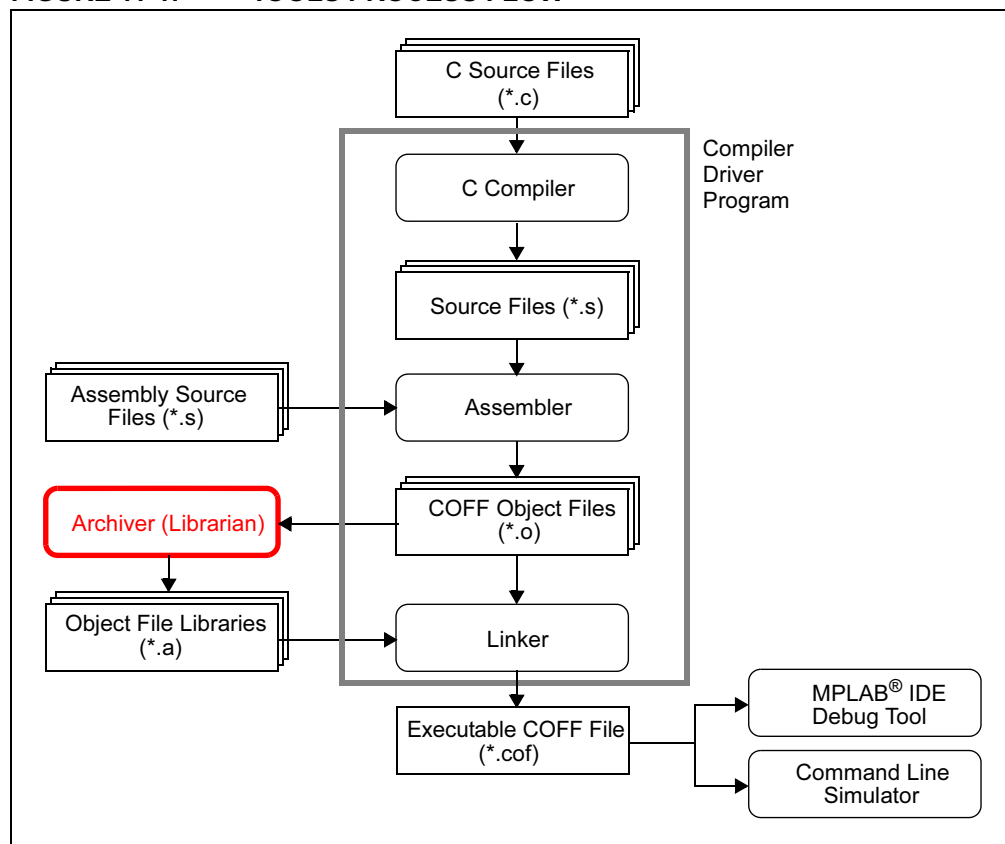
Topics covered in this chapter are:

- MPLAB LIB30 and Other Development Tools
- Feature Set
- Input/Output Files
- Syntax
- Options
- Scripts

11.3 MPLAB LIB30 AND OTHER DEVELOPMENT TOOLS

MPLAB LIB30 creates an archive file from object files created by the dsPIC assembler (MPLAB ASM30). Archive files may then be linked by the dsPIC linker (MPLAB LINK30) with other relocatable object files to create an executable COFF file. See Figure 11-1 for an overview of the tools process flow.

FIGURE 11-1: TOOLS PROCESS FLOW



11.4 FEATURE SET

Notable features of the assembler include:

- Available for Windows
- Command Line Interface

11.5 INPUT/OUTPUT FILES

MPLAB LIB30 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.

11.6 SYNTAX

```
pic30-ar [-]P[MOD [RELPOS] [COUNT]] ARCHIVE [MEMBER...]  
pic30-ar -M [ <mri-script ]
```

11.7 OPTIONS

When you use MPLAB LIB30 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.

MPLAB LIB30 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 11-1: OPERATION TO EXECUTE

Option	Function
d	Delete modules from the archive. Specify the names of modules to be deleted as <i>MEMBER . . .</i> ; the archive is untouched if you specify no files to delete. If you specify the <i>v</i> modifier, MPLAB LIB30 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 <i>m</i> , any members you name in the <i>MEMBER</i> arguments are moved to the end of the archive; you can use the <i>a</i> , <i>b</i> or <i>i</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 <i>v</i> modifier is specified, show the member name before copying its contents to standard output. If you specify no <i>MEMBER</i> arguments, all the files in the archive are printed.
q	Append the files <i>MEMBER . . .</i> into <i>ARCHIVE</i> .
r	Insert the files <i>MEMBER . . .</i> into <i>ARCHIVE</i> (with replacement). If one of the files named in <i>MEMBER . . .</i> 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 <i>a</i> , <i>b</i> or <i>i</i> to request placement relative to some existing member. The modifier <i>v</i> used with this operation elicits a line of output for each file inserted, along with one of the letters <i>a</i> or <i>r</i> to indicate whether the file was appended (no old member deleted) or replaced.
t	Display a table listing the contents of <i>ARCHIVE</i> , or those of the files listed in <i>MEMBER . . .</i> , 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 <i>v</i> modifier. If you do not specify a <i>MEMBER</i> , all files in the archive are listed. For example, if there is more than one file with the same name (<i>file</i>) in an archive (<i>b.a</i>), then <code>pic30-ar t b.a file</code> lists only the first instance; to see them all, you must ask for a complete listing in <code>pic30-ar t b.a</code> .
x	Extract members (named <i>MEMBER</i>) from the archive. You can use the <i>v</i> modifier with this operation, to request that the archiver list each name as it extracts it. If you do not specify a <i>MEMBER</i> , 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 11-2: 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. MPLAB LIB30 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. MPLAB LIB30 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 MPLAB LIB30.

11.8 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 MPLAB LIB30 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 MPLAB LIB30 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 MPLAB LIB30 for developers who already have scripts written for the MRI “librarian” program.

The syntax for the MPLAB LIB30 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 11-3 shows the commands you can use in archiver scripts, or when using the archiver interactively. Three of them have special significance.

TABLE 11-3: 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 11-3: 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 <code>pic30-ar -d ARCHIVE MODULE ... MODULE</code> . 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 <code>pic30-ar -t ARCHIVE MODULE...</code> When verbose output is on, the listing is like <code>pic30-ar -tv ARCHIVE MODULE...</code> Output normally goes to the standard output stream; however, if you specify OUTPUTFILE as a final argument, MPLAB LIB30 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 <code>pic30-ar -x ARCHIVE MODULE...</code> 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 <code>pic30-ar tv ARCHIVE</code> . (This single command is an MPLAB LIB30 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 <code>pic30-ar -tv ...</code>
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.

Part 4 – Utilities

Chapter 12. Utilities Overview	147
Chapter 13. pic30-bin2hex Utility	149
Chapter 14. pic30-nm Utility	151
Chapter 15. pic30-objdump Utility	155
Chapter 16. pic30-ranlib Utility	159
Chapter 17. pic30-strings Utility	161
Chapter 18. pic30-strip Utility	163
Chapter 19. pic30-lm Utility	165

NOTES:

Chapter 12. Utilities Overview

12.1 INTRODUCTION

This chapter discusses general information about the utilities.

12.2 HIGHLIGHTS

Topics covered in this chapter are:

- What are Utilities

12.3 WHAT ARE UTILITIES

Utilities are tools available for use with MPLAB ASM30 and/or MPLAB LINK30. The archiver/librarian utility, MPLAB LIB30, was discussed in a previous chapter.

TABLE 12-1: AVAILABLE UTILITIES

Utility	Description
<code>pic30-ar</code> (MPLAB LIB30)	Creates, modifies and extracts files from archives (libraries.)
<code>pic30-bin2hex</code>	Converts a linked object file into an Intel® HEX file.
<code>pic30-nm</code>	Lists symbols from an object file.
<code>pic30-objdump</code>	Displays information about object files.
<code>pic30-ranlib</code>	Generates an index from the contents of an archive and stores it in the archive.
<code>pic30-strings</code>	Prints the printable character sequences.
<code>pic30-strip</code>	Discards all symbols from an object file.

NOTES:

Chapter 13. pic30-bin2hex Utility

13.1 INTRODUCTION

The binary-to-hexadecimal (`pic30-bin2hex`) utility converts binary COFF files (from MPLAB LINK30) to Intel HEX format files, suitable for loading into device programmers.

13.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

13.3 INPUT/OUTPUT FILES

- Input: COFF-formatted binary object files
- Output: Intel HEX files

13.4 SYNTAX

Command line syntax is:

```
pic30-bin2hex file
```

Example 13.1: hello.cof

Convert the absolute COFF executable file `hello.cof` to `hello.hex`

```
pic30-bin2hex hello.cof
```

After converting the binary file into Intel HEX format, `pic30-bin2hex` writes a table of program memory usage information to standard 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)
```

13.5 OPTIONS

`pic30-bin2hex` does not support any options.

NOTES:

Chapter 14. pic30-nm Utility

14.1 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.

14.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options
- Output Formats

14.3 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`.

14.4 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 ] [ -p | --no-sort ]
        [ -P | --portability ] [ -r | --reverse-sort ]
        [ -s --print-armap ] [ --size-sort ]
        [ -t radix | --radix=radix ] [ -V | --version ]
        [ OBJFILE... ]
```

14.5 OPTIONS

The long and short forms of options, shown in Table 14-1 as alternatives, are equivalent.

TABLE 14-1: 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 <i>format</i> --format= <i>format</i>	Use the output format <i>format</i> , which can be <i>bsd</i> , <i>sysv</i> or <i>posix</i> . The default is <i>bsd</i> . Only the first character of <i>format</i> 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 <i>pic30-nm</i> 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.
-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 <i>posix</i> .
-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 <i>pic30-ar</i> or <i>pic30-ranlib</i>) 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 <i>radix</i> --radix= <i>radix</i>	Use <i>radix</i> as the radix for printing the symbol values. It must be <i>d</i> for decimal, <i>o</i> for octal or <i>x</i> for hexadecimal.
-V --version	Show the version number of <i>pic30-nm</i> and exit.

14.6 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 14-2 shows the symbol types:

TABLE 14-2: 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:

Chapter 15. pic30-objdump Utility

15.1 INTRODUCTION

The `pic30-objdump` utility displays information about one or more object files. The options control what particular information to display.

15.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

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-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 ]
               [ --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.

15.5 OPTIONS

The long and short forms of options, shown in Table 15-1, 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 15-1: 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 <code>ls -l</code>). Besides the information you could list with <code>pic30-ar tv</code> , <code>pic30-objdump -a</code> 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 <code>ld</code> . However, some object file formats, such as <code>a.out</code> , do not store the starting address of the file segments. In those situations, although <code>ld</code> relocates the sections correctly, using <code>pic30-objdump -h</code> 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 <code>pic30-objdump</code> and exit.
-j <i>name</i> --section= <i>name</i>	Display information only for section <i>name</i> .
-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 <i>options</i> --disassembler- options= <i>options</i>	Pass target specific information to the disassembler. The dsPIC30F device supports the following target specific options: <code>symbolic</code> - Will perform symbolic disassembly.
--prefix-addresses	When disassembling, print the complete address on each line. This is the older disassembly format.

TABLE 15-1: pic30-objdump OPTIONS (CONTINUED)

Option	Function
-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:

Chapter 16. pic30-ranlib Utility

16.1 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-arnamap` 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., MPLAB LIB30 with the `-s` option).

16.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

16.3 INPUT/OUTPUT FILES

- Input: Archive files
- Output: Archive files

16.4 SYNTAX

Command line syntax is:

```
pic30-ranlib [-v | -V | --version] ARCHIVE
```

16.5 OPTIONS

The long and short forms of options, shown here as alternatives, are equivalent.

TABLE 16-1: pic30-ranlib OPTIONS

Option	Function
-v -V --version	Show the version number of <code>pic30-ranlib</code>

NOTES:

Chapter 17. pic30-strings Utility

17.1 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.

17.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

17.3 INPUT/OUTPUT FILES

- Input: Any files
- Output: Standard output

17.4 SYNTAX

Command line syntax is:

```
pic30-strings [-a | --all | -] [-f | --print-file-name]
               [--help] [-min-len | -n min-len | --bytes=min-len]
               [-t radix | --radix=radix] [-v | --version] FILE...
```

17.5 OPTIONS

The long and short forms of options, shown in Table 17-1 as alternatives, are equivalent.

TABLE 17-1: 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 <i>-min-len</i> characters long, instead of the default 4.
-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.

Chapter 18. pic30-strip Utility

18.1 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.

18.2 HIGHLIGHTS

Topics covered in this chapter are:

- Input/Output Files
- Syntax
- Options

18.3 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`.

18.4 SYNTAX

Command line syntax is:

```
pic30-strip [ -g | -S | --strip-debug ] [ --help ]  
           [ -K symbolname | --keep-symbol=symbolname ]  
           [ -N symbolname | --strip-symbol=symbolname ]  
           [ -o file ] [ -p | --preserve-dates ]  
           [ -R sectionname | --remove-section=sectionname ]  
           [ -s | --strip-all ] [--strip-unneeded]  
           [ -v | --verbose ] [ -V | --version ]  
           [ -x | --discard-all ] [ -X | --discard-locals ]  
OBJFILE...
```

18.5 OPTIONS

The long and short forms of options, shown in Table 18-1 as alternatives, are equivalent.

TABLE 18-1: pic30-strip OPTIONS

Option	Function
-g -S --strip-debug	Remove debugging symbols only.
--help	Show a summary of the options to <code>pic30-strip</code> and exit.
-K <i>symbolname</i> --keep-symbol= <i>symbolname</i>	Keep only symbol <i>symbolname</i> from the source file. This option may be given more than once.
-N <i>symbolname</i> --strip-symbol= <i>symbolname</i>	Remove symbol <i>symbolname</i> from the source file. This option may be given more than once, and may be combined with strip options other than -K.
-o <i>file</i>	Put the stripped output in <i>file</i> , rather than replacing the existing file. When this argument is used, only one OBJFILE argument may be specified.
-p --preserve-dates	Preserve the access and modification dates of the file.
-R <i>sectionname</i> --remove-section= <i>sectionname</i>	Remove any section named <i>sectionname</i> 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, <code>pic30-strip -v</code> lists all members of the archive.
-V --version	Show the version number for <code>pic30-strip</code> .
-x --discard-all	Remove non-global symbols.
-X --discard-locals	Remove compiler-generated local symbols. (These usually start with <code>_L</code> or <code>."</code> .)

Chapter 19. pic30-lm Utility

19.1 INTRODUCTION

The `pic30-lm` utility displays information about the MPLAB C30 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.

19.2 HIGHLIGHTS

Topics covered in this chapter are:

- Syntax
- Options

19.3 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 MPLAB C30 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 MPLAB C30 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.

19.4 OPTIONS

The `pic30-lm` options are shown below.

TABLE 19-1: PIC30-LM OPTIONS

Option	Function
-?	Displays usage information for <code>pic30-lm</code> . 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 MPLAB C30 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:



MPLAB® ASM30, MPLAB® LINK30 AND UTILITIES USER'S GUIDE

Part 5 – dsPIC30F Simulator

Chapter 20. Command Line Simulator	169
--	-----

Part
5

Simulator

NOTES:

Chapter 20. Command Line Simulator

20.1 INTRODUCTION

These tools include a basic command line simulator (`sim30.exe`) that may be used to test and debug dsPIC DSC programs when the MPLAB IDE simulator (MPLAB SIM30) is not available.

20.2 HIGHLIGHTS

Topics covered in this chapter are:

- Syntax
- Options

20.3 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 20-1: 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
```

20.4 OPTIONS

Table 20-1 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 20-1: SUPPORTED SIMULATOR COMMANDS

Option	Description
AF	AF [<frequency>] Alter or display the oscillator frequency. If the <code>frequency</code> 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 <code>ON</code> enables halt on error; specifying <code>OFF</code> 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 <code>ON</code> enables halt on warning; specifying <code>OFF</code> disables halt on warning. Omitting the parameter causes the current halt on warning status to be displayed.

TABLE 20-1: SUPPORTED SIMULATOR COMMANDS (CONTINUED)

Option	Description
IO	IO [stdin [stdout]] Enable simulated file I/O.
IF	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, <code>stdin</code> , <code>stdout</code> and <code>stderr</code> use the UART1 peripheral. By default, a stimulus file named <code>UartIn.txt</code> (for <code>stdin</code>) and a response file named <code>UartOut.txt</code> (for both <code>stdout</code> and <code>stderr</code>) are attached to the UART. Both files are opened in eight-bit binary format. The simulator looks for <code>UartIn.txt</code> 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 <code>UartOut.txt</code> in the current working directory. The default filenames <code>UartIn.txt</code> and <code>UartOut.txt</code> may be overridden by explicitly naming the files with the IO command's <code>stdin</code> and <code>stdout</code> parameters, respectively. The special name <code>null</code> 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. The following devices are supported. <code>dspic30f2010dspic30f2011dspic30f2012dspic30f3010</code> <code>dspic30f3011dspic30f3012dspic30f3012a2dspic30f3013</code> <code>dspic30f3013a2dspic30f3014dspic30f4011dspic30f4011a2</code> <code>dspic30f4012dspic30f4012a2dspic30f4013dspic30f4013a2</code> <code>dspic30f5011dspic30f5013dspic30f6010dspic30f6011</code> <code>dspic30f6012dspic30f6013dspic30f6014dspic30f6014a2</code>
LF	LF <filename> [displacement] Load File Registers from an Intel HEX file starting at offset <code>displacement</code> .
LP	LP <filename> [displacement] Load Program Memory from an Intel HEX file starting at the offset <code>displacement</code> .
LS	LS [<filename>] Load a Stimulus Control Language (SCL) file. If the <code>filename</code> 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 <code>filename</code> 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.

TABLE 20-1: SUPPORTED SIMULATOR COMMANDS (CONTINUED)

Option	Description
PS	PS <value> PC Set.
Q	Q Quit.
RC	RC Reset the simulation clock to cycle zero.
RP	RP Reset processor.
S	S Step.
VF	VF Verbose off.
VO	VO Verbose on.

Part 6 – Appendices

Appendix A. Assembler Errors/Warnings/Messages	175
Appendix B. Linker Errors/Warnings	189
Appendix C. MPASM™ Assembler Compatibility	197
Appendix D. MPLINK™ Linker Compatibility	207
Appendix E. MPLIB™ Librarian Compatibility	209
Appendix F. Useful Tables	211
Appendix G. GNU Free Documentation License	213

NOTES:

Appendix A. Assembler Errors/Warnings/Messages

A.1 INTRODUCTION

This appendix contains a descriptive list of errors, warnings and messages generated by MPLAB ASM30.

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 nbytes

A.4 ERRORS

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 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.

Alignment in CODE section must be at least 4 bytes.

The alignment value for the `.align` directive must be at least 4 bytes. Either no alignment was specified or an alignment less than 4 was specified. Modify the `.align` directive to have an alignment of at least 4.

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

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 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 operator on a symbol (name) that is not located in an executable or read-only 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` or read-only 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.

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 “Conditional Assembler Directives”** 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.

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 "Conditional Assembler Directives"** for more details.

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 "Conditional Assembler Directives"** 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.

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.

S

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 4.5.4 "Obtaining the Size of a Specific Section"](#) and [Section 4.5.5 "Obtaining the Starting Address of a Specific Section"](#) for details on the `.startof.` and `.sizeof.` operators.

T

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.

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 than eight. The maximum size allowed is eight.

.fillupper expects a constant positive byte value. 0xFF 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. 0xFF 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.

.In 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 than eight. The maximum size allowed is eight.

.pfillvalue expects a constant positive byte value. 0xFF 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.

.size pseudo-op used outside of .def/.endif ignored.

The specified directive is only allowed within a *.def/.endif* 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/.endif ignored.

The specified directive is only allowed within a *.def/.endif* 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/.endif ignored.

The specified directive is only allowed within a *.def/.endif* 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/.endif ignored.

The specified directive is only allowed within a *.def/.endif* 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/.endif ignored.

The specified directive is only allowed within a *.def/.endif* 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.

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.

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

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.

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 *class*

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 *.sc/* 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.

NOTES:

Appendix B. Linker Errors/Warnings

B.1 INTRODUCTION

This appendix contains a description list of errors and warnings generated by MPLAB LINK30.

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 *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.

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 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.

--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 runtime initialization of data, `--no-data-init` does not. Only one option can be used.

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.

MPLAB LINK30 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

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.

R

READONLY section *secname* exceeds 32K bytes (actual size = *num*).

The constant data table may not exceed the program memory page size that is implied by the PSVPAG register which is 32K bytes.

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.

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 MPLAB LINK30.

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 startup 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 MPLAB C30. If you are using a precompiled object or library file from another vendor, request an update that is compatible with the latest version of MPLAB C30.

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 MPLAB LINK30. The list of supported emulations follows the error message.

unrecognized -a option ‘*argument*.’

The `-a` option is not supported by dsPIC devices; so it is ignored.

unrecognized -assert option ‘*option*.’

The `-assert` option is not supported by dsPIC 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.

B.4 WARNINGS

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.

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 persistent data section (.pbss). 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.

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.

NOTES:

Appendix C. MPASM™ Assembler Compatibility

C.1 INTRODUCTION

This appendix is provided for users of the MPASM assembler, Microchip Technology's PICmicro® device assembler. MPLAB ASM30 (dsPIC DSC assembler) is not compatible with the MPASM assembler. This appendix provides details on the compatibility issues and provides examples and suggestions for migrating to the dsPIC assembler.

For more on the MPASM assembler, see the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014).

C.2 HIGHLIGHTS

Topics covered in this appendix are:

- Compatibility
- Examples
- Converting PIC18FXXX Assembly Code to dsPIC30FXXXX Assembly Code

C.3 COMPATIBILITY

Users migrating from MPASM assembler will face the following compatibility issues:

- Differences in the assembly language
- Differences in command line options
- Differences in directives

C.3.1 Differences in Assembly Language

The instruction set for dsPIC devices has been expanded to support the new functionality of the architecture. Please refer to individual dsPIC data sheets and Programmer's Reference for more details.

In addition, the following syntactical differences exist:

- A colon ':' must precede label definitions suffix.
- Directives must be preceded by a dot '.'.

C.3.2 Differences in Command Line Options

The MPLAB ASM30 command line is incompatible with the MPASM assembler command line. Table C-1 summarizes the command line incompatibilities.

TABLE C-1: COMMAND LINE INCOMPATIBILITIES

MPASM Assembler	MPLAB ASM30	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:** MPLAB ASM30 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 MPLAB ASM30 is decimal. See **Section 3.5.1.1 “Integers”**, for a complete description.
- 5:** MPLAB ASM30 listing files utilize the tab settings of the operating system.
- 6:** MPLAB ASM30 does not generate cross-reference files. See the MPLAB LINK30 section of this manual for information on creating cross-referenced files.

C.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 C-2 summarizes the assembler directive incompatibilities:

TABLE C-2: ASSEMBLER DIRECTIVE INCOMPATIBILITIES

MPASM Assembler	MPLAB ASM30	Description
<code>__BADRAM</code>	Not supported	Specify invalid RAM locations
<code>BANKISEL</code>	Not supported	Generate RAM bank selecting code for indirect addressing
<code>BANKSEL</code>	Not supported	Generate RAM bank selecting code
<code>CBLOCK</code>	Not supported	Define a block of constants
<code>CODE</code>	<code>.text</code>	Begins executable code section
<code>__CONFIG</code>	Not supported	Specify configuration bits
<code>CONSTANT</code>	<code>.equ (syntax)</code>	Declare symbol constant
<code>DA</code>	<code>.ascii (syntax)</code>	Store strings in program memory
<code>DATA</code>	Not supported	Create numeric and text data
<code>DB</code>	<code>.byte</code>	Declare data of one byte
<code>DE</code>	Not supported	Define EEPROM data
<code>#DEFINE</code>	<code>.macro (syntax)</code>	Define a text substitution label
<code>DT</code>	Not supported	Define table
<code>DW</code>	<code>.word</code>	Declare data of one word
<code>ELSE</code>	<code>.else</code>	Begin alternative assembly block to IF
<code>END</code>	<code>.end</code>	End program block
<code>ENDC</code>	Not supported	End an automatic constant block
<code>ENDIF</code>	<code>.endif</code>	End conditional assembly block
<code>ENDM</code>	<code>.endm (not equivalent)</code>	End a macro definition
<code>ENDW</code>	Not supported	End a while loop
<code>EQU</code>	<code>.equ (syntax)</code>	Define an assembly constant
<code>ERROR</code>	<code>.error</code>	Issue an error message
<code>ERRORLEVEL</code>	Not supported	Set error level
<code>EXITM</code>	Not supported	Exit from a macro
<code>EXPAND</code>	Not supported	Expand a macro listing
<code>EXTERN</code>	<code>.extern</code>	Declares an external label
<code>FILL</code>	<code>.fill (syntax)</code>	Fill memory
<code>GLOBAL</code>	<code>.global</code>	Exports a defined label
<code>IDATA</code>	<code>.data</code>	Begins initialized data section
<code>__IDLOCS</code>	Not supported	Specify ID locations
<code>IF</code>	<code>.if</code>	Begin conditionally assembled code block
<code>IFDEF</code>	<code>.ifdef</code>	Execute if symbol has been defined
<code>IFNDEF</code>	<code>.ifndef</code>	Execute if symbol has not been defined
<code>#INCLUDE</code>	<code>.include (syntax)</code>	Include additional source file

TABLE C-2: ASSEMBLER DIRECTIVE INCOMPATIBILITIES (CONTINUED)

MPASM Assembler	MPLAB ASM30	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

C.4 EXAMPLES

C.4.1 EQU vs .equ

In MPASM assembler, the `EQU` directive is used to define an assembler constant.

```
CORCONH EQU 0x45
```

In MPLAB ASM30, the `.equ` directive is used to define an assembler constant.

```
.equ CORCONH, 0x45
```

C.4.2 UDATA vs .bss

In MPASM assembler, the `UDATA` directive is used to begin an uninitialized data section.

```
UDATA
```

In MPLAB ASM30, the `.bss` directive is used to begin an uninitialized data section.

```
.bss
```


C.5 CONVERTING PIC18FXXX ASSEMBLY CODE TO dsPIC30FXXXX ASSEMBLY CODE

C.5.1 Direct Translations

Table C-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 C-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 C-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	Bitwise AND literal with WREG	WREG
ANDWF f,0,a	AND.b f,WREG	Bitwise AND file register contents with WREG	WREG
ANDWF f,1,a	AND.b f	Bitwise 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
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

Note 1: No direct translation.

2: No direct translation. See **Section C.5.2 “Emulation Model”**.

MPLAB® ASM30, MPLAB® LINK30 and Utilities User's Guide

TABLE C-4: INSTRUCTION SET COMPARISON (CONTINUED)

PIC18FXXX Instruction	dsPIC30FXXXX Instruction	Description	Result Location
BZ n	BRA Z, S1it16	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	Bitwise inclusive-or literal with WREG	WREG
IORWF f, 0, a	IOR.b f, WREG	Bitwise inclusive-or file register contents with WREG	WREG
IORWF f, 1, a	IOR.b f	Bitwise 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 C.5.2 “Emulation Model”**.

TABLE C-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	Bitwise exclusive-or WREG with literal	WREG
XORWF f, 0, a	XOR.b f, WREG	Bitwise exclusive-or WREG with contents of file register	WREG
XORWF f, 1, a	XOR.b f	Bitwise exclusive-or WREG with contents of file register	file register (f)

Note 1: No direct translation.

2: No direct translation. See **Section C.5.2 “Emulation Model”**.

C.5.2 Emulation Model

The PIC18FXXX parts can be modeled on a dsPIC30FXXXX by dedicating working registers to emulate PIC18FXXX special function registers.

TABLE C-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.

C.5.2.1 LFSR f,k

If k=0:

```
MOV #f, W6
```

If k=1:

```
MOV #f, W7
```

If k=2:

```
MOV #f, W8
```

C.5.2.2 MOVFF fs,fd

This is equivalent to the following sequence of instructions:

```
MOV fs, W1
```

```
MOV W1, fd
```

C.5.2.3 MULLW k

If k <= 0x1f:

```
MUL.UU W0, #k, W2
```

If k > 0x1f:

```
MOV #k, W1
```

```
MUL.UU W0, W1, W2
```

C.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
```

C.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

C.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]

C.5.2.7 TSTFSZ f,a

This instruction can be emulated using a two-instruction sequence:

MOV f

BRA Z, .+2

C.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++]

NOTES:

Appendix D. MPLINK™ Linker Compatibility

D.1 INTRODUCTION

This appendix contains information on compatibility with the MPLINK object linker, examples and recommendations for migrating to MPLAB LINK30 from MPLINK linker.

For more on the MPLINK linker, see the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014).

D.2 HIGHLIGHTS

Topics covered in this appendix are:

- Compatibility
- Migration to MPLAB LINK30

D.3 COMPATIBILITY

The MPLAB LINK30 command line is incompatible with the MPLINK command line. The following table summarizes the command line incompatibilities.

TABLE D-1: COMMAND LINE INCOMPATIBILITIES

MPLINK Linker	MPLAB LINK30	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	-Ll	Add directories to linker script search path
/n	Not supported (Note 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 (Note 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.

D.4 MIGRATION TO MPLAB LINK30

MPLAB LINK30 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:

Appendix E. MPLIB™ Librarian Compatibility

E.1 INTRODUCTION

This appendix contains information on compatibility with the MPLIB object librarian, examples and recommendations for migrating to MPLAB LIB30 from MPLIB librarian.

For more on the MPLIB librarian, see the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014).

E.2 HIGHLIGHTS

Topics covered in this appendix are:

- Compatibility
- Examples

E.3 COMPATIBILITY

The MPLAB LIB30 command line is incompatible with the MPLIB librarian command line. The following table summarizes the command line incompatibilities.

TABLE E-1: COMMAND LINE INCOMPATIBILITIES

MPLIB Librarian	MPLAB LIB30	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

E.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 MPLAB LIB30 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 MPLAB LIB30:

```
pic30-ar -t dsp.a
```

NOTES:

Appendix F. Useful Tables

F.1 ASCII CHARACTER SET

Least Significant Character	Most 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

F.2 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

Appendix G. 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.

0. 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. Secondly, 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.

1. 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.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, 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.

3. 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.

4. 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.

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.

5. 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”.

6. 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.

7. 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 in 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.

8. 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.

9. 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.

10. 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.

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 PICmicro 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 the 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 PICmicro 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.

Digital Signal Controller

A microcontroller device with digital signal processing capability, i.e., Microchip dsPIC 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 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.

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

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 PICmicro microcontroller devices, KeeLoq devices and Microchip memory devices.

MPLAB ASM30

Microchip's relocatable macro assembler for dsPIC30F digital signal controller devices.

MPLAB C1X

Refers to both the MPLAB C17 and MPLAB C18 C compilers from Microchip. MPLAB C17 is the C compiler for PIC17CXXX devices and MPLAB C18 is the C compiler for PIC18CXXX and PIC18FXXXX devices.

MPLAB C30

Microchip's C compiler for dsPIC30F digital signal controller devices.

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 LIB30

MPLAB LIB30 archiver/librarian is an object librarian for use with COFF object modules created using either MPLAB ASM30 or MPLAB C30 C compiler.

MPLAB LINK30

MPLAB LINK30 is an object linker for the Microchip MPLAB ASM30 assembler and the Microchip MPLAB C30 C compiler.

MPLAB SIM30

Microchip's simulator that works with MPLAB IDE in support of dsPIC DSC 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 C1X C compilers.

MPLINK Object Linker

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip MPLAB C17 or C18 C compilers. 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.

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.

PICmicro MCUs

PICmicro 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 PICmicro 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. MPLAB ASM30 currently knows how to RELAX a CALL instruction into an RCALL instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

Relocatable

An object 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 or 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 PICmicro 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.

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 MPLAB C30, warning messages report the source file name and line number, but include the text 'warning:' to distinguish them from error messages.

Watchdog Timer

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

WDT

See Watchdog Timer.

NOTES:

Index

Symbols

\$	43
-(-)	74
.	43
.abort	61
.align	48, 54, 123
.apline	61
.ascii	49
.asciz	50
.bss	46, 52
.bss section	77, 89, 93, 106, 126
.byte	50
.comm	52
.comm symbol, length	52
.const section	95, 123, 128, 129
.data	46
.data section	77, 89, 93, 126
.dconst section	126
.def	62
.dim	62
.dinit section	89, 92, 127, 129
.double	50
.eedata section	90
.eject	57
.else	58
.elseif	58
.end	61
.endif	62
.endif	58
.endm	60
.endr	59, 60
.equ	41, 53
.equiv	41, 53
.err	58
.error	58
.exitm	59
.extern	52
.fail	61
.file	62
.fill	54
.fillupper	48
.fillvalue	48
.fixed	51
.float	51
.global	53
.globl	53
.handle	78
.handle section	89, 120, 125
.hword	51
.icd section	91
.ident	61

.if	58
.ifdef	58
.ifndef	58
.ifnotdef	58
.include	28, 29, 61
.int	51
.irp	59
.irpc	59
.lcomm	53
.lib* section	89
.libc section	89
.libdsp section	89
.libm section	89
.libperi section	89
.line	62
.list	57
.ln	61
.loc	61
.long	52
.macro	60
.nbss section	92, 126
.ndata section	92, 126
.ndconst section	126
.nolist	57
.org	55
.palign	54
.pbss section	78, 92, 126, 129
.pbyte	50, 122, 129
.pfill	55
.pfillvalue	49
.porg	55
.print	61
.psize	57
.pskip	56
.pspace	56
.purgem	60
.pword	52, 129
.rept	60
.reset section	88
.sbttl	57
.scl	62
.section name	47
.set	41, 43, 53
.short	52
.single	51
.size	62
.sizeof	39
.skip	56
.sleb128	62
.space	56
.startof	39

MPLAB® ASM30, MPLAB® LINK30 and Utilities User's Guide

.string	52	MIN	116
.struct	56	NEXT	117
.tag	63	SIZEOF	117
.text	48	C	
.text section	77, 88, 126	Character Constants	34
.title	57	Characters	34
.type	63	--check-sections	79
.val	63	Command-Line Information	
.weak	53, 124	Linker Scripts	84
.word	52	Command-Line Interface	
.xbss section	91, 126	MPLAB ASM30	15
.xdata section	91, 126	MPLAB LIB30	141
.ybss section	94, 126	MPLAB LINK30	73
.ydata section	94, 126	Simulator	169
__reset	128	Comments	32, 98
_main	128	Computing Absolute Addresses	120
A		Condition Codes	31
-A	74	Conditional Assembly Directives	
-a	16	.else	58
a.out	11, 27, 76	.elseif	58
-a=file	24	.endif	58
-ac	17	.err	58
Accessing Data	37	.error	58
Accumulator Select	32	.if	58
-ad	19	.ifdef	58
ADDR	115	.ifndef	58
-ah	20	.ifnotdef	58
-ai	21	Configuration Region	87
Aiwt Region	86	Constant Data	37
-al	21	Constants	112
ALIGN	116	Fixed-Point Numbers	33
Allocatable Section	102	Floating-Point Numbers	33
Allocating Memory	120	Integer	33
Allocating Unmapped Sections	124	Constants, Numeric	32
-am	21	COPY	108
-an	23	Creating Special Sections	120
ar utility	139	--cref	82
--architecture	74	crt0	70, 89, 128, 131
Archiver	139	crt1	128
Arguments	32	Current Address	43
-as	24	Custom Linker Script	98
ASCII Character Set	211	Customer Notification Service	6
Assembler Directives	30	Customer Support	6
Assembler Source	12	D	
ASSERT	111	-d	74
Assigning Output Sections to Regions	123	Data Flash Memory	90
Assigning Values	100	Data Initialization Template	89, 127
B		Data Memory	37, 121
Base Memory Addresses	87	Data Region	85
bin2hex utility	149	--data-init	78
Binary File	69	-dc	74
BLOCK	116	Debug Information Directives	
Building the Output File	121	.def	62
Built-in Functions	115	.dim	62
ADDR	115	.endif	62
ALIGN	116	.file	62
BLOCK	116	.line	62
DEFINED	116	.scl	62
LOADADDR	116	.size	62
MAX	116	.sleb128	62

.tag	63
.type	63
.val	63
Declare Symbols Directives	
.bss	52
.comm	52
.extern	52
.global	53
.globl	53
.lcomm	53
.weak	53
Define Symbols Directives	
.equ	53
.equiv	53
.set	53
DEFINED	116
--defsym	28, 74
Destination Select	31
Directive	30
Directives	
Conditional	58
Debug Information	62
Declare Symbols	52
Define Symbols	53
Fill	48
Initialization	49
Location Counter	54
Miscellaneous	61
MPLAB ASM30	45
Output Listing	57
Section	46
Substitution/Expansion	59
--discard-all	75
--discard-locals	75
Document	
Conventions	3
Numbering Conventions	4
Updates	4
Document Layout	1
DOT Symbol	43
Dot Variable	113, 131
-dp	74
DSECT	108
E	
EEDATA Memory Region	87
Empty Expressions	35
--end-group	74
ENTRY	111
Entry Point	84
Escape Characters	34
Evaluation	114
EXCLUDE_FILE	104
Executable Section	102
Expression Syntax and Operation	
MPLAB ASM30	35
Expressions	35
Expressions, Empty	35
Expressions, Integer	35
EXTERN	111

F	
--fatal-warnings	26
File Commands, Linker Scripts	
GROUP	99
INCLUDE	99
INPUT	99
OUTPUT	99
SEARCH_DIR	99
STARTUP	99
File Extensions	
Assembler	10
Linker	68
File Registers	31
Files	
Library	68
Linker Output	69
Linker Script	68
Listing	11
Map	69
Object	11, 68
Source	10
Fill Directives	
.fillupper	48
.fillvalue	48
.pfillvalue	49
Fixed-Point Numbers	33
Floating-Point Numbers	33
FORCE_COMMON_ALLOCATION	111
--force-exe-suffix	75
G	
Getting a Grip	125
Global Symbols	124
GROUP	99
H	
handle()	39, 120, 121, 125
Handles	125
--handles	78
Header	11
--heap	79
Heap Allocation	132
--help	26, 79
Hexadecimal to Decimal Conversion	212
High-level Source	12, 20
I	
-I	28
-i	76
ICD Memory	91
INCLUDE	99
Infix Operators	36
INFO	108
Informational Output Options, Assembler	
--fatal-warnings	26
--help	26
-J	26
--no-warn	26
--target-help	26
-v	26
--verbose	26
--version	26

MPLAB® ASM30, MPLAB® LINK30 and Utilities User's Guide

-W	26	Label	30, 42, 121, 125
--warn	26	LENGTH	102
Informational Output Options, Linker		Librarian	139
--check-sections	79	--library	75
--help	79	Library Files	68
--no-check-sections	79	--library-path	75
--no-warn-mismatch	80	license manager utility	165
-t	80	Link Map Options, Linker	
--trace	80	--cref	82
--trace-symbol	80	-M	82
-V	80	-Map	82
-v	80	--print-map	82
--verbose	80	Linker Allocation	122
--version	80	Linker Output File	69
--warn-common	80	Linker Processing	119
--warn-once	81	Linker Script File	68
--warn-section-align	81	Linker Scripts	83
-y	80	Command Language	98
Initialized Section	102	Command-Line Information	84
Initialization Directives		Concepts	98
.ascii	49	Contents	84
.asciz	50	Custom	98
.byte	50	Expressions	112
.double	50	File Commands	99
.fixed	51	Other Commands	111
.float	51	Listing Files	11
.hword	51	Listing Output Options, Assembler	16
.int	51	-a=file	24
.long	52	-ac	17
.pbyte	50	-ad	19
.pword	52	-ah	20
.short	52	-ai	21
.single	51	-al	21
.string	52	-am	21
.word	52	-an	23, 24
Initialized Data	126	--listing-cont-lines	25
Initialized Section	102	--listing-lhs-width	25
INPUT	99	--listing-lhs-width2	25
Input Section		--listing-rhs-width	25
Common Symbols	106	--listing-cont-lines	25
Example	106	--listing-lhs-width	25
Wildcard Patterns	105	--listing-lhs-width2	25
Input/Output Section Map	87	--listing-rhs-width	25
Integer Expressions	35	Literals	31
Integers	33	LMA	98, 108, 116
Internal Preprocessor	29	Load Memory Address	98, 108, 116
Internet Address	5	LOADADDR	116
Interrupt		Loading Input Files	119
Handlers	132	Local Symbols	42
Vector Tables	96, 132	Location Counter	89, 113
Invert Sense	102	Location Counter Directives	
Ivt Region	86	.align	54
J		.fill	54
-J	26	.org	55
K		.palign	54
K Suffix	112	.pfill	55
--keep-locals	27	.porg	55
L		.pskip	56
-L	27, 75	.pspace	56
-l	75	.skip	56

.space	56	Command-Line Interface	73
.struct	56	Linker Processing	119
M		Linker Scripts	83
-M	82	Overview	67
M Suffix	112	N	
-Map	82	NEXT	117
Map File	69	nm utility	151
Mapping Sections	122	--no-check-sections	79
MAX	116	NOCROSSREFS	112
-MD	27	--no-data-init	78
Memory Addressing	121	--no-handles	79
MEMORY Command	101	--noinhibit-exec	75
!	102	--no-keep-memory	75
A	102	NOLOAD	93, 108
I	102	--no-pack-data	79
L	102	--no-relax	27
R	102	--no-undefined	77
W	102	--no-warn	26
X	102	--no-warn-mismatch	80
Memory Region Information	85	Numeric Constants	32
Microchip Internet Web Site	5	O	
MIN	116	-o	27, 76
Miscellaneous Directives		objdump utility	155
.abort	61	Object Files	11, 68
.apline	61	Operands	31
.end	61	Operators	36, 114
.fail	61	Infix	36
.include	61	Prefix	36
.indent	61	Optimize	75
.In	61	Options, Archiver/Librarian	
.loc	61	d	141
.print	61	m	141
Mnemonic	30	p	141
Modification Options, Archiver/Librarian		q	141
a	142	r	141
b	142	t	141
c	142	x	141
f	142	Options, Assembler	
i	142	Informational Output	26
l	142	Listing Output	16
N	142	Other	28
o	142	Output File Creation	27
P	142	Options, Linker	
S	142	Informational Output	79
s	142	Link Map Output	82
u	142	Output File Creation	74
V	142	Runtime Initialization	78
v	142	Options, pic30-lm	
MPLAB ASM30		-?	165
Command-Line Interface	15	-u	165
Directives	45	Options, pic30-nm	
Expression Syntax and Operation	35	-A	152
Overview	9	-a	152
Symbols	41	-B	152
Syntax	29	--debug-syms	152
MPLAB ICD 2 Debugger Memory	91	--defined-only	152
MPLAB LIB30	139	--extern-only	152
Command-Line Interface	141	-f	152
Scripts	143	--format=	152
MPLAB LINK30		-g	152

--help	152	--start-address=	157
-l	152	--stop-address=	157
--line-numbers	152	--syms	157
-n	152	-t	157
--no-sort	152	-V	157
--numeric-sort	152	--version	157
-o	152	-w	157
-P	152	--wide	157
-p	152	-x	157
--portability	152	-Z	157
--print-armap	152	Options, pic30-ranlib	
--print-file-name	152	-V	159
-r	152	-v	159
--radix=	152	--version	159
--reverse-sort	152	Options, pic30-strings	
-s	152	-	162
--size-sort	152	-a	162
-t	152	--all	162
-u	152	--bytes=	162
--undefined-only	152	-f	162
-V	152	--help	162
-v	152	-n	162
--version	152	--print-file-name	162
Options, pic30-objdump		--radix=	162
-a	156	-t	162
--all-header	157	-v	162
--archive-header	156	--version	162
-D	156	Options, pic30-strip	
-d	156	--discard-all	164
--debugging	156	--discard-locals	164
--disassemble	156	-g	164
--disassemble-all	156	--help	164
--disassembler-options=	156	-K	164
--disassemble-zeroes	157	--keep-symbol=	164
-EB	156	-N	164
-EL	156	-o	164
--endian=	156	-p	164
-f	156	--preserve-dates	164
--file-header	156	-R	164
--file-start-context	156	--remove-section=	164
--full-contents	157	-S	164
-g	156	-s	164
-H	156	--strip-all	164
-h	156	--strip-debug	164
--header	156	--strip-symbol=	164
--help	156	--strip-unneeded	164
-j	156	-V	164
-l	156	-v	164
--line-numbers	156	--verbose	164
-M	156	--version	164
--no-show-raw-insn	157	-X	164
--prefix-addresses	156	-x	164
-r	157	Options, Simulator	
--reloc	157	AF	170
-S	157	BC	170
-s	157	BS	170
--section=	156	DA	170
--section-header	156	DB	170
--show-raw-insn	157	DC	170
--source	157	DF	170

DH	170	-dc	74
DM	170	--defsym	74
DP	170	--discard-all	75
DS	170	--discard-locals	75
DW	170	-dp	74
E	170	--end-group	74
FC	170	--force-exe-suffix	75
FS	170	-i	76
H	170	-L	75
HE	170	-l	75
HW	170	--library	75
IF	171	--library-path	75
IO	171	--noinhibit-exec	75
LC	171	--no-keep-memory	75
LD	171	--no-undefined	77
LF	171	-o	76
LP	171	--output	76
LS	171	-r	76
MC	171	--relocateable	76
MS	171	--retain-symbols-file	76
PS	172	-S	77
Q	172	-s	77
RC	172	--script	76
RP	172	--sort-common	76
S	172	--start-group	74
VF	172	--strip-all	77
VO	172	--strip-debug	77
ORG	102	-T	76
ORIGIN	102	-Tbss	77
Other Linker Script Commands		-Tdata	77
ASSERT	111	-Ttext	77
ENTRY	111	-u	77
EXTERN	111	--undefined	77
FORCE_COMMON_ALLOCATION	111	-Ur	76
NOCROSSREFS	112	--wrap	78
OUTPUT_ARCH	112	-X	75
OUTPUT_FORMAT	112	-x	75
TARGET	112	Output Formats, pic30-nm	
Other Options, Assembler		?	153
--defsym	28	A	153
-l	28	B	153
-p	28	C	153
--processor	28	D	153
OUTPUT	99	N	153
--output	76	R	153
Output File Creation Options, Assembler		T	153
--keep-locals	27	U	153
-L	27	V	153
-MD	27	W	153
--no-relax	27	Output Listing Directives	
-o	27	.eject	57
-R	27	.list	57
--relax	27	.nolist	57
-Z	27	.psize	57
Output File Format	84	.sbttl	57
Output File Options, Linker		.title	57
-(-)	74	Output Section	
-A	74	.const	95
--architecture	74	.reset	88
-d	74	.text	88

MPLAB® ASM30, MPLAB® LINK30 and Utilities User's Guide

Address	104	psvoffset()	38
Attributes	108	psvpage()	38
Data	107	R	
Description	103	-R	27
Discarding	107	-r	76
Fill	109	Range Checking	95
LMA	108	ranlib utility	159
Region	109	Read/Write Section	102
Type	108	Read-Only Data	129
COPY	108	Read-Only Section	102
DSECT	108	References	4
INFO	108	Registers	31
NOLOAD	108	Relative Branches	27
OVERLAY	108	Relative Calls	27
Output Sections in		--relax	27
Configuration Memory	90	relocatable	11
General Data Memory	93	--relocateable	76
Near Data Memory	92	Reserved Names	41
X Data Memory	91	RESET	132
Y Data Memory	94	Reset Region	86
OUTPUT_ARCH	112	Resolving Symbols	120
OUTPUT_FORMAT	112	--retain-symbols-file	76
OVERLAY	108	Runtime Initialization Options, Linker	
Overlay Description	110	--data-init	78
Overview		--handles	78
MPLAB ASM30	9	--heap	79
MPLAB LINK30	67	--no-data-init	78
P		--no-handles	79
-p	28	--no-pack-data	79
--pack-data	79	--pack-data	79
paddr()	39	--stack	79
Page Size	38	Runtime Library Support	128
Persistent Data	92, 126, 129	S	
pic30	156, 164	-S	77
pic30-ar utility	139	-s	77
pic30-bin2hex utility	149	--script	76
pic30-lm utility	165	Scripts	
pic30-nm utility	151	MPLAB LIB30	143
pic30-objdump utility	155	Scripts, Archiver/Librarian	
pic30-ranlib utility	159	ADDLIB	143
pic30-strings utility	161	ADDMOD	143
pic30-strip utility	163	CLEAR	143
Pointer	37	CREATE	143, 144
Precedence	114	DELETE	144
precedence	36	DIRECTORY	144
Prefix Operators	36	END	144
Preprocessor, Internal	29	EXTRACT	144
--print-map	82	LIST	144
Process Flow		OPEN	143, 144
Assembler	9	REPLACE	144
Linker	67	SAVE	143, 144
MPLAB LIB30	140	VERBOSE	144
--processor	28	SEARCH_DIR	99
Program Address	39	Section Directives	
Program Memory	37, 121	.bss	46
Program Region	86	.data	46
Program Space Visibility		.section name	47
Window	37, 38, 95, 110, 122, 123, 128	.text	48
PROVIDE	101	Section of an Expression	115
PSV Window	37, 38, 47, 95, 110, 122, 123, 128	SECTIONS Command	103

SFR Addresses	97	pic30-strings	161
SFRs	85, 97, 119	pic30-strip	163
sim30	84, 169	Simulator	169
Simple Assignments	100	T	
Simulator Command-Line Interface	169	-T	76
SIZEOF	117	-t	80
--sort-common	76	Table Access Instructions	121
Source Code	30	TARGET	112
Source Files	10	--target-help	26
Special Function Registers	85, 97, 119	tbloffset()	38, 121
Special Operators	37	tblpage()	38, 121
.sizeof	37	-Tbss	77
.startof	37	-Tdata	77
handle	37	Title	57
paddr	37	Title Line	11
psvoffset	37	--trace	80
psvpage	37	--trace-symbol	80
tbloffset	37	Troubleshooting	5
tblpage	37	-Ttext	77
SPLIM	128, 131	U	
--stack	79	-u	77
Stack Allocation	131	--undefined	77
Stack Pointer	128, 131	-Ur	76
Stack Pointer Limit Register	128, 131	User-Defined Section in Data Memory	93
standard	134	User-Defined Section in Program Memory	89
Standard Data Section Names	126	Utilities	145
--start-group	74	V	
Starting Address	39	-V	80
STARTUP	99	-v	26, 80
Startup Code	127	--verbose	26, 80
Startup Module	128	--version	26, 80
Statement Format	30	Virtual Memory Address	98, 108
Strings	34	VMA	98, 108
strings utility	161	W	
strip utility	163	-W	26
--strip-all	77	W15	128, 131
--strip-debug	77	--warn	26
Substitution/Expansion Directives		--warn-common	80
.endm	60	--warn-once	81
.endr	59, 60	--warn-section-align	81
.exitm	59	Watchdog Timer, Disabling	90
.irpc	59	Weak Symbols	124
.macro	60	Whitespace	30
.purgem	60	--wrap	78
.rept	60	WWW Address	5
irp	59	X	
Subtitle	11, 57	-X	75
supported	170	-x	75
Symbol Names	112	X Data	91, 95, 126
Symbol Table	12, 24, 62, 63, 99, 116	Y	
Symbols	41	-y	80
MPLAB ASM30	41	Y Data	94, 126
Syntax		Z	
Archiver/Librarian	141	-Z	27
Assembler	15, 29		
Linker	73		
pic30-bin2hex	149		
pic30-nm	151		
pic30-objdump	155		
pic30-ranlib	159		



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: 480-792-7627
Web Address: <http://www.microchip.com>

Atlanta

3780 Mansell Road, Suite 130
Alpharetta, GA 30022
Tel: 770-640-0034
Fax: 770-640-0307

Boston

2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848
Fax: 978-692-3821

Chicago

333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071
Fax: 630-285-0075

Dallas

4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423
Fax: 972-818-2924

Detroit

Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo

2767 S. Albright Road
Kokomo, IN 46902
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles

18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888
Fax: 949-263-1338

Phoenix

2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966
Fax: 480-792-4338

San Jose

1300 Terra Bella Avenue
Mountain View, CA 94043
Tel: 650-215-1444

Toronto

6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Australia

Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing

Unit 706B
Wan Tai Bei Hai Bldg.
No. 6 Chaoyangmen Bei Str.
Beijing, 100027, China
Tel: 86-10-85282100
Fax: 86-10-85282104

China - Chengdu

Rm. 2401-2402, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-86766200
Fax: 86-28-86766599

China - Fuzhou

Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506
Fax: 86-591-7503521

China - Hong Kong SAR

Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Shanghai

Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6275-5700
Fax: 86-21-6275-5060

China - Shenzhen

Rm. 1812, 18/F, Building A, United Plaza
No. 5022 Binhe Road, Futian District
Shenzhen 518033, China
Tel: 86-755-82901380
Fax: 86-755-8295-1393

China - Shunde

Room 401, Hongjian Building
No. 2 Fengxiangnan Road, Ronggui Town
Shunde City, Guangdong 528303, China
Tel: 86-765-8395507 Fax: 86-765-8395571

China - Qingdao

Rm. B505A, Fullhope Plaza,
No. 12 Hong Kong Central Rd.
Qingdao 266071, China
Tel: 86-532-5027355 Fax: 86-532-5027205

India

Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaughnessy Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan

Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471-6166 Fax: 81-45-471-6122

Korea

168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5932 or
82-2-558-5934

Singapore

200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-6334-8870 Fax: 65-6334-8850

Taiwan

Kaohsiung Branch
30F - 1 No. 8
Min Chuan 2nd Road
Kaohsiung 806, Taiwan
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan

Taiwan Branch
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Austria

Durisolstrasse 2
A-4600 Wels
Austria
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

Denmark

Regus Business Centre
Lautrup høj 1-3
Ballerup DK-2750 Denmark
Tel: 45-4420-9895 Fax: 45-4420-9910

France

Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - 1er Etage
91300 Massy, France
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany

Steinheilstrasse 10
D-85737 Ismaning, Germany
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy

Via Quasimodo, 12
20025 Legnano (MI)
Milan, Italy
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands

P. A. De Biesbosch 14
NL-5152 SC Drunen, Netherlands
Tel: 31-416-690399
Fax: 31-416-690340

United Kingdom

505 Eskdale Road
Wokingham
Berkshire, England RG41 5TU
Tel: 44-118-921-5869
Fax: 44-118-921-5820

11/24/03